D2.2: Technical report on connecting CAV control logic and CAV simulator

Version: 1.0
Date: 01.12.2017

Authors: Mohamed Cherif Rahal
Steve Pechberti
Bart Heijke
Peter Sukennik

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

Document Control Page

Title	D2.2 Technical Report on connecting CAV control logic and CAV simulator
Creator	Mohamed Cherif Rahal
Editor	
Brief Description	
Contributors	Mohamed Cherif Rahal (VEDECOM) Steve Pechberti (VEDECOM) Bart Heijke (TASS International) Peter Sukennik (PTV Group)
Reviewer	Bernard Gyergyay (Rupprecht Consult) Syrus Gomari (Rupprecht Consult) Charlotte Fléchon (PTV Group)
Type (Deliverable/Milestone)	Deliverable
Creation date	01.12.2017
Version number	1.0
Version date	
Last modified by	
Rights	
Audience	☐ Internal ☒ Public ☐ Restricted, access granted to: EU Commission
Action requested	☐ To be revised by Partners involved in the preparation of the Deliverable ☐ For approval of the WP Manager ☐ For approval of the Internal Reviewer (if required) ☐ For approval of the Project Co-ordinator
Deadline for approval	

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

Version	Date	Modified by	Comments
0			

Table of contents

1 General introduction and context 5
2 Simulation 5
2.1 Presentation of VISSIM 6
2.2 Presentation of PreScan 7
2.2.1 Simulation in PreScan with Simulink 7
3 Presentation of control Logic (CL) 9
3.1 Renault CL 9
3.2 VEDECOM CL 11
3.2.1 Overview 11
4 Connection between VEDECOM CL and PreScan 12
4.1 Description 12
4.1.1 Adaptation of the vehicle model in PreScan 13
4.2 Results 13
5 Connection between Renault CL and PreScan 14
6 Connection between PreScan and Vissim 15
6.1 PTV Vissim - Driving simulator interface 15
6.1.1 Architecture 16
6.2 New development in PTV Vissim 18
7 Next steps 19
7.1.1 VEDECOM CL to VISSIM 19
7.2 Simulations & analysis 20
8 Conclusion 20
9 Development partners for D2.2 21
10 Appendices 21
10.1 PTV Vissim API - Driver Model Interface 22
10.2 PTV Vissim API - Driving Simulator Interface 37

1 General introduction and context

Within CoEXist, work package 2 (WP2) will develop and validate transport modelling software extensions aimed at the analysis of impacts arising from the coexistence of conventional vehicles (CVs) and connected and automated vehicles (CAVs), and the impact on safety and traffic efficiency of a step-wise introduction of CAVs.

CoEXist will overcome the technical limitations that currently exist in traffic simulation and transport modelling, by developing default driver behaviour parameter values for CAVs. This enables transport planners without access to a CAV control logic (e.g. transport planners in road authorities) to model CAVs with realistic default values or with own assumptions about the driving behaviour of CAVs, independent of a CAV manufacturer. The default values are validated through a validation process with data collected from three CAVs that are operated by TASS[1], which were driven on the DICTM test track in Helmond, Netherlands. Deliverable D2.6 (due April 2018) will further elaborate on this within a “technical report on data collection and validation process”.

Deliverable D2.2 builds on the brief note (deliverable D2.1) on the proof of technology of a real-time closed loop connection (software) between the CAV control logics, CAV simulator (PreScan[2]) and the traffic simulator (Vissim[3]). D2.2 describes how the CAV behaviour parameter sets are extracted from the connection with the control logics for sub-models (incl. lane change, car following and lateral positioning, reaction on signals) to preset and define the CAV behaviour parameters in PTV’s microscopic traffic simulation (Vissim).

This technical report aims to provide guidance for other CAV developers and traffic modellers to replicate the same process.

2 Simulation

The initial simulations are performed in PreScan. Here all details concerning the ego vehicle, such as vehicle dynamics, can be specified. Furthermore, all sensors that are attached to the ego vehicle can be designed in detail. A wide range of sensor models is available, from simple ground-truth sensors to complex physics-based sensors.

The PreScan graphical user interface (GUI) also provides a large amount of options for environment design. Infrastructure and nature elements can be added from the library and simulated according to the wishes of the user. Also light and weather settings can be changed accordingly.

Large traffic flows can be simulated in PreScan with the Vissim plugin. With this plugin, a co-simulation is set up between PreScan and Vissim. The location and orientation of PreScan controlled vehicles and

[1] TASS International: https://www.tassinternational.com/
[2] For further information about PreScan: www.tassinternational.com/prescan
[3] For further information about PTV Vissim: http://vision-traffic.ptvgroup.com/en-us/products/ptv-vissim/

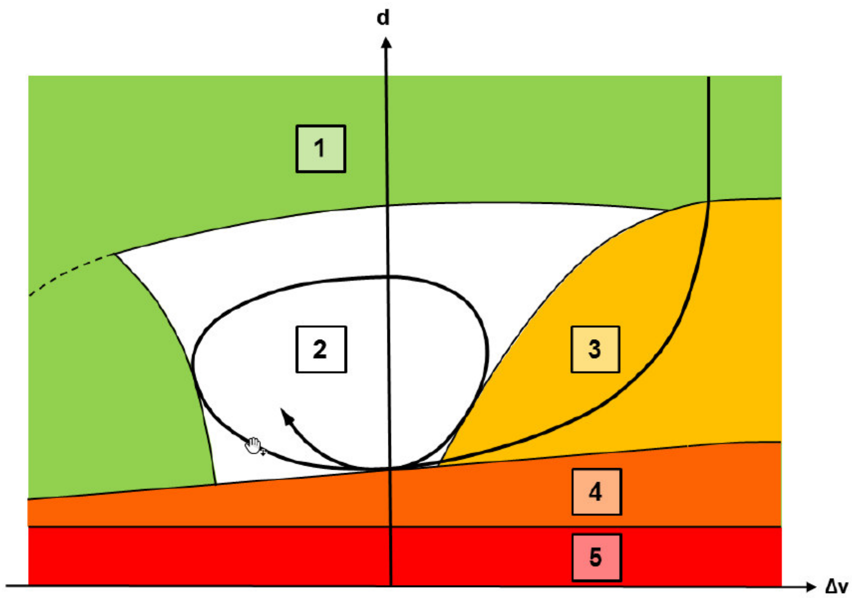
This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 723201

Vissim controlled vehicles are communicated between the two software programs. In this way, reactive traffic can be added and adjusted easily.

2.1 Presentation of VISSIM

PTV Vissim is the leading microscopic simulation program for modelling multimodal transport operations and belongs to the Vision Traffic Suite software of PTV. Realistic and accurate in every detail, Vissim creates the best conditions to test different traffic scenarios before their realization. Vissim is now being used worldwide by the public sector, consulting firms and universities. It is a microscopic, time step oriented, and behaviour-based simulation tool for modelling urban and rural traffic as well as pedestrian flows. For pedestrian flows, PTV Viswalk is available as a module or standalone application. PTV Viswalk is the leading software for pedestrian simulation. Based on the social force model of Prof. Dr. Dirk Helbing, it reproduces the human walking behaviour realistically and reliably. This software solution with powerful features is used when it is necessary to simulate and analyse pedestrian flows, be it outdoors or indoors.

Vissim is based on a traffic flow model and light signal control. The traffic flow model is based on a car-following model (see Figure 1) (for the modelling of driving in a stream on a single lane) and on a lane changing model.



Car following model (according to: Wiedemann 1974)

Legend

Axes: **d**: Distance, **Δv**: Change in speed	**3**: Approaching state
1: "Free flow" state	**4**: Braking state
2: Following state	**5**: Collision state

Figure 1 Car following model in PTV Vissim

Vehicles are moving in the network using a traffic flow model. The quality of the traffic flow model is essential for the quality of the simulation. In contrast to simpler models in which a largely constant speed and a deterministic car following logic are provided, Vissim uses the psycho-physical perception model developed by Wiedemann (1974) as shown in Figure 1. The basic concept of this model is that the driver

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

of a faster moving vehicle starts to decelerate as he reaches his individual perception threshold to a slower moving vehicle. Since he cannot exactly determine the speed of that vehicle, his speed will fall below that vehicle's speed until he starts to slightly accelerate again after reaching another perception threshold. There is a slight and steady acceleration and deceleration. The different driver behaviour is taken into consideration with distribution functions of the speed and distance behaviour.

2.2 Presentation of PreScan

PreScan is a simulation platform consisting of a GUI-based preprocessor to define scenarios and a run-time environment to execute them. The engineer's prime interface for making and testing algorithms includes MATLAB and Simulink. PreScan can be used from model-based controller design (MIL) to real-time tests with software-in-the-loop (SIL) and hardware-in-the-loop (HIL) systems. PreScan can operate in open-loop & closed-loop, and offline & online mode. It is an open software platform which has flexible interfaces to link to 3rd party vehicle dynamics model (e.g. CarSIM, dSPACE ASM, etc) and 3rd party HIL simulators/hardware (e.g. ETAS, dSPACE, Vector).

PreScan comprises several modules that together provide everything an automated driving system developer needs. An intuitive graphical user interface allows you to build your scenario and model your sensors, while the Matlab/Simulink interface enables you to add a control system. This interface can also be used to import existing Matlab/Simulink models such as vehicle dynamics models. When running your experiment, the visualisation viewer provides a realistic 3D representation of the scenario (see Figure 2). Optionally, tools such as dSPACE ControlDesk and NI LabVIEW can be used for activities such as 'data acquisition' and 'test automation'.

Figure 2: PreScan 3D visualisation viewer

2.2.1 Simulation in PreScan with Simulink

Scenario building and simulating in PreScan are performed in four steps as shown in Figure 3:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

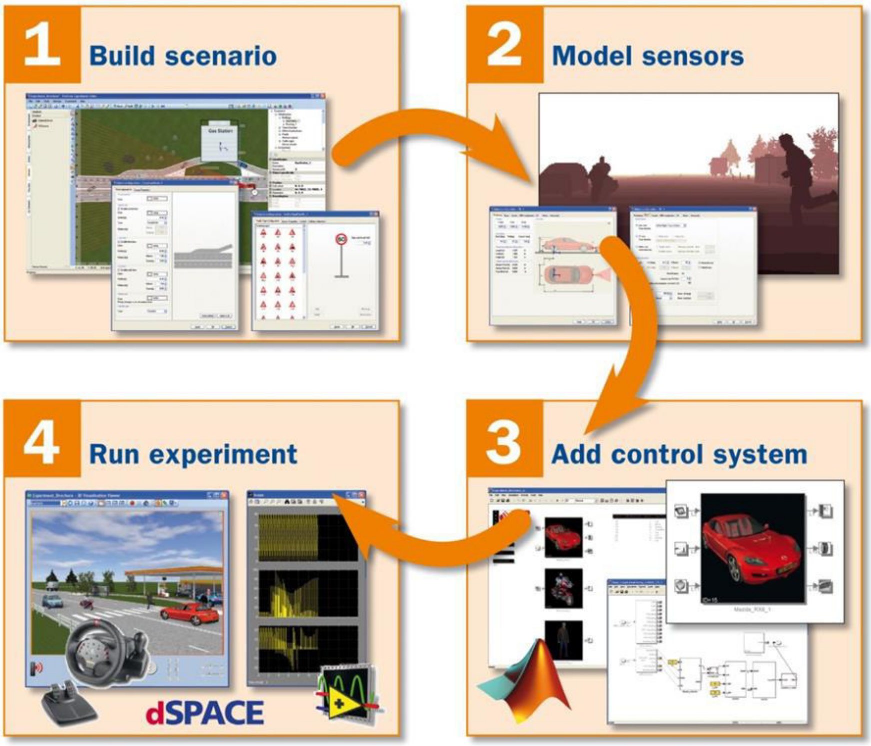


Figure 3: PreScan experiment process

1. **Build scenario**
   A dedicated preprocessor (GUI) allows users to build and modify traffic scenarios within minutes using a database of road sections, infrastructure components (trees, buildings, traffic signs), actors (cars, trucks, bikes and pedestrians), weather conditions (such as rain, snow and fog) and light sources (such as the sun, headlights and lampposts). Representations of real roads can be quickly made by reading in information from OpenStreetMap, Google Earth, Google 3D Warehouse and/or a GPS navigation device.

2. **Model sensors**
   Vehicle models can be equipped with different sensor types, including radar, laser, camera, ultrasound, infrared, GPS and antennas for vehicle-to-X (V2X) communication. Sensor design and benchmarking is facilitated by easy exchange and modification of sensor type and sensor characteristics.

3. **Add control system**
   A Matlab/Simulink interface enables users to design and verify algorithms for data processing, sensor fusion, decision making and control as well as the re-use of existing Simulink models such as vehicle dynamics models from CarSim, Dyna4 or ASM.

4. **Run experiment**
   A 3D visualisation viewer allows users to analyse the results of the experiment. It provides multiple viewpoints, intuitive navigation controls, and picture and movie generation capabilities. Also, interfaces with ControlDesk and LabView can be used to automatically run an experiment batch of scenarios as well as to run hardware-in-the-loop (HIL) simulations.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

3 Presentation of control Logic (CL)

3.1 Renault CL

The Renault CL is relying on Robotics Operating System (ROS)[4], a messages and services oriented framework, inside an Ubuntu Computer. The CL is an Ubuntu and ROS application, written entirely in C++ language. To interface this CL to PreScan there are two alternatives:

- Development of a C++ interface, which handle the needed messages and send/ or receive them through TCP/UDP functions to/from PreScan, which enable communication.
- The other method, which seems simpler and more flexible is to rely on Matlab/Simulink software, which is natural to PreScan, and has a ROS toolbox. The idea is that Matlab/Simulink are acting simply as a ROS node, and then there is no need to develop any kind of function interfaces.

The simulation would be playing the scenarios generated with the navigation system and decision making (NSDM) software (see Figure 4), where the scenarios rely on the perception output from the sensors provided by PreScan, and the same simple maps configured on PreScan and the Autonomous Driving Commuter Car (ADCC) software (see Figure 5 to 7).

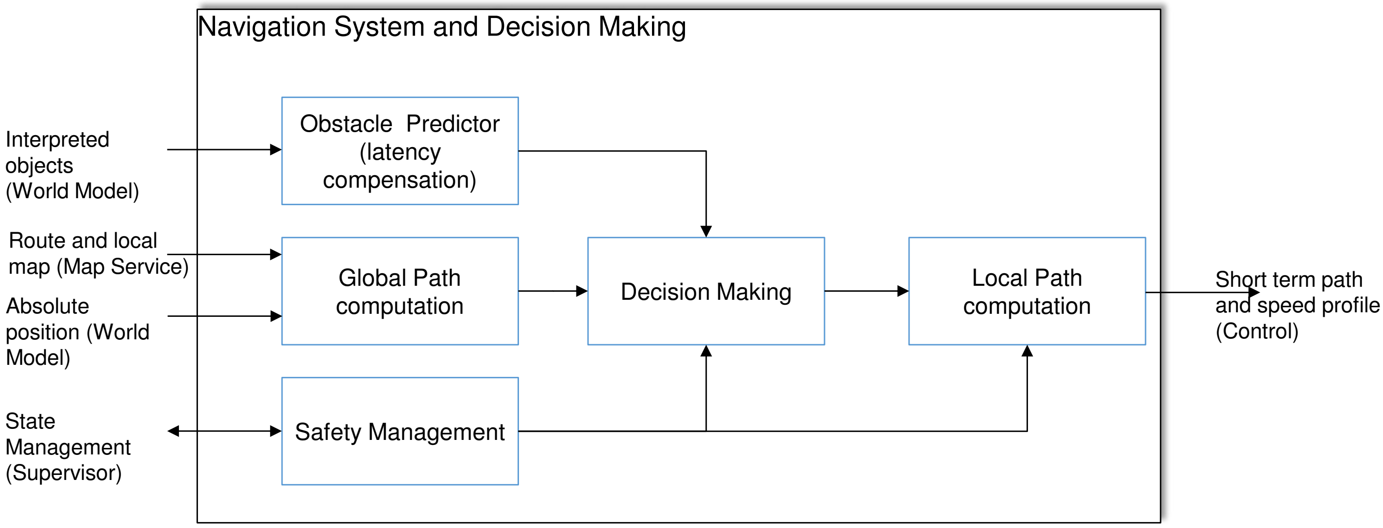


Figure 4: The functional architecture of the NSDM

[4] www.ros.org

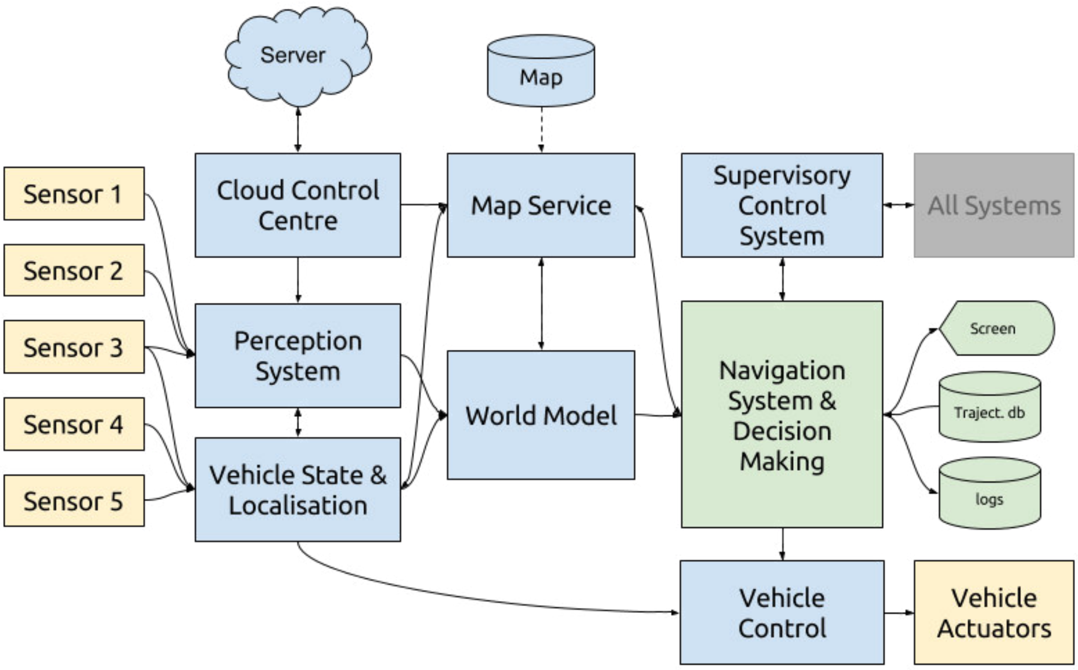


Figure 5: Architecture of ADCC A.V. vehicle software

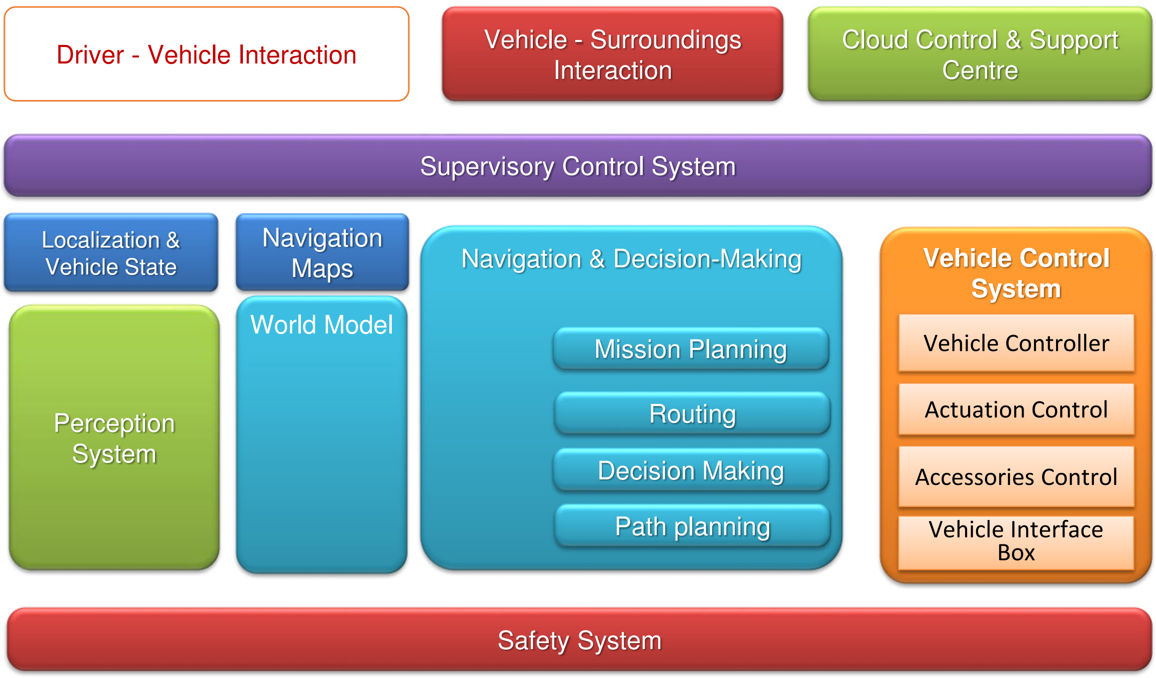


Figure 6: Architecture of ADCC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

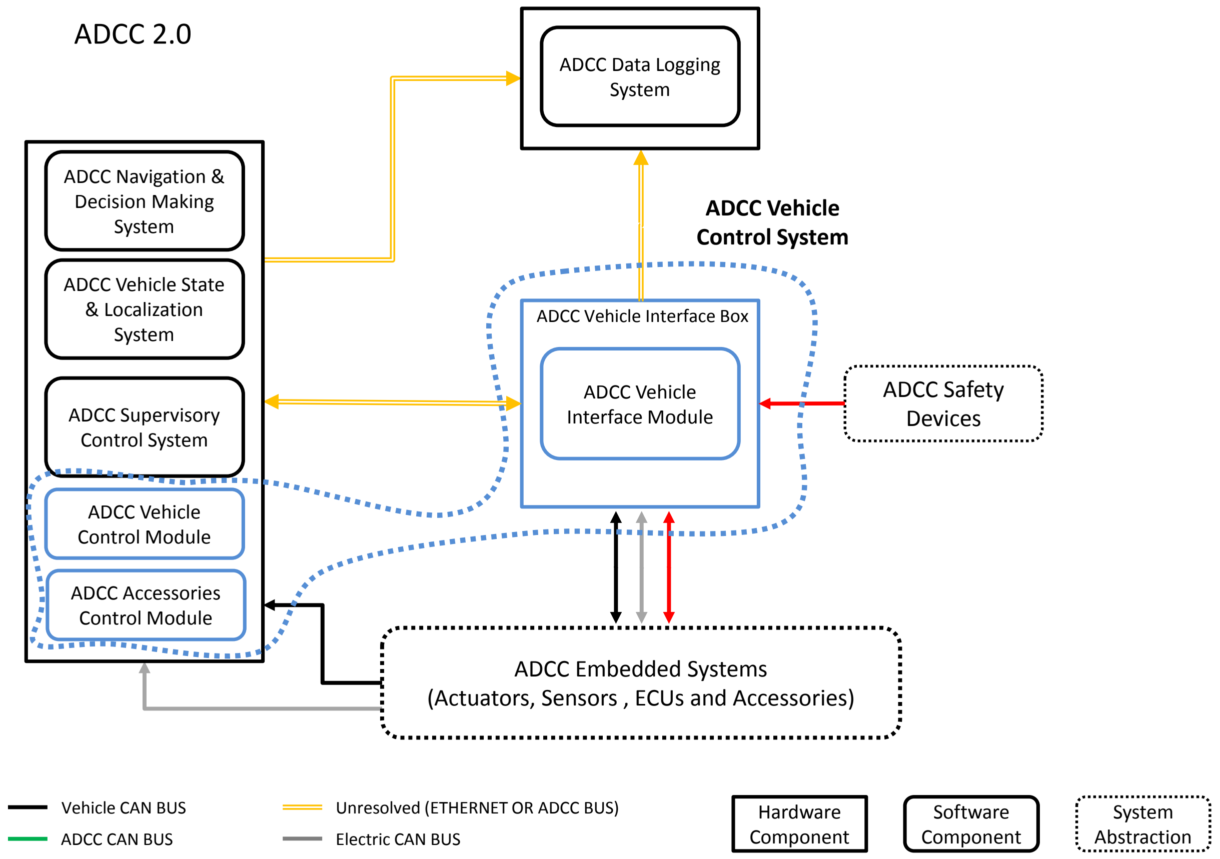


Figure 7: The Vehicle Control System

3.2 VEDECOM CL

The VEDECOM-CL software is defined in the MATLAB-Simulink platform which enables an easy integration with the PreScan software as there is an existing interface with Simulink.

The current defined model is a lighter version of the current VEDECOM-CL software. The objective of this initial version was to resolve all interface issues that could happen when there is an interface between different software.

This version provides a reduced set of functionalities, defined in the next section (4. Connection between VEDECOM CL and PreScan).

3.2.1 Overview

Figure 8 (see below) presents the integrated version of VEDECOM CL. It has already integrated several aspects of the complete CAV CL: lateral and longitudinal control, speed and steering regulation.

Currently lateral and longitudinal controls rely on a pre-defined trajectory, but already take into account several aspects of a more complex control logic (see Section 4.2: Results).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

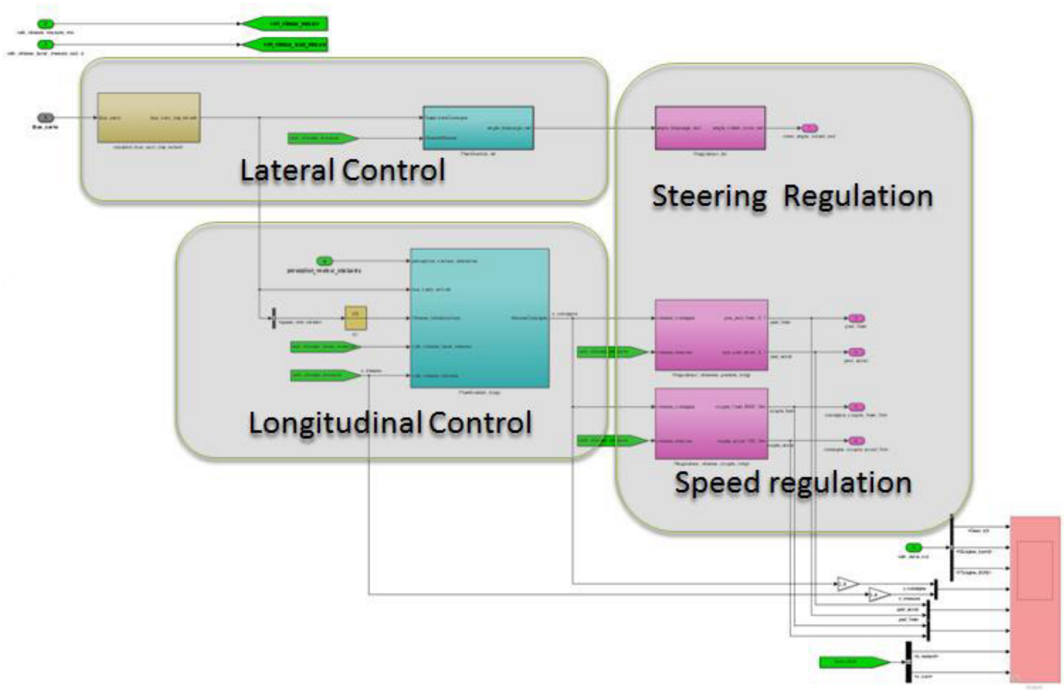


Figure 8 Composition of the VEDECOM CL for CoEXist

4 Connection between VEDECOM CL and PreScan

4.1 Description

To develop a vehicle behaviour simulation acting like CAVs, some CAV behaviour models have to be introduced into the simulation framework. One of these models operated with PreScan is a simplified VEDECOM control logic (VCL). It is currently developed with Simulink: a part of the MATLAB product that enables developers to build complex systems based on a diagram for interaction between the different subsystems that compose the defined software. It embeds different already existing components and allows users to build their own configurations, typically as in the case of CoEXist, using the different parts that compose the control logic of a CAV (see. https://fr.mathworks.com/products/simulink.html).

As PreScan provides an existing integration with Simulink, the integration of the VEDECOM CL into the simulation loop does not suffer technological issue due to the interface. Figure 9 below provides an overview of the VEDECOM CL integration into the simulation loop of PreScan.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

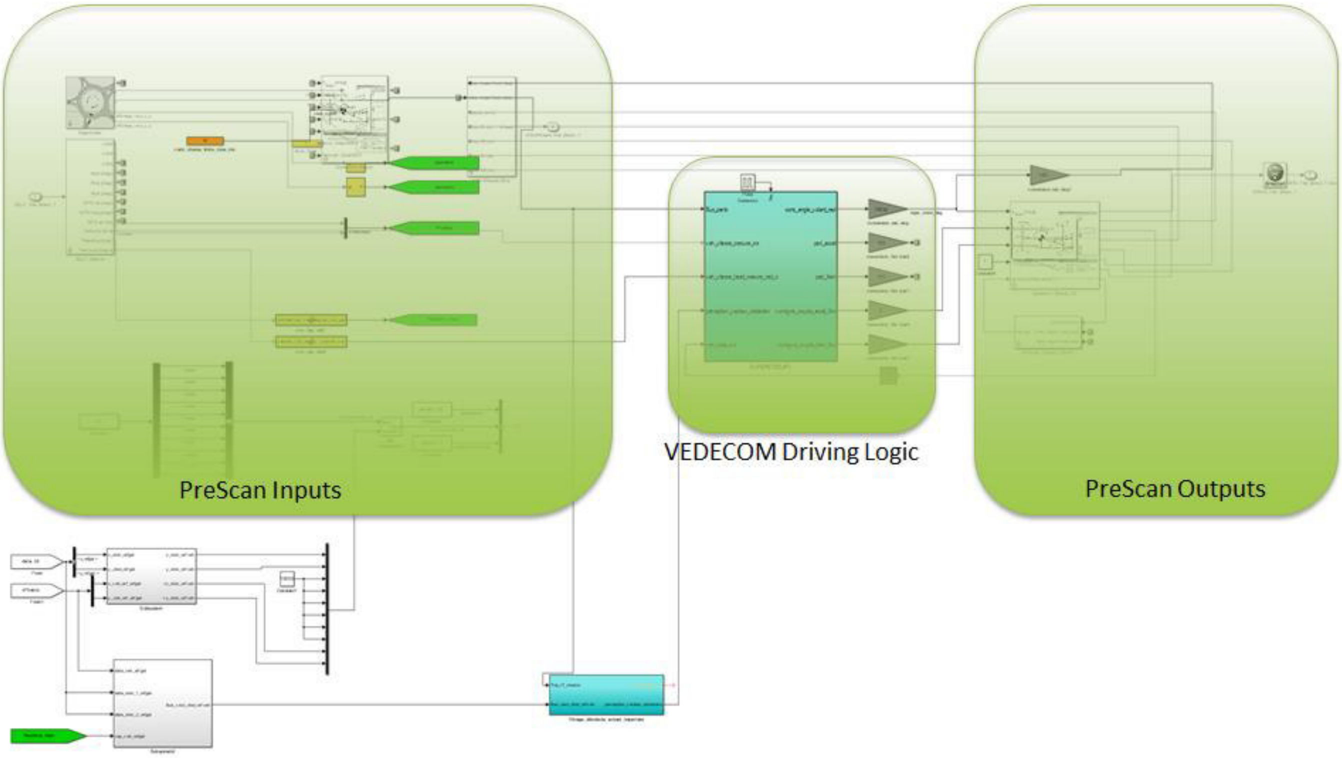


Figure 9 Overview of the VEDECOM CL integration into the simulation loop of PreScan

From left to right, Figure 9 illustrates first the PreScan input information needed by VCL, then the control logic (described in 3.2.1 Overview) that computes the requested trajectory based on the input information, i.e. the actuator orders, and thereafter the trajectory information is sent back to PreScan to control the vehicle model.

4.1.1 Adaptation of the vehicle model in PreScan

One major issue for the interface between PreScan and VEDECOM CL was the dynamic model of the controlled vehicle and particularly about the defined engine system. Indeed, all PreScan defined vehicle models are currently based on thermal motors which use a gear box, whereas the VEDECOM CL is defined for an electric vehicle.One of the main task about this issue was to adapt an existing thermal engine model to fulfil VEDECOM CL requirement. This adaptation has been made in partnership with PreScan engineer Bart Heijke (TASS).

4.2 Results

During the PreScan training session on 13.07.2017, which took place at the VEDECOM Institute in Versailles, France a very simple scenario has been developed in order to make initial tests for the VCL/PreScan integration.

From this scenario, we could test several functionalities, among them, we can cite:

- Trajectory following, where speed regulation is computed according to the following aspects:

    - **Road curvature:** depending on the curvature, the vehicle adapts its velocity for passenger safety and comfort. It is possible to make it more reactive depending on the payload (passenger or goods)
    - **Obstacles:** if an obstacle is on the trajectory of the CAV, the CL adapts the velocity to always respect a safe gap distance between the previous vehicle and the CAV, in order to be able to realize an emergency break if something goes wrong.
    - **Mandatory speed limit (SML):** the third aspect of speed regulation is about MSL, the CAV must respect them and the defined CL takes them into account to compute the current velocity.

- Steering regulation

    - This parameter relies on the road geometry only, it is associated to the lateral control.

A demonstration video is available here that illustrates these results.

5 Connection between Renault CL and PreScan

The Autonomous Driving Commuter Car (ADCC) A.V. software will be running on an Ubuntu platform with ROS, this may be a dedicated computer connected to a Windows PC running Prescan and Matlab/Simulink, or a single Windows platform running Prescan and Matlab/Simulink and the ADCC software will be running inside an Ubuntu virtual machine hosted by this PC.

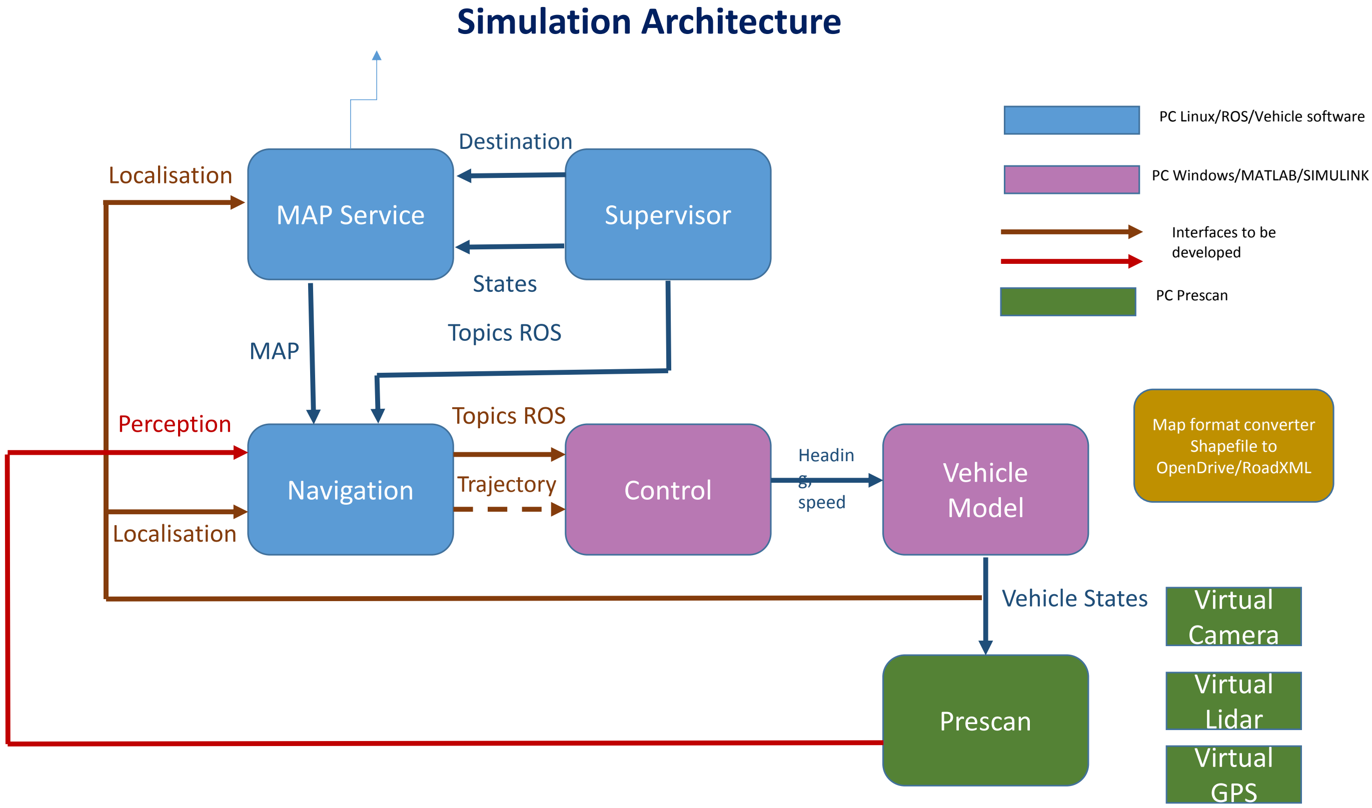


Figure 10: Connection between Renault ADCC and Prescan

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

As shown in Figure 10, the ADCC ROS messages have been integrated inside Matlab/Simulink, but there are blocking version problems (i.e. compatibility issues). Renault owns Matlab/Simulink version 2016b, which is the official version inside Renault, and Matlab/Simulink 2017b for testing purposes, but unfortunately these two versions are not supported by PreScan, they will be supported in version 8.3, which is planned for the first quarter of 2018. In the meantime, a temporary license for 2015b (supported by PreScan) will be used.

Initially a virtual machine with scenarios will be set up and very simple scenarios will be run. Currently real maps used by ADCC, due to the proprietary format, are not supported by PreScan (limitation with OpenStreetMap).This can potentially be overcome by developing an XML module translator to ease the import of real world ADCC maps to PreScan. The following steps need to be addressed now:

- Ensure the link between ROS/ADCC – Matlab/Simulink – PreScan is working.
- Set up simple scenarios with simple maps.
- Integrate more elaborate vehicles dynamics models.

6 Connection between PreScan and Vissim

6.1 PTV Vissim - Driving simulator interface

The PTV Vissim add-on module "Driving Simulator Interface" (see Figure 11) allows to connect Vissim to a driving (cycling, walking) simulator (DS). That DS can either be a simulator hardware used by a human or a piece of software representing the algorithms of a CAV (or multiple CAVs).

Vissim provides the surrounding traffic (vehicles, bicycles, pedestrians) to be visualized in the DS, and the DS passes back the current position and orientation of the simulator vehicle(s) (bicycle(s) / pedestrian(s)). The vehicles and pedestrians in the Vissim network react to this simulator data as to all other vehicles and pedestrians in the microscopic simulation model. In addition, Vissim passes traffic signal states to the DS for visualization, and the DS can set detectors in Vissim explicitly in order to affect the signalization.

The DS does not need to know the Vissim data model where the network is modelled from links, connectors, areas, ramps and obstacles. The DS needs to have its own world model (for simulation and visualization). As all vehicle and pedestrian positions are exchanged in Cartesian world coordinates (x/y/z), the DS must be able to provide/use such coordinates, and the coordinates of the networks on both sides (Vissim / DS) must match precisely.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

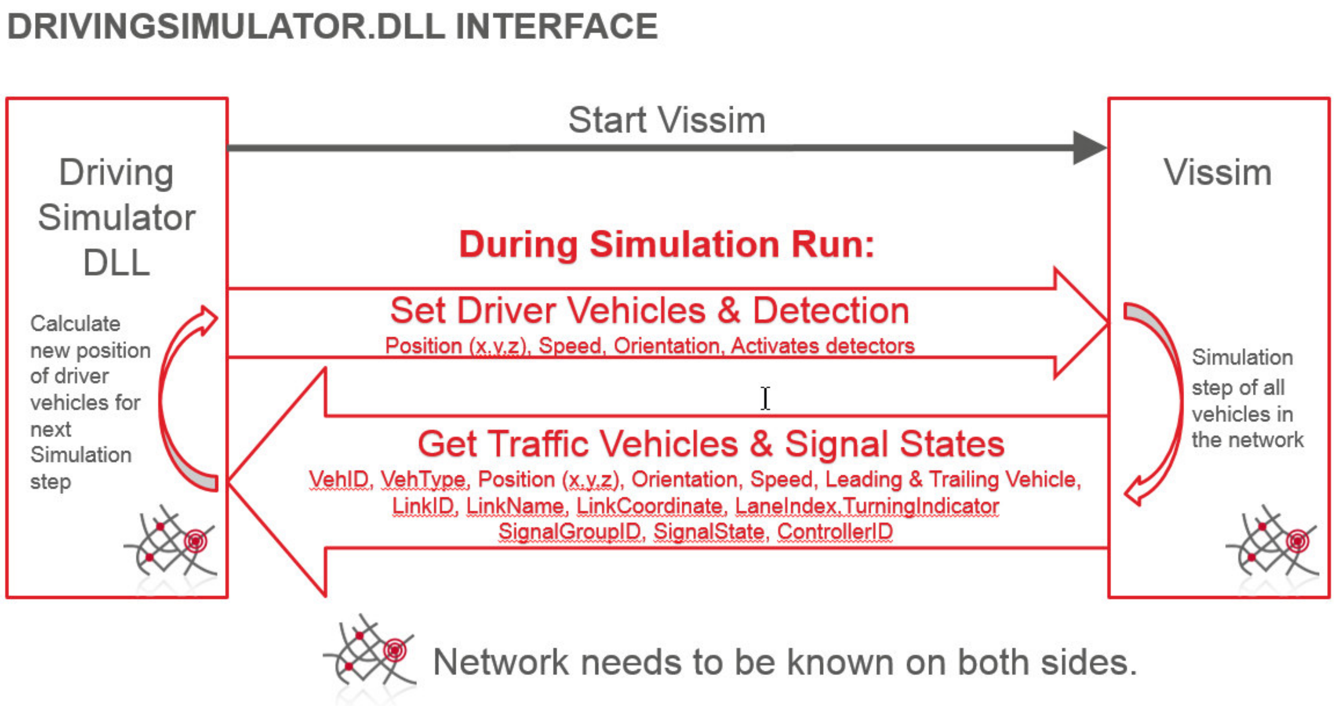


Figure 11 Driving simulator interface

6.1.1 Architecture

The DS software must connect to a Windows DLL provided by PTV. This DLL communicates with a Vissim instance on the same computer through shared memory. If the DS software doesn't run on Windows, it needs to connect to a Windows application to be built by the user through sockets or any other means of process communication.

The start of the Vissim instance is triggered through a function call from the DLL, with the desired Vissim version and the network file name to be loaded as parameters. Additional optional parameters allow to set the frequency of the simulator, a visibility radius, the maximum number of data sets for vehicles, pedestrians, signal groups and detectors to be passed in one or both directions. Vissim is started through the COM interface, so it needs to be registered as COM server. In the Vissim network settings, the option "driving simulator" needs to be activated, and a vehicle type must be selected to be used for the DS vehicles. After loading the network, Vissim establishes the communication to the DLL and waits for the starting location of the DS vehicle (in world coordinates, fitting with the Vissim network) to be passed from the DLL.

Vissim determines the most probable location for the DS vehicle in the Vissim network by selecting from all links at the passed world coordinates the link with the driving direction closest to the orientation of the DS vehicle. After the vehicle has been placed at that location in the Vissim model, the first time step of the Vissim simulation is executed. After that time step, Vissim passes the world coordinates of all vehicles in the network (except the DS vehicle) to the DLL and waits for the new location of the DS vehicle. The simulator can now read the new locations of the Vissim vehicles, visualize them and then pass back the new location of the DS vehicle. After this, Vissim immediately calculates the next time step, and so on.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

Before the DS passes the new location of the simulator vehicle to Vissim, it can optionally set detector states. Before or after retrieving the new locations of the Vissim vehicles, the DS can optionally retrieve the signal states passed from Vissim and/or a list of all vehicles which have entered the Vissim network in this time step, a list of all vehicles which have left the Vissim network in this time step and a list of all vehicles which have changed their location in this time step. This data exchange between Vissim and the DS must be executed at least once per Vissim simulation time step. If the simulator runs at a higher frequency (frame rate) than Vissim, the interface provides automatic interpolation of the positions of the Vissim vehicles and pedestrians between Vissim simulation time steps.

The timing of the co-simulation is controlled by the DS. It can run faster than real time (if this is possible for the DS) but not faster than a stand-alone Vissim simulation of the same network. (The Vissim simulation of a time step can take much less or much more than real time, depending on the used hardware and the size of the network. A big network might not be possible to be simulated with very short time steps in real time). The simulation speed in Vissim needs to be set to “maximum”, and the visualization in Vissim should be switched off in order to achieve the highest possible simulation speed. The DS can slow down the co-simulation as much as it requires (usually to exactly real time) without hurting the Vissim simulation at all.

The Vissim plugin of PreScan is a tool that enables co-simulation and data transfer between PreScan and Vissim. This makes it possible to have traffic on a road network in a PreScan simulation that is generated by Vissim as shown in Figure 12.

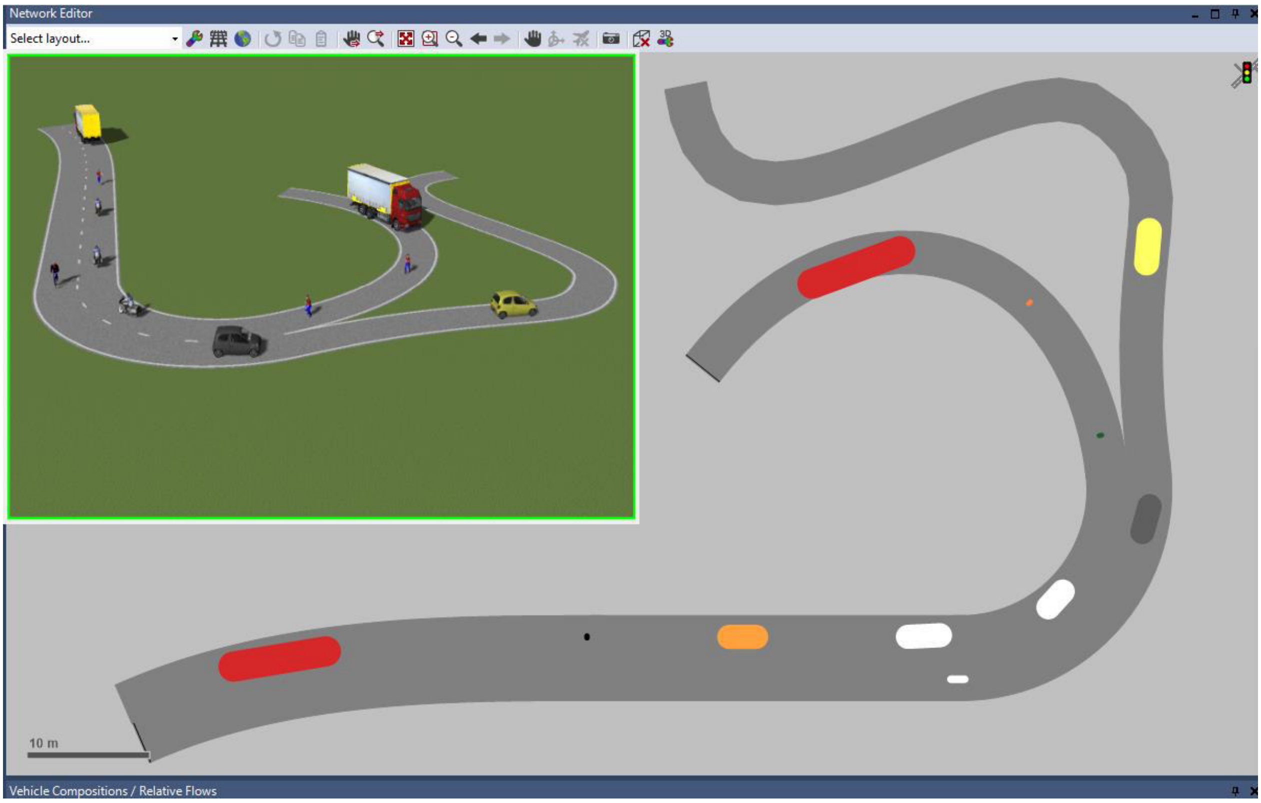


Figure 12: Vissim scenario playing with 3D rendering

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 723201

6.2 New development in PTV Vissim

PTV released the new version PTV Vissim 10 in October 2017. This version has several new functionalities & improvements, which were partially developed under CoEXist project and can be used by users with correspondent application programming interface (API). Both interfaces can be used for the simulation of automated vehicles by users who have the control algorithms for automated vehicles. Most interesting new features and improvements are:

For DrivingSimulator.DLL interface:

- Automatic interpolation of Vissim vehicle/pedestrian world coordinates, orientations and speeds between Vissim time steps if the simulator informs Vissim about a higher frame rate than the Vissim simulation resolution in the call of VISSIM_Connect().
- Optionally, simulator pedestrians can be passed to Vissim now. The pedestrian type for those needs to be selected in the network settings on the new tab page "Driving simulator". (The checkbox for activation of the driving simulator interface and the selection box for the vehicle type for simulator vehicles have been moved there as well.)
- The maximum number of objects of each type to be exchanged between the simulator and Vissim can be set by the simulator in the call of VISSIM_Connect().
- The simulator can pass a maximum visibility radius to Vissim in the call of VISSIM_Connect(). Vissim vehicles and pedestrians will be passed to the simulator only if they are inside of this radius from the centre of the front end of a simulator vehicle or pedestrian.

For DriverModel.DLL interface:

- External driver model DLLs may be used now in multithreaded simulation runs if all DLLs confirm that they support multithreading. (Unless there is only one externally controlled vehicle or all externally controlled vehicles are on the same link, the DLL needs to be programmed accordingly, of course.)
- If the DLL requests it, Vissim sends the data of all nearby vehicles that the ego vehicle sees according to the current driving behaviour (min./max. look ahead and look back distances, number of observed vehicles) instead of at most 2 each upstream and downstream per lane.
- User-defined vehicle attribute values can be passed to the DLL and can be modified by the DLL.
- World coordinates of the front end and rear end of nearby vehicles are passed from Vissim to the DriverModel.DLL as well. The polyline of the current lane of the ego vehicle (along its route/path, within the visibility distance) is passed to the DLL as well.

The full description of both mentioned PTV Vissim interfaces can be found in Appendix (see appendix 1: “PTV Vissim API - Driver Model Interface.pdf“ and appendix 2: “PTV Vissim API - Driving Simulator Interface.pdf”).

7 Next steps

7.1.1 VEDECOM CL to VISSIM

Next steps by VEDECOM:

- Run co-simulations in PreScan and Vissim in several traffic situations in simple (principal) models providing outputs for Vissim developers (requires deliverables from WP3).
- Work on an additional solution using Vissim´s “driver model interface” which would allow to run the VEDECOM´s control logic directly in Vissim, without PreScan (DLL will be provided CW48 and documentation CW49). This approach does not include the sensor simulation sensors (that part can be done only with co-simulation), but would be a great help for the further development of Vissim.

Next steps by PTV until May 2018:

- Analyse the outputs from the co-simulations and incorporate necessary results into the Vissim driving behaviour model.
- Provide support for VEDECOM for a successful transfer of the control algorithms to Vissim´s driver model interface.
- Use the DLL from VEDECOM with driver model interface for simulation test – observing and analysing the driving behaviour (especially differences to actual human driver behaviour model).

7.1.1.1 Driver model interface approach

The Eexternal driver model DLL interface of Vissim provides the option to replace the internal driving behaviour by a fully user-defined behaviour for some or all vehicles in a simulation run. The user-defined algorithm must be implemented in a DLL written in C/C++ which contains specific functions (as specified below). During a simulation run, Vissim calls the DLL code for each affected vehicle in each simulation time step to determine the behaviour of the vehicle. Vissim passes the current state of the vehicle and its surroundings to the DLL and the DLL computes the acceleration / deceleration of the vehicle and the lateral behaviour (mainly for lane changes) and passes the updated state of the vehicle back to Vissim.

The external driver model can be activated for each vehicle type separately in the dialog box “Vehicle Type” by checking the checkbox "Use external driver model” on the tab page “External Driver Model" and selecting a driver model DLL file and optionally a parameter file to be used. If this option is checked, the driving behaviour of all vehicles of this vehicle type will be calculated by the selected DLL. A subdirectory DriverModelData\ must exist in the directory of vissim.exe in order to avoid a warning message when Vissim is started.

7.1.1.2 Use of driver model interface within CoExist

PTV will use the DLL delivered by VEDECOM to run test simulations in PTV Vissim and to run several evaluations in order to find out the differences in driving behaviour between automated vehicles and conventional vehicles. Automated vehicles will be powered by VEDECOM control algorithms, conventional vehicles by existing PTV Vissim driving behaviour algorithms. Understanding the differences in driving

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 723201

behaviour of automated vehicles is a precondition for successful AV-ready development of the microscopic simulation software PTV Vissim.

7.2 Simulations & analysis

Next steps for VEDECOM:

- Provide PTV with a “Black-Box” DLL of VEDECOM-CL with documentation about its inputs and outputs before middle of December 2017.
- Integrate into the initial CL used to prototype the global architecture of CoEXist platform more functionalities to obtain a more realistic CAV model.

Next steps for Renault:

- As the initial major issue about interfacing Renault-CL and PreScan is going to be solved, Renault will discuss with PTV to define the correct way to interface their CL and VISSIM. Different options have been evocated, one seems to be the more feasible: providing a Virtual Machine with Renault-CL embedded with a network communication API to exchange information with Vissim.

Next steps for VEDECOM and Renault:

- Run co-simulations in PreScan and Vissim in several traffic situations of simple (principal) models providing outputs for Vissim developers.

Next steps for PTV Group until May 2018:

- Analyse the outputs from co-simulations and incorporate necessary results into Vissim driving behaviour model.
- Provide support for VEDECOM to successfully transfer the control algorithms to Vissim´s driver model interface.
- Use the DLL from VEDECOM with driver model interface for simulation test – observe and analyse the driving behaviour (especially differences to actual human driver behaviour model).

8 Conclusion

CoExist software solution for traffic simulation including AV models is currently making good progress.Initial issues have mainly been solved and a first version of the CAV model is going to be integrated in the final platform used by CoEXist cities.

As inter operation problems are resolved, partners (VEDECOM) will work on modelling improvements to provide a more advanced solution, more realistic and including higher level functionality to fulfil the expected behaviour of a level-5 autonomous vehicle.

Different solutions have been discussed between partners to integrate these improvements inside the final CoEXist platform. As the CoEXist platform will mainly rely on Vissim, VEDECOM-CL and Renault-CL models should be integrated inside Vissim as a new vehicle behaviour model. Two main tasks are required to achieve this goal: improvement and integration. Integration in Vissim suffers, from the VEDECOM point

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 723201

of view, from requiring a lot of adaptation of the VEDECOM-CL to be compliant with Vissim data format. As this work requires a good knowledge of Vissim, it has been agreed that VEDECOM will provide its model as a Black Box with documentation of its interfaces. With this material, Vissim team could realise the interface between the V-CL and its software and analyse the V-CL behaviour to define its own CAV model.

This way, during the next phases of the work package 2 of CoEXist, VEDECOM will focus on the CAV modelling improvement and will periodically provide new model functionalities for Vissim. As the interfaces will not evolve between two deliveries, integration inside Vissim will not add overloads for the PTV team.

9 Development partners for D2.2

10 Appendices

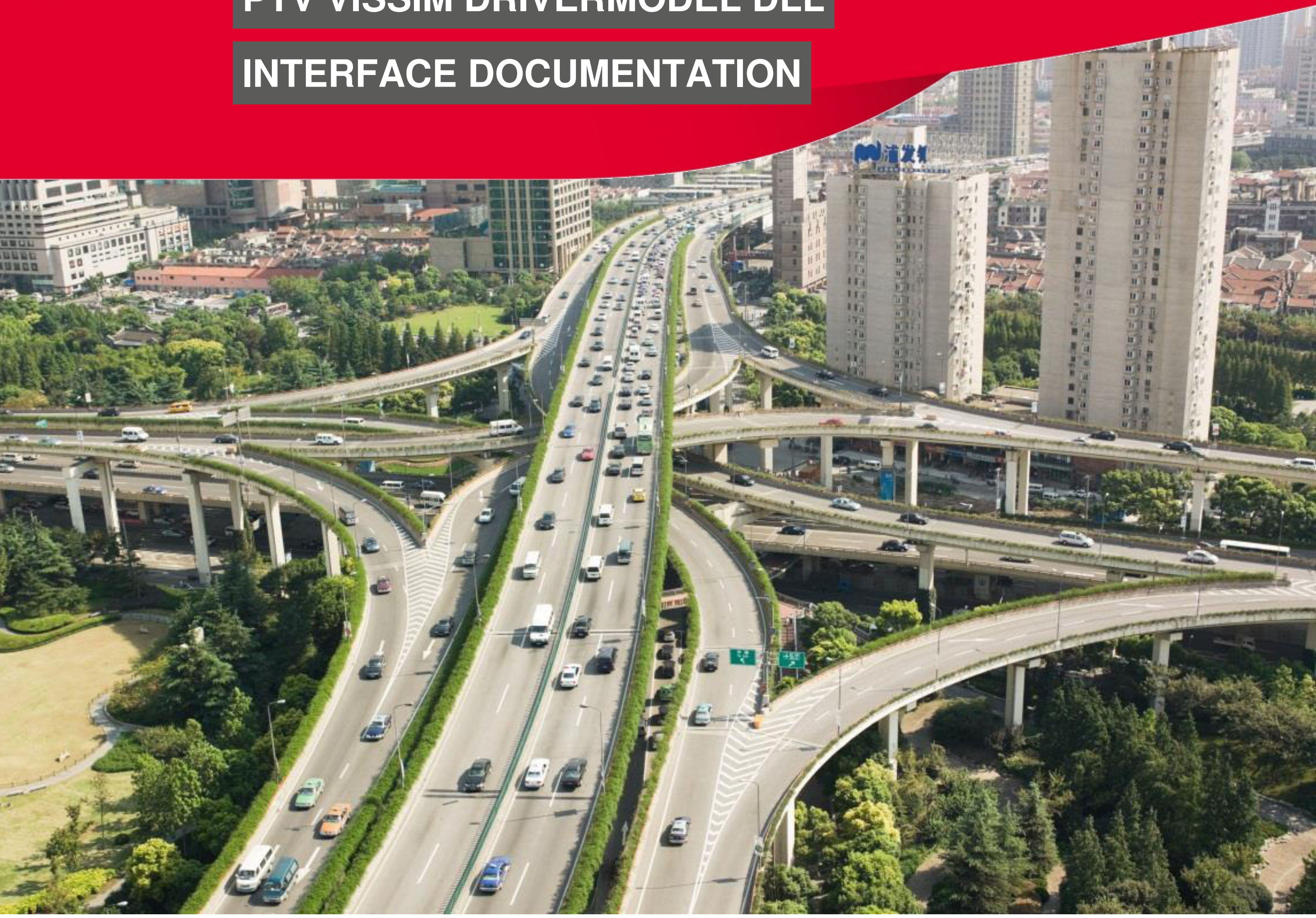
10.1 PTV Vissim API - Driver Model Interface

10.2 PTV Vissim API - Driving Simulator Interface

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 723201

PTV VISION API

PTV VISSIM DRIVERMODEL DLL
INTERFACE DOCUMENTATION



Imprint

This documentation is valid for PTV Vissim version 10.00-03 and later.
Modifications added after Vissim 10.00-02 are marked like this.
Edited by: Lukas Kautzsch

PTV Planung Transport Verkehr AG
Haid-und-Neu-Str. 15
D-76131 Karlsruhe
Germany

Phone: +49 721 9651-0
Fax: +49 721 9651-699
info@vision.ptvgroup.com
www.ptvgroup.com

Contents

Imprint 2

Contents 2

1 General 3

2 DLL Interface 3

3 Commands 5

3.1 Init ... 5
3.2 CreateDriver ... 6
3.3 MoveDriver ... 6
3.4 KillDriver ... 13

4 Lane Change 14

4.1 Simple lane change - handled by Vissim ... 14
4.2 Lane change - handled by the driver model DLL ... 15

5 Multithreading 15

1 General

The External Driver Model DLL Interface of Vissim provides the option to replace the internal driving behavior by a fully user-defined behavior for some or all vehicles in a simulation run. The user-defined algorithm must be implemented in a DLL written in C/C++ which contains specific functions (as specified below). During a simulation run, Vissim calls the DLL code for each affected vehicle in each simulation time step to determine the behavior of the vehicle. Vissim passes the current state of the vehicle and its surroundings to the DLL and the DLL computes the acceleration / deceleration of the vehicle and the lateral behavior (mainly for lane changes) and passes these values back to Vissim to be used in the current time step.

The external driver model can be activated for each vehicle type separately in the dialog box “Vehicle Type” by checking the checkbox "Use external driver model” on the tab page “External Driver Model" and selecting a driver model DLL file and optionally a parameter file to be used. If this option is checked, the driving behavior of all vehicles of this vehicle type will be calculated by the selected DLL.

A subdirectory `DriverModelData\` must exist in the directory of vissim.exe in order to avoid a warning message when a simulation run is started.

If the number of cores is not set to 1 in the Simulation Parameters of the Vissim network, the DLL needs to confirm that it supports multithreading. If it doesn’t, the simulation run is canceled with an error message.

2 DLL Interface

For the implementation of the DLL, several source code files are provided:

- DriverModel.h: Header file for a driver model DLL.
  This file should not be changed. It contains the definitions of all "type" and "number" constants used by Vissim in calls of the “DriverModel*” DLL functions which are declared here, too.
- DriverModel.cpp: Main source file of a driver model DLL.
  This file is the place where calculations or calls of functions of the driving behavior model algorithm should be added. Provided is a dummy version which does "nothing" (sets DRIVER_DATA_WANTS_SUGGESTION and DRIVER_DATA_USE_INTERNAL_MODEL).
  (In contrast, the file DriverModelExample.cpp contains a very simple following model which actually overrides the internal model of Vissim.)
  The preprocessor #define DRIVERMODEL_EXPORTS must be set in the compiler options for compilation of DriverModel.cpp! (This is included in the provided project file – see below.)
- DriverModel.vcproj: Visual C++ 2010 project file for a driver model DLL. This file can be used if the DLL is to be created with Microsoft Visual C++.

A driver model DLL must contain and export 3 functions which are called from Vissim:
`DriverModelSetValue`, `DriverModelGetValue` and
`DriverModelExecuteCommand`.
The signature of these functions and their meaning is as follows:

```
int   DriverModelSetValue (long   type,
                           long   index1,
                           long   index2,
                           long   long_value,
                           double double_value,
                           char*  string_value);
```

Vissim passes the current value of the data item indicated by `type` and (for most types) indexed by `index1` and sometimes `index2`. The value is passed in `long_value`, `double_value` or `string_value`, depending on `type`.
The code in the function must make sure to save the value somewhere if it is required later for the driving behavior calculation because the next call of this function from Vissim will overwrite the local parameter.
(In the provided dummy DLL the values suggested by Vissim (via several calls to `DriverModelSetValue`) are saved in global variables in order to be able to send them back when Vissim calls `DriverModelGetValue` after the one call of `DriverModelExecuteCommand (DRIVER_COMMAND_MOVE_DRIVER)` per vehicle per time step.)
The function must return 1 for all types which are not marked as optional in the documentation below. For optional types, it can return 0 to inform Vissim that it doesn't handle this type.

```
int   DriverModelGetValue (long   type,
                           long   index1,
                           long   index2,
                           long   *long_value,
                           double *double_value,
                           char*  *string_value);
```

Vissim retrieves the value of the data item indicated by `type` and (for most types) indexed by `index1` and sometimes `index2`. Before returning from this function, the value must be written to either `*long_value`, `*double_value` or `*string_value`, depending on the data type of the data item.
The function must return 1 for all types which are not marked as optional in the documentation below. For optional types, it can return 0 to inform Vissim that it doesn't handle this type.

```
int   DriverModelExecuteCommand (long number);
```

Vissim requests execution of the command indicated by `number`.
Currently available command constants are `DRIVER_COMMAND_INIT`, `DRIVER_COMMAND_CREATE_DRIVER`, `DRIVER_COMMAND_MOVE_DRIVER` and `DRIVER_COMMAND_KILL_DRIVER`. The function must return 1 for all these commands lest Vissim stop the simulation run.

Before Vissim requests execution of one of the available commands (`Init`, `CreateDriver`, `MoveDriver`, `KillDriver`) of the DLL there are always several calls of the DLL function `DriverModelSetValue`, one for each data item that might be used by the DLL when executing the command.

After the command `MoveDriver` has finished computation, the resulting state of the vehicle is fetched from the DLL in a similar manner again by several calls of `DriverModelGetValue`.

For a complete list of defined values for `type` and `number` see `DriverModel.h`.

3 Commands

There are 4 commands that a driver model DLL must implement:
`Init`, `CreateDriver`, `MoveDriver` and `KillDriver`.

3.1 Init

This command is executed through a call of
`DriverModelExecuteCommand (DRIVER_COMMAND_INIT)`
at the start of a Vissim simulation run to initialize the driver model DLL.
Several basic parameters are passed to the DLL before this through
`DriverModelSetValue ()` (shortened "`Set`" below),
and some values are retrieved from the DLL through
`DriverModelGetValue ()` ("`Get`").

The sequence of calls to the DLL is as follows:

For each vehicle type which uses that DLL:

Get DRIVER_DATA_STATUS (optional)
Get DRIVER_DATA_STATUS_DETAILS (only if STATUS is not 0; optional)
Set DRIVER_DATA_PATH
Set DRIVER_DATA_PARAMETERFILE (optional)
Set DRIVER_DATA_TIMESTEP
Set DRIVER_DATA_TIME
Set DRIVER_DATA_VEH_TYPE
Get DRIVER_DATA_WANTS_SUGGESTION
Get DRIVER_DATA_SIMPLE_LANECHANGE
Get DRIVER_DATA_WANTS_ALL_NVEHS (optional)

Then only once:

Get DRIVER_DATA_ALLOWS_MULTITHREADING (optional)
DriverModelGetValue must set *long_value to 1 and return 1 if multiple cores are set to be used in the simulation parameters

For each user-defined attribute for vehicles in Vissim:
Set DRIVER_DATA_USE_UDA (optional)
index1 = key of the UDA; string_value = short name of the UDA
DriverModelSetValue must return 1 if values for this UDA are to be sent later for each vehicle and nearby vehicle, else 0

Execute DRIVER_COMMAND_INIT
Get DRIVER_DATA_STATUS (optional)
Get DRIVER_DATA_STATUS_DETAILS (only if STATUS is not 0; optional)

3.2 CreateDriver

`DriverModelExecuteCommand (DRIVER_COMMAND_CREATE_DRIVER)` is called from Vissim during the simulation run whenever a new vehicle is set into the network in Vissim (at the start of a time step, from a vehicle input, a PT line or a parking lot). In the same time step, a command `MoveDriver` for the same vehicle will follow later.

The sequence of calls to the DLL is as follows:

Set DRIVER_DATA_TIMESTEP
Set DRIVER_DATA_TIME
Set DRIVER_DATA_VEH_TYPE = VehicleTypeNumber
Set DRIVER_DATA_VEH_ID = NumberOfNewVehicle
Set DRIVER_DATA_VEH_DESIRED_VELOCITY = InitialDesiredVelocity
Set DRIVER_DATA_VEH_X_COORDINATE
Set DRIVER_DATA_VEH_Y_COORDINATE
Set DRIVER_DATA_VEH_Z_COORDINATE (optional)
Set DRIVER_DATA_VEH_REAR_X_COORDINATE (optional)
Set DRIVER_DATA_VEH_REAR_Y_COORDINATE (optional)
Set DRIVER_DATA_VEH_REAR_Z_COORDINATE (optional)

Execute DRIVER_COMMAND_CREATE_DRIVER

3.3 MoveDriver

`DriverModelExecuteCommand (DRIVER_COMMAND_MOVE_DRIVER)` is called from Vissim during the simulation run once per stime step for each vehicle of a vehicle type which uses this DriverModel DLL. *Before* this call, there are many calls of `DriverModelSetValue ()` to pass the current state of the vehicle and its surroundings to the DLL. *After* the execution of the command, there are several calls of `DriverModelGetValue ()` to retrieve the new values for acceleration and lateral behavior and optionally user-defined attributes from the DLL. Before any vehicle specific data is exchanged, Vissim passes the states of all signal groups and priority rules to the DLL.

The sequence of calls to the DLL is as follows:

3.3.1 Global data

Set DRIVER_DATA_TIMESTEP
Set DRIVER_DATA_TIME

For each SC (passed in index1), for each signal head (number passed in index2):
Set DRIVER_DATA_SIGNAL_STATE =
red = 1, amber = 2, green = 3, red/amber = 4, amber flashing = 5, off = 6,
other = 0

For each priority rule section (“yield sign”) (index 1 = 0; number passed in index2):
Set DRIVER_DATA_SIGNAL_STATE =
blocked = 1, free = 3

3.3.2 Vehicle specific data

For each vehicle of a vehicle type using this driver model DLL first its own data is passed from Vissim to the DLL, then data of all nearby vehicles along the route of the vehicle and then some data about the upcoming link / lanes geometry and other network objects. Then, the command to move the vehicle is called, and finally, the data calculated by the DLL is retrieved.

Data of the subject vehicle (at the start of the current time step)

Set DRIVER_DATA_TIMESTEP =
    time step length [simulation seconds] [0.1 .. 1.0]
Set DRIVER_DATA_TIME =
    current simulation time (simulation seconds since simulation start)
Set DRIVER_DATA_VEH_ID =
    ID of the vehicle to be moved
Set DRIVER_DATA_VEH_LANE =
    current lane number (rightmost = 1)
Set DRIVER_DATA_VEH_ODOMETER =
    total elapsed distance in the network [m]
Set DRIVER_DATA_VEH_LANE_ANGLE =
    angle relative to the middle of the lane [rad]
Set DRIVER_DATA_VEH_LATERAL_POSITION =
    distance of the front end from the middle of the lane [m]
Set DRIVER_DATA_VEH_VELOCITY =
    current speed [m/s]
Set DRIVER_DATA_VEH_ACCELERATION =
    current acceleration [m/s²]
Set DRIVER_DATA_VEH_LENGTH =
    vehicle length [m]
Set DRIVER_DATA_VEH_WIDTH =
    vehicle width [m]
Set DRIVER_DATA_VEH_WEIGHT =
    vehicle weight [kg]
Set DRIVER_DATA_VEH_MAX_ACCELERATION =
    maximum possible acceleration [m/s²]
Set DRIVER_DATA_VEH_TURNING_INDICATOR =
    left = 1, right = -1, none = 0, both = 2
    (non-zero only if set in the last time step from this DLL)
Set DRIVER_DATA_VEH_CATEGORY =
    car = 1, truck = 2, bus = 3, tram = 4, pedestrian = 5, bike = 6
Set DRIVER_DATA_VEH_COLOR =
    vehicle color (24 bit RGB value)
Set DRIVER_DATA_VEH_PREFERRED_REL_LANE =
    desired direction of a lane change because of a downstream connector
    of the vehicle's route or path or because of the right-side/left-side rule,
    positive = left, 0 = current lane, negative = right
Set DRIVER_DATA_VEH_USE_PREFERRED_LANE =
    0 = only preferable (e.g. European highway with right-side rule),
    1 = necessary (e.g. before a connector)

Set DRIVER_DATA_VEH_DESIRED_VELOCITY =
desired speed [m/s]
Set DRIVER_DATA_VEH_X_COORDINATE =
world coordinate X (vehicle front end)
Set DRIVER_DATA_VEH_Y_COORDINATE =
world coordinate Y (vehicle front end)
Set DRIVER_DATA_VEH_Z_COORDINATE =
world coordinate Z (vehicle front end), optional
(calculated correctly only if 3D visualization or a vehicle record
containing "Coordinate front" or "Coordinate rear" is active)
Set DRIVER_DATA_VEH_REAR_X_COORDINATE =
world coordinate X (vehicle rear end), optional
Set DRIVER_DATA_VEH_REAR_Y_COORDINATE =
world coordinate Y (vehicle rear end), optional
Set DRIVER_DATA_VEH_REAR_Z_COORDINATE =
world coordinate Z (vehicle rear end), optional
(calculated correctly only if 3D visualization or a vehicle record
containing "Coordinate front" or "Coordinate rear" is active)
Set DRIVER_DATA_VEH_TYPE =
vehicle type number (user defined)
Set DRIVER_DATA_VEH_CURRENT_LINK =
current link number, optional

Only if `DriverModelSetValue (DRIVER_DATA_VEH_CURRENT_LINK)` returned 1:
For each link in the vehicle's route/path:
Set DRIVER_DATA_NEXT_LINKS =
link number, optional

Set DRIVER_DATA_VEH_ACTIVE_LANE_CHANGE =
direction of an active lane change movement
(+1 = to the left, 0 = none, -1 = to the right), optional
Set DRIVER_DATA_VEH_REL_TARGET_LANE =
target lange (+1 = next one left, 0 = current lane, -1 = next one right), optional
Set DRIVER_DATA_VEH_INTAC_STATE =
interaction state as determined by the internal car following model
(sent only if `DRIVER_DATA_WANTS_SUGGESTION` has been set!):
FREE = 1, CLOSEUP = 2, FOLLOW = 3, BRAKEAX = 4,
BRAKEBX = 5,BRAKEZX = 6, BRAKESPW = 7, BRAKEKOOP = 8,
PELOPS = 9,PASS = 10, SLEEP = 11, DWELL = 12; optional
Set DRIVER_DATA_VEH_INTAC_TARGET_TYPE =
type of the relevant interaction target as determined by the internal car following model (sent only if `DRIVER_DATA_WANTS_SUGGESTION` has been set!):
no target = 0, real vehicle = 1, signal head = 2, priority rule = 3, conflict area = 4,
reduced speed area = 5, stop sign = 6, parking lot = 7, PT stop = 8; optional
Set DRIVER_DATA_VEH_INTAC_TARGET_ID =
number of the relevant interaction target as determined by the internal car following model (sent only if `DRIVER_DATA_WANTS_SUGGESTION` has been set!); optional

Set DRIVER_DATA_VEH_INTAC_HEADWAY =
distance to the relevant interaction target as determined by the internal car following model [m], front bumper to front bumper, including length of leading vehicle (sent only if `DRIVER_DATA_WANTS_SUGGESTION` has been set!); optional

For each user-defined attribute for vehicles in Vissim which has been selected in the Init step (key of the UDA passed in index1):
Set DRIVER_DATA_DRIVER_DATA_VEH_UDA =
value of that user-defined attribute (bool passed as long (0 or 1),
all floating point types passed as double [in their currently selected unit]), optional

Data of the nearby vehicles

For each nearby vehicle (up to two each downstream and upstream, on up to 2 lanes each on both sides and on the current lane) several values are passed from Vissim:

index1 and index2 are used as follows for `DRIVER_DATA_NVEH_*`:
index1 = relative lane: +2 = second lane to the left, +1 = next lane to the left,
0 = current lane,
-1 = next lane to the right, -2 = second lane to the right
(exception for DRIVER_DATA_NVEH_UDA: index 1 = ID of the UDA!)
index2 = relative position: positive = downstream (+1 next, +2 second next, possibly more if DRIVER_DATA_WANTS_ALL_NVEHS is set)
negative = upstream (-1 next, -2 second next, possibly more if DRIVER_DATA_WANTS_ALL_NVEHS is set)

First, for each index combination with index2 in {-2, -1, +1, +2} (the DLL needs to initialize further index2 values itself) an initialization:

Set DRIVER_DATA_NVEH_ID = -1

Then, for each *existing* nearby vehicle the real data:

Set DRIVER_DATA_NVEH_ID =
vehicle number
Set DRIVER_DATA_NVEH_LANE_ANGLE =
angle relative to the middle of the lane [rad] (positive = turning left)
Set DRIVER_DATA_NVEH_LATERAL_POSITION =
distance of the front end from the middle of the lane [m]
(positive = left of the middle, negative = right)
Set DRIVER_DATA_NVEH_DISTANCE =
gross distance [m] (front end to front end, negative = nveh is upstream)
Set DRIVER_DATA_NVEH_REL_VELOCITY =
speed difference [m/s] (veh. speed - nveh. speed)
Set DRIVER_DATA_NVEH_ACCELERATION =
current acceleration [m/s²]
Set DRIVER_DATA_NVEH_LENGTH =
vehicle length [m]
Set DRIVER_DATA_NVEH_WIDTH =
vehicle width [m]
Set DRIVER_DATA_NVEH_WEIGHT =
vehicle weight [kg]

Set DRIVER_DATA_NVEH_TURNING_INDICATOR =
left = 1, right = -1, none = 0, both = 2
(non-zero only if set in the last time step from this DLL)
Set DRIVER_DATA_NVEH_CATEGORY =
car = 1, truck = 2, bus = 3, tram = 4, pedestrian = 5, bike = 6
Set DRIVER_DATA_NVEH_LANE_CHANGE =
direction of a current lane change (+1 = to the left, 0 = none, -1 = to the right)
Set DRIVER_DATA_NVEH_TYPE =
number of the vehicle type in Vissim, optional
Set DRIVER_DATA_NVEH_X_COORDINATE =
world coordinate X (vehicle front end), optional
Set DRIVER_DATA_NVEH_Y_COORDINATE =
world coordinate Y (vehicle front end) , optional
Set DRIVER_DATA_NVEH_Z_COORDINATE =
world coordinate Z (vehicle front end), optional
(calculated correctly only if 3D visualization or a vehicle record
containing "Coordinate front" or "Coordinate rear" is active)
Set DRIVER_DATA_NVEH_REAR_X_COORDINATE =
world coordinate X (vehicle rear end), optional
Set DRIVER_DATA_NVEH_REAR_Y_COORDINATE =
world coordinate Y (vehicle rear end), optional
Set DRIVER_DATA_NVEH_REAR_Z_COORDINATE =
world coordinate Z (vehicle rear end), optional
(calculated correctly only if 3D visualization or a vehicle record
containing "Coordinate front" or "Coordinate rear" is active)
For each user-defined attribute for vehicles in Vissim which has been selected in the Init step (key of the UDA passed in index1!):
Set DRIVER_DATA_DRIVER_DATA_NVEH_UDA =
value of that user-defined attribute (bool passed as long (0 or 1),
all floating point types passed as double [in their currently selected unit]), optional

Data of the current link

Set DRIVER_DATA_NO_OF_LANES =
number of lanes of the link the vehicle is currently on

Data of all lanes of the current link of the subject vehicle

For each lane of the current link of the vehicle, several values are passed from Vissim:

For DRIVER_DATA_LANE_* index1 and index2 are used as follows:
index1 = lane number (rightmost = 1), index2 is irrelevant.

Set DRIVER_DATA_LANE_WIDTH =
lane width [m]
Set DRIVER_DATA_LANE_END_DISTANCE =
distance to end of lane [m] (can be emergency stop position before
connector, negative = no end of lane in visibility range)

Data of the current lane

Set DRIVER_DATA_CURRENT_LANE_POLY_N =
number of downstream lane polygon points within visibility distance
along the route/path of the vehicle, optional

For each polygon point (index1 = 0..n-1):
Set DRIVER_DATA_CURRENT_LANE_POLY_X =
X world coordinate of polygon point in the middle of the lane, optional
Set DRIVER_DATA_CURRENT_LANE_POLY_Y =
Y world coordinate of polygon point in the middle of the lane, optional
Set DRIVER_DATA_CURRENT_LANE_POLY_Z =
Z world coordinate of polygon point in the middle of the lane, optional

Data of the current and upcoming environment

Set DRIVER_DATA_RADIUS =
current curve radius [m]
Set DRIVER_DATA_MIN_RADIUS =
minimum curve radius [m] in visibility range
Set DRIVER_DATA_DIST_TO_MIN_RADIUS =
distance [m] to spot of minimum curve radius
Set DRIVER_DATA_SLOPE =
current slope (negative = drop)
Set DRIVER_DATA_SLOPE_AHEAD =
slope at end of visibility range

Data of the next signal head or priority rule stop line

index1 = signal controller number / priority rule: zero,
index2 = signal head number / priority rule: number (before 8.00-10: zero)

Set DRIVER_DATA_SIGNAL_DISTANCE =
distance [m] to next signal head or priority rule stop line (negative = none visible)

Set DRIVER_DATA_SIGNAL_STATE =
red = 1, amber = 2, green = 3, red/amber = 4, amber flashing = 5, off = 6,
other = 0 / priority rule (since 8.00-10): blocked = 1, free = 3
Set DRIVER_DATA_SIGNAL_STATE_START =
simulation time [s] when signal changed to current state / priority rule: zero

Data of the next reduced speed area or previous desired speed decision

Set DRIVER_DATA_SPEED_LIMIT_DISTANCE =
distance [m] to "speed limit sign" (reduced speed area: real distance,
desired speed decision: 1.0 m when just passed, negative: no sign visible)
Set DRIVER_DATA_SPEED_LIMIT_VALUE =
speed limit [km/h] (0 = end of reduced speed area)

Behavior data suggested for the current time step by Vissim's internal model

These values are passed only if `*long_value` has been set to 1 in the call of `DriverModelGetValue (DRIVER_DATA_WANTS_SUGGESTION).`

Set DRIVER_DATA_DESIRED_ACCELERATION =
    desired acceleration [m/s²] in this time step
Set DRIVER_DATA_DESIRED_LANE_ANGLE =
    desired angle relative to the middle of the lane [rad] (positive = turning left)
Set DRIVER_DATA_ACTIVE_LANE_CHANGE =
    direction of an active lane change movement (+1 = to the left, 0 = none,
    -1 = to the right, must be != 0 while lane change is not completed,
    will be used for NVEH_LANE_CHANGE seen from other vehicles)
Set DRIVER_DATA_REL_TARGET_LANE =
    target lane (+1 = next one left, 0 = current lane, -1 = next one right)

Execute Move Command

Execute DRIVER_COMMAND_MOVE_DRIVER

Pass new data calculated by the behavior model in the DLL back to Vissim

Get DRIVER_DATA_VEH_TURNING_INDICATOR =
    left = 1, right = -1, none = 0, both = 2
    (can be set by the DLL to make the ego vehicle in Vissim show a blinker (in the visualization) and to allow other vehicles to see that active blinker when they check `DRIVER_DATA_NVEH_TURNING_INDICATOR`)
Get DRIVER_DATA_VEH_DESIRED_VELOCITY =
    desired speed [m/s]
Get DRIVER_DATA_VEH_COLOR =
    vehicle color (24 bit RGB value)
Get DRIVER_DATA_USE_INTERNAL_MODEL =
    use the values passed from the driver model DLL = 0,
    use Vissim's internal model for this time step = 1
    (1 is only possible if `DRIVER_DATA_WANTS_SUGGESTION` has been set!), optional

Only if `*long_value` has been set to 0 in the call of `DriverModelGetValue (DRIVER_DATA_USE_INTERNAL_MODEL)`:
Get DRIVER_DATA_DESIRED_ACCELERATION =
    new acceleration [m/s²], optional
    [This value is limited by Vissim to the minimum of desired acceleration and maximum acceleration and to the maximum deceleration for the vehicle at the current speed. It is *not* affected by the limitation of the change of acceleration which is used in the internal car following model. If this value is set, Vissim shows the interaction state of the vehicle as "PELOPS" in the vehicle record.]

Get DRIVER_DATA_DESIRED_LANE_ANGLE =

desired angle relative to the middle of the lane [rad] (positive = turning left), optional

[If `*long_value` was set to 1 in the call of `DriverModelGetValue (DRIVER_DATA_SIMPLE_LANECHANGE)` this angle does not need to be calculated by the DLL and the return value of `DriverModelGetValue(DRIVER_DATA_DESIRED_LANE_ANGLE)` should be zero.]

Get DRIVER_DATA_ACTIVE_LANE_CHANGE =

direction of an active lane change movement (+1 = to the left, 0 = none, -1 = to the right, must be != 0 while lane change is not completed)

[If `*long_value` was set to 1 in the call of `DriverModelGetValue (DRIVER_DATA_SIMPLE_LANECHANGE)` setting this and `DRIVER_DATA_REL_TARGET_LANE` to +1 or -1 is sufficient to start a lane change which will be completed automatically by Vissim.]

Get DRIVER_DATA_REL_TARGET_LANE =

target lane (+1 = next one left, 0 = current lane, -1 = next one right)

[This is used by Vissim only if `*long_value` was set to 0 in the call of `DriverModelGetValue (DRIVER_DATA_SIMPLE_LANECHANGE)`.]

Regardless of `DRIVER_DATA_USE_INTERNAL_MODEL`:

For each user-defined attribute for vehicles in Vissim which has been selected in the Init step (key of the UDA passed in index1!):

Get DRIVER_DATA_DRIVER_DATA_VEH_UDA =

value of that user-defined attribute (bool passed as long (0 or 1), all floating point types passed as double [in their currently selected unit]), optional

3.4 KillDriver

`DriverModelExecuteCommand (DRIVER_COMMAND_KILL_DRIVER)` is called from Vissim when a vehicle reaches its destination and thus leaves the network (so memory which the DLL might have allocated for that vehicle can be freed):

Set DRIVER_DATA_VEH_ID =

ID of the vehicle to be killed

Execute DRIVER_COMMAND_KILL_DRIVER

4 Lane Change

The driver model DLL interface provides 2 different ways to control lane changes of vehicles affected by the external dll:

► Simple lane change

► Full control over the lane change

If simple lane change is selected, the driver model DLL only needs to initiate a lane change for a vehicle. Vissim assumes control of the lateral behavior of this vehicle while the lane change proceeds and informs the driver model DLL when it is done.

Without simple lane change the driver model DLL has full control of the vehicle and manages the lane change on its own – the driver model DLL must inform Vissim about the current state of the vehicle.

How lane changes are actually handled is determined by the data types `DRIVER_DATA_SIMPLE_LANECHANGE` and `DRIVER_DATA_WANTS_SUGGESTION`.

The lane change itself involves the data types `DRIVER_DATA_VEH_ACTIVE_LANE_CHANGE`, `DRIVER_DATA_VEH_REL_TARGET_LANE`, `DRIVER_DATA_ACTIVE_LANE_CHANGE`, `DRIVER_DATA_REL_TARGET_LANE` and `DRIVER_DATA_DESIRED_LANE_ANGLE`.

4.1 Simple lane change - handled by Vissim

This mode is chosen by setting `*long_value` to 1 as result of Vissim's initial request for `DRIVER_DATA_SIMPLE_LANECHANGE`.

If the driver model sets `*long_value` to 1 as result for `DRIVER_DATA_WANTS_SUGGESTION` as well, Vissim will send a suggestion whenever it detects that a lane change is necessary. As long as the driver model sets `*long_value` to 1 as result of Vissim's requests for `DRIVER_DATA_USE_INTERNAL_MODEL`, Vissim has complete control of lane changes.

A lane change can be initiated from the driver model DLL the following way:

1. The driver model sets `DRIVER_DATA_ACTIVE_LANE_CHANGE` to 1 (to the left) or -1 (to the right).
2. Vissim starts the lane changing mode for the current vehicle. While in this mode, the vehicle ignores the values sent from the DLL for `DRIVER_DATA_ACTIVE_LANE_CHANGE`, `DRIVER_DATA_REL_TARGET_LANE` and `DRIVER_DATA_DESIRED_LANE_ANGLE`. (The DLL must still return 1 from the `DriverModelGetValue` calls for these types!)
3. Vissim sets `DRIVER_DATA_VEH_REL_TARGET_LANE`, `DRIVER_DATA_VEH_LANE` and `DRIVER_DATA_VEH_LANE_ANGLE` while in lane changing mode. (`DRIVER_DATA_VEH_REL_TARGET_LANE` is set to zero when the middle of the front end of the vehicle has reached the target lane, and `DRIVER_DATA_VEH_LANE` is set to the new lane at this time, too.)
4. When Vissim has finished the lane change (i.e. when the whole width of the vehicle is on the new lane), it sends 0 as value of `DRIVER_DATA_VEH_ACTIVE_LANE_CHANGE`. (This does not necessarily mean that the vehicle has reached its desired lateral position within the destination lane.)

Hints:

- A current lane change cannot be interrupted when in simple lane change mode.
- Do not send back Vissim's suggestions for `DRIVER_DATA_DESIRED_LANE_ANGLE` while no lane change is currently active, instead, **return zero** from `DriverModelGetValue(DRIVER_DATA_DESIRED_LANE_ANGLE)`.

4.2 Lane change - handled by the driver model DLL

This mode is chosen by setting `*long_value` to 0 as result of Vissim's initial request for `DRIVER_DATA_SIMPLE_LANECHANGE`.

A lane change can be executed by the driver model DLL the following way:

1. The driver model DLL sets `DRIVER_DATA_ACTIVE_LANE_CHANGE`, `DRIVER_DATA_REL_TARGET_LANE` and `DRIVER_DATA_DESIRED_LANE_ANGLE` to the desired values.
2. Vissim moves the vehicle according to these values and sets `DRIVER_DATA_VEH_REL_TARGET_LANE` to zero when the middle of the front end of the vehicle has reached the target lane and sets `DRIVER_DATA_VEH_LANE` to the new lane number at this time, too.
   (If there is no new lane on this side, Vissim sets the vehicle back to the middle of the old lane and stops the lane change itself, setting `DRIVER_DATA_ACTIVE_LANE_CHANGE` and `DRIVER_DATA_DESIRED_LANE_ANGLE` to zero.)
3. The driver model DLL has to determine itself when the lane change is over – it has to reset the values for `DRIVER_DATA_ACTIVE_LANE_CHANGE` and `DRIVER_DATA_DESIRED_LANE_ANGLE` to zero.

Hints:

- The driver model has full control in this mode – it can even interrupt a current lane change.
- Vissim does not finish a lane change on its own in this mode. This means that the vehicle will continue moving laterally even beyond the next lane if the driver model does not stop the lane change.

5 Multithreading

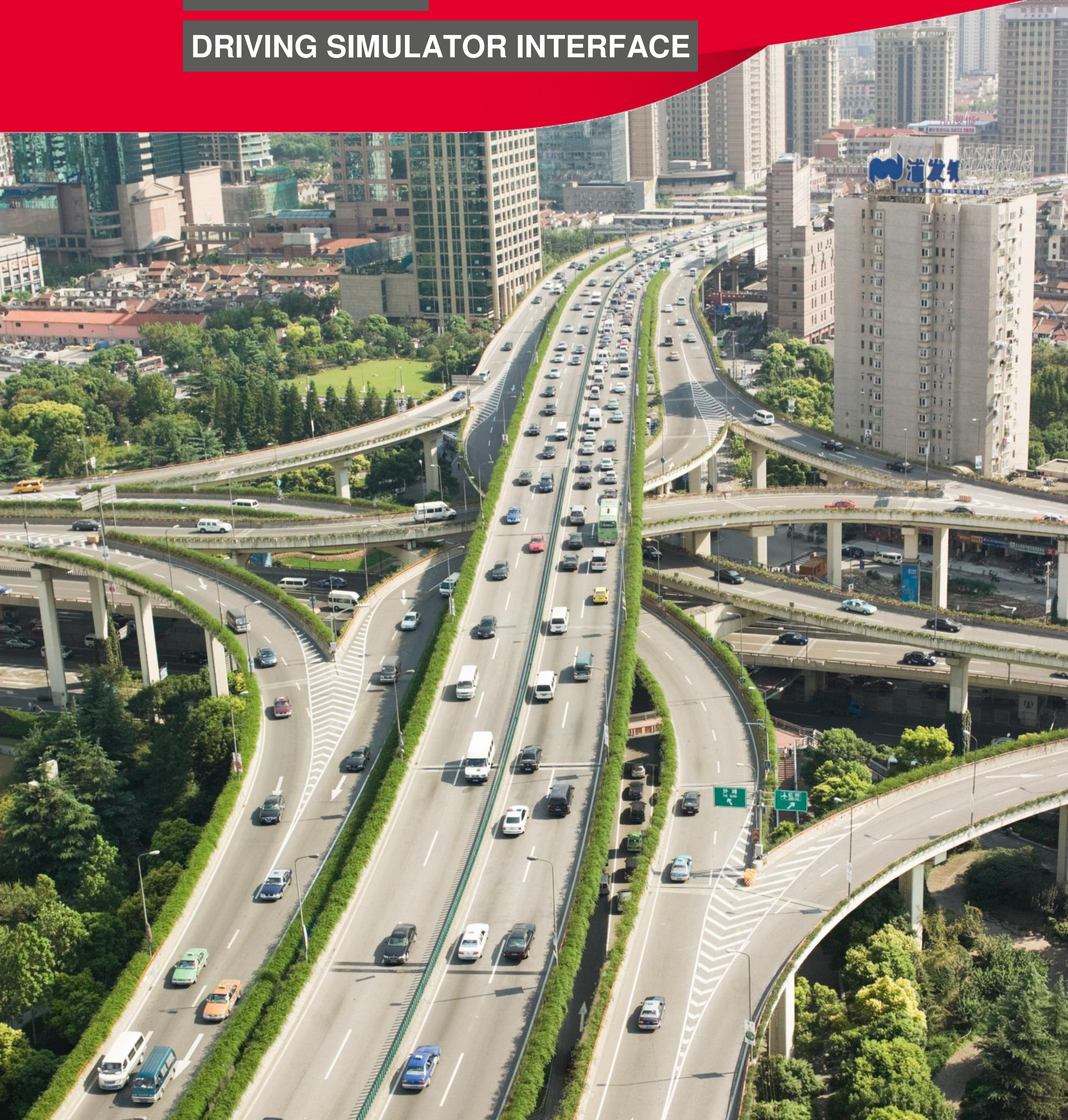
If multiple cores are used for a simulation run, vehicles on different Vissim links can be handled by different threads. The assignment of links to these threads can be different in each time step. The DLL functions are called during one time step in a non-deterministic sequence from multiple threads, i.e. possibly alternating and overlapping for different subject vehicles. So the data can only be assigned correctly if thread-local storage is used (instead of global variables which are fine for singlethreaded use) or if there is only one subject vehicle (using that DLL) or if all subject vehicles are on the same link.

PTV GROUP

the mind of movement

PTV VISSIM API

DRIVING SIMULATOR INTERFACE



Imprint

This documentation is based on Vissim 10.00-01.

PTV Planung Transport Verkehr AG
Haid-und-Neu-Str. 15
D-76131 Karlsruhe, Germany

Tel.: +49 (0)721-9651-0
Fax: +49 (0)721-9651-699
info@ptvgroup.com
www.ptvgroup.com

Disclaimer

Limited Warranty

Although this documentation was compiled with great care, we cannot guarantee for its correctness. We are thankful for hints on errors or shortcomings.

Contents

1	**Introduction**	**4**
2	**Architecture**	**5**
3	**DLL Interface Details**	**6**
4	**Using the DLL**	**11**
5	**Examples**	**12**
5.1	DrivingSimulatortextClient	12
5.2	Unity	12

1 Introduction

The PTV Vissim add-on module "Driving Simulator Interface" allows to connect Vissim to a driving (cycling, walking) simulator (DS). That DS can either be simulator hardware used by a human or a piece of software representing the algorithms of an autonomous vehicle (or multiple vehicles).

Vissim provides the surrounding traffic (vehicles, bicycles, pedestrians) to be visualized in the DS, and the DS passes back the current position and orientation of the simulator vehicle(s) (bicycle(s) / pedestrian(s)). The vehicles and pedestrians in the Vissim network react to this simulator data as to all other vehicles and pedestrians in the microscopic simulation model. In addition, Vissim passes traffic signal states to the DS for visualization, and the DS can set detectors in Vissim explicitly in order to affect the signalization.

The DS does not need to know the Vissim data model where the network is modeled from links, connectors, areas, ramps and obstacles. The DS needs to have its own world model (for simulation and visualization). As all vehicle and pedestrian positions are exchanged in cartesian world coordinates (x/y/z), the DS must be able to provide/use such coordinates, and the coordinates of the networks on both sides (Vissim / DS) must match precisely.

Modifications since Vissim 10.00-00 are highlighted like this in this document.

2 Architecture

The DS software must connect to a Windows DLL provided by PTV. This DLL communicates with a Vissim instance on the same computer through shared memory. If the DS software doesn't run on Windows, it needs to connect to a Windows application to be built by the user through sockets or any other means of process communication.

The start of the Vissim instance is triggered through a function call from the DLL, with the desired Vissim version and the network file name to be loaded as parameters. Additional optional parameters allow to set the frequency of the simulator, a visibility radius, the maximum number of data sets for vehicles, pedestrians, signal groups and detectors to be passed in one or both directions. Vissim is started through the COM interface, so it needs to be registered as COM server. In the Vissim network settings, the option "driving simulator" needs to be activated, and a vehicle type must be selected to be used for the DS vehicles. After loading the network, Vissim establishes the communication to the DLL and waits for the starting location of the DS vehicle (in world coordinates, fitting with the Vissim network) to be passed from the DLL.

Vissim determines the most probable location for the DS vehicle in the Vissim network by selecting from all links at the passed world coordinates the link with the driving direction closest to the orientation of the DS vehicle. After the vehicle has been placed at that location in the Vissim model, the first time step of the Vissim simulation is executed. After that time step, Vissim passes the world coordinates of all vehicles in the network (except the DS vehicle) to the DLL and waits for the new location of the DS vehicle. The simulator can now read the new locations of the Vissim vehicles, visualize them and then pass back the new location of the DS vehicle. After this, Vissim immediately calculates the next time step, and so on.

Before the DS passes the new location of the simulator vehicle to Vissim, it can optionally set detector states. Before or after retrieving the new locations of the Vissim vehicles, the DS can optionally retrieve the signal states passed from Vissim and/or a list of all vehicles which have entered the Vissim network in this time step, a list of all vehicles which have left the Vissim network in this time step and a list of all vehicles which have changed their location in this time step.

This data exchange between Vissim and the DS must be executed at least once per Vissim simulation time step. If the simulator runs at a higher frequency (frame rate) than Vissim, the interface provides automatic interpolation of the positions of the Vissim vehicles and pedestrians between Vissim simulation time steps.

The timing of the co-simulation is controlled by the DS. It can run faster than real time (if this is possible for the DS) but not faster than a stand-alone Vissim simulation of the same network. (The Vissim simulation of a time step can take much less or much more than real time, depending on the used hardware and the size of the network. A big network might not be possible to be simulated with very short time steps in real time.) The simulation speed in Vissim needs to be set to “maximum”, and the visualization in Vissim should be switched off in order to achieve the highest possible simulation speed. The DS can slow down the co-simulation as much as it requires (usually to exactly real time) without hurting the Vissim simulation at all.

3 DLL Interface Details

These functions and data types are provided by this DLL:

```
bool VISSIM_Connect (unsigned short versionNo,
                     wchar_t const *networkFileName,
                     unsigned short simulatorFrequency =   10,
                     double         visibilityRadius   = -1.0,
                     unsigned short maxSimulatorVeh    = 1000,
                     unsigned short maxSimulatorPed    = 1000,
                     unsigned short maxSimulatorDet    =  100,
                     unsigned short maxVissimVeh       = 1000,
                     unsigned short maxVissimPed       = 1000,
                     unsigned short maxVissimSigGrp    = 1000);
```

/* Establishes a connection between the DLL and a GUI version of Vissim.
This starts Vissim and passes the global data handles for the shared memory to Vissim.
Returns true upon success, else false.

unsigned short versionNo is the Vissim version to be used, e.g. 1000 for Vissim 10.
wchar_t const* networkFileName is the .inpx Vissim traffic network filename to start Vissim with.

Optional parameters:
unsigned short simulatorFrequency is the frequency of the simulator (frames/sec).
double visibilityRadius is the maximum distance of a Vissim veh/ped from a simulator veh/ped to be seen by the simulator (<= 0 means unlimited radius).
unsigned short maxSimulatorVeh is the maximum number of simulator vehicles (to be passed to Vissim).
unsigned short maxSimulatorPed is the maximum number of simulator pedestrians (to be passed to Vissim).
unsigned short maxSimulatorDet is the maximum number of simulator detector impulses (to be passed to Vissim).
unsigned short maxVissimVeh is the maximum number of Vissim vehicles (to be passed to the simulator).
unsigned short maxVissimPed is the maximum number of Vissim pedestrians (to be passed to the simulator).
unsigned short maxVissimSigGrp is the maximum number of Vissim signal groups (to be passed to the simulator). */

```
bool VISSIM_ConnectToConsole (wchar_t const *consoleFileName,
                              wchar_t const *networkFileName,
                              unsigned short simulatorFrequency =   10,
                              double         visibilityRadius   = -1.0,
                              unsigned short maxSimulatorVeh    = 1000,
                              unsigned short maxSimulatorPed    = 1000,
                              unsigned short maxSimulatorDet    =  100,
                              unsigned short maxVissimVeh       = 1000,
                              unsigned short maxVissimPed       = 1000,
                              unsigned short maxVissimSigGrp    = 1000);
```

/* Establishes a connection between the DLL and a console version of Vissim.
This starts Vissim and passes the global data handles for the shared memory to Vissim.
Returns true upon success, else false.

string* consoleFileName is the location where the console version can be found.
string* networkFileName is the .inpx Vissim traffic network filename to start Vissim with.

Optional parameters: see above for VISSIM_Connect(). */

```
bool VISSIM_Disconnect (void);
```

/* Stops the simulation run in Vissim, closes Vissim and disconnects the DLL from Vissim.
Returns true upon success, else false. */

```
struct Simulator_Veh_Data {
  double Position_X;     /* front center of the vehicle in m */
  double Position_Y;     /* front center of the vehicle in m */
  double Position_Z;     /* front center of the vehicle in m */
  double Orient_Heading; /* in radians,
                            eastbound = zero, northbound = +Pi/2 */
  double Orient_Pitch;   /* in radians, uphill = positive */
  double Speed;          /* in m/s */
};

bool VISSIM_SetDriverVehicles (int Num_Vehicles, Simulator_Veh_Data *VehicleData);
```

/* To be called once per simulator frame if there are no simulator pedestrians (only vehicles).
VehicleData[0] to VehicleData[Num_Vehicles - 1] are passed from the DLL to Vissim.
Vissim starts the calculation of the next time step immediately after receipt of this call.
Num_Vehicles must be in [0..maxSimulatorVeh] – if it is zero, no simulator vehicle data is passed. Returns true upon success, else false. */

```
struct Simulator_Ped_Data {
  double Position_X;            /* in m */
  double Position_Y;            /* in m */
  double Position_Z;            /* in m */
  double Orient_Heading;        /* in radians */
  double DistanceSinceBirth;    /* in m */
  double Speed;                 /* in m/s */
};

bool VISSIM_SetDriverPedestrians (int Num_Pedestrians,
                                  Simulator_Ped_Data *PedestrianData);
```

/* To be called once per simulator frame if there are no simulator vehicles (only pedestrians).
PedestrianData[0] to PedestrianData[Num_Vehicles - 1] are passed to Vissim.
Vissim starts the calculation of the next time step immediately after receipt of this call.
Num_Pedestrians must be in [0..maxSimulatorPed] – if it is zero, no simulator pedestrian data is passed. Returns true upon success, else false. */

```
bool VISSIM_SetDriverVehiclesAndPedestrians (int Num_Vehicles,
                                             Simulator_Veh_Data *VehicleData,
                                             int Num_Pedestrians,
                                             Simulator_Ped_Data *PedestrianData);
```

/* To be called once per simulator frame if there are simulator vehicles and simulator pedestrians.
VehicleData[0] to VehicleData[Num_Vehicles - 1] and PedestrianData[0] to PedestrianData[Num_Vehicles - 1] are passed to Vissim.
Vissim starts the calculation of the next time step immediately after receipt of this call.
Num_Vehicles must be in [0..maxSimulatorVeh] and Num_Pedestrians in [0.. maxSimulatorPed] – if a value is zero, no simulator vehicle data respectively pedestrian data is passed.
Returns true upon success, else false. */

```
bool VISSIM_SetDetection (long DetectorID, long ControllerID);
```

/* Can be called before the call of VISSIM_SetDriverVehicles(), VISSIM_SetPedestrians() or VISSIM_SetDriverVehiclesAndPedestrians().
Causes the Vissim detector <DetectorID> of the controller <ControllerID> to behave as if a vehicle or pedestrian arrives on the detector during the next time step and leaves the detector during the subsequent time step. Returns true upon success, else false. */

```
bool VISSIM_DataReady ();
```

/* *Can* be called to determine if Vissim has completed the calculation of the time step (return value = true) or not yet (return value false). Does not block. */

```
#define NAME_MAX_LENGTH 100

struct VISSIM_Veh_Data {
  long   VehicleID;
  long   VehicleType;              /* vehicle type number from Vissim */
  char   ModelFileName[NAME_MAX_LENGTH]; /* .v3d */
  long   color;                    /* RGB */
  double Position_X;               /* front center of the vehicle in m */
  double Position_Y;               /* front center of the vehicle in m */
  double Position_Z;               /* front center of the vehicle in m */
  double Orient_Heading;           /* in radians,
                                      eastbound = zero, northbound = +Pi/2 */
  double Orient_Pitch;             /* in radians, uphill = positive */
  double Speed;                    /* in m/s */
  long   LeadingVehicleID;         /* relevant vehicle in front */
  long   TrailingVehicleID;        /* next vehicle back on the same lane */
  long   LinkID;                   /* Vissim link attribute "Number" */
  char   LinkName[NAME_MAX_LENGTH]; /* empty if "Name" not set in Vissim */
  double LinkCoordinate;           /* in m */
  int    LaneIndex;                /* 0 = rightmost */
  int    TurningIndicator;         /* 1 = left, 0 = none, -1 = right */
  long   PreviousIndex;            /* for interpolation: index in the array
                                      in the previous Vissim time step,
                                      < 0 = new in the visibility area */
};

VISSIM_GetTrafficVehicles (int *Num_Vehicles, VISSIM_Veh_Data **VehicleData);
```

/* To be called once per simulator frame in order to get new data of Vissim vehicles.
*Num_Vehicles is set to the number of all Vissim vehicles (excluding simulator vehicle(s)) in the network; (*VehicleData)[0] to (*VehicleData)[*Num_Vehicles - 1] contain their data.
Doesn't block after VISSIM_DataReady() has returned true.
Blocks while the calculation of the time step in Vissim hasn't finished yet. */

```
struct VISSIM_Ped_Data {
  long   PedestrianID;
  long   PedestrianType;                  /* Vissim pedestrian type number */
  char   ModelFileName[NAME_MAX_LENGTH];  /* *.v3d */
  double Length;                          /* in m */
  double Width;                           /* in m */
  double Height;                          /* in m */
  double Position_X;                      /* in m */
  double Position_Y;                      /* in m */
  double Position_Z;                      /* in m */
  double Orient_Heading;                  /* in radians */
  double Orient_Pitch;                    /* in radians */
  double DistanceSinceBirth;              /* in m */
  double Speed;                           /* in m/s */
  Pedestrian_Motion_State_Type MotionState; /* see header file */
  Pedestrian_Construction_Element_Type ConstructionElementType;
                                          /* see header file */
  long   ConstructionElementID;           /* the construction element the
                                             pedestrian is currently on */
  char   ConstructionElementName[NAME_MAX_LENGTH];
                                          /* empty if not set in Vissim */
  long   PreviousIndex;                   /* for interpolation: index in the
        array in the previous Vissim time step, < 0 = new in the visibility area */
};

VISSIM_GetTrafficPedestrians (int *Num_Pedestrians,
                              VISSIM_Ped_Data **PedestrianData);
```

/* To be called once per simulator frame in order to get new data of Vissim pedestrians.
*Num_Pedestrians is set to the number of all Vissim pedestrians (excluding simulator pedestrian(s)) in the network; (*PedestrianData)[0] to (*PedestrianData)[*Num_Pedestrians - 1] contain their data.
Doesn't block after VISSIM_DataReady() has returned true.
Blocks while the calculation of the time step in Vissim hasn't finished yet. */

```
VISSIM_GetVehicleLists (int *NumNewVehicles, long **NewVehicleIds,
                        long **NewVehicleTypes, int *NumMovedVehicles,
                        long **MovedVehicleIds, int *NumDeletedVehicles,
                        long **DeletedVehicleIds);
```

/* Can be called once per simulator frame in order to get filtered lists of Vissim vehicle numbers.
*NumNewVehicles is set to the number of new Vissim vehicles created in the last time step,
(*NewVehicleIds)[0] to (*NewVehicleIds)[NumNewVehicles - 1] contain their IDs,
(*NewVehicleTypes)[0] to (*NewVehicleTypes)[NumNewVehicles - 1] contain their vehicle type numbers;
*NumMovedVehicles is set to the number of Vissim vehicles that have moved in the last time step,
(*MovedVehicleIds)[0] to (*MovedVehicleIds)[NumMovedVehicles - 1] contain their IDs,
*NumDeletedVehicles is set to the number of Vissim vehicles that have been deleted in the last time step,
(*DeletedVehicleIds)[0] to (*DeletedVehicleIds)[NumDeletedVehicles - 1] contain their IDs.
Doesn't block after VISSIM_DataReady() has returned true. Might block before this. */

```
VISSIM_GetPedestrianLists (int *NumNewPedestrians, long **NewPedestrianIds,
                           long **NewPedestrianTypes, int *NumMovedPedestrians,
                           long **MovedPedestrianIds, int *NumDeletedPedestrians,
                           long **DeletedPedestrianIds);
```

/* Can be called once per simulator frame in order to get filtered lists of Vissim pedestrian numbers.
*NumNewPedestrians is set to the number of new Vissim pedestrians created in the last time step,
(*NewPedestrianIds)[0] to (*NewPedestrianIds)[NumNewPedestrians - 1] contain their IDs,
(*NewPedestrianTypes)[0] to (*NewPedestrianTypes)[NumNewPedestrians - 1] contain their pedestrian type numbers;
*NumMovedPedestrians is set to the number of Vissim pedestrians that have moved in the last time step,
(*MovedPedestrianIds)[0] to (*MovedPedestrianIds)[NumMovedPedestrians - 1] contain their IDs,
*NumDeletedPedestrians is set to the number of Vissim pedestrians that have been deleted in the last time step,
(*DeletedPedestrianIds)[0] to (*DeletedPedestrianIds)[NumDeletedPedestrians - 1] contain their IDs.
Doesn't block after VISSIM_DataReady() has returned true. Might block before this. */

```
struct VISSIM_Sig_Data {
  long ControllerID;
  long SignalGroupID;
  int  SignalState; /* 1 = Red, 2 = Red+Amber, 3 = Green, 4 = Amber,
                       5 = Off (black), 6 = Undefined, 7 = Flashing Amber,
                       8 = Flashing Red, 9 = Flashing Green,
                      10 = Alternating Red/Green, 11 = Green+Amber */
};

VISSIM_GetSignalStates (int *NumSignals,
                        VISSIM_Sig_Data **SignalStateData);
```

/* Can be called once per simulator frame in order to get new data of Vissim signal groups.
Doesn't block after VISSIM_DataReady() has returned true. Might block before this. */

If the simulator frequency is identical with the Vissim simulation resolution, the data passed from Vissim through VISSIM_Get*() functions doesn't need to be copied to a different memory location before VISSIM_SetDriverVehicles() is called again because Vissim uses two alternating buffers.

If the simulator frequency is a multiple of the Vissim simulation resolution, the interface uses linear interpolation to estimate the position, orientation and speed values of the Vissim vehicles/pedestrians between Vissim time steps. In this case, the same buffer is used for all interpolation time steps (i.e. the pointer passed from a VISSIM_Get*() function stays the same)! In the very first time step of a Vissim vehicle/pedestrian inside the network, there is no interpolation (because a previous position is not available), so the position, orientation and speed have constant values in all simulator frames.

4 Using the DLL

In order to use the provided DrivingSimulatorProxy.dll which contains the functions listed above, a Windows program needs to be built which includes DrivingSimulatorProxy.h and is linked with DrivingSimulatorProxy.lib in order to be able to call the functions from the DLL. Two editions of the *.lib and *.dll files are provided, one each for a 32-bit and a 64-bit program. Both have been built with the Platform Toolset Visual Studio 2013 (v120), so Visual Studio 2013 or later can be used to build the simulator program.

5 Examples

5.1 DrivingSimulatortextClient

A simple example "DrivingSimulatorTextClient" is provided with this API package, including the example network file driving_simulator_test.inpx and some additional files:

This “simulator” is a Windows console application. It does not calculate vehicle positions nor does it read actions from a human driver but it simply reads a vehicle record file of a single vehicle (driving_simulator_test.fzp, originally produced by Vissim 5.30 in a normal simulation run of the same network with 10 time steps per second) and sends the coordinates of that vehicle (number 1) to Vissim, so the Vissim vehicles can react on the simulator vehicle. In addition, this text client example reads a pedestrian record file of a single pedestrian (driving_simulator_test.pp, originally produced by Vissim 10 in a normal simulation run of the same network with 10 time steps per second) and sends the coordinates of that pedestrian (number 1) to Vissim, so the Vissim pedestrians can react on the simulator pedestrian. (In this example network, vehicles and pedestrians have no overlapping links, so they don’t react on each other.)

In order to run this test client, the environment variable DRIVING_SIMULATOR_INTERFACE must be set to the absolute path to the folder containing the example network file driving_simulator_test.inpx. The client needs two command line parameters: first the desired “simulator frequency” and second the desired visibility radius.

When you start the test client, it starts a new Vissim 10 instance through VISSIM_Connect() (unless there is already a running Vissim instance which has been started with the command line parameter -automation). This Vissim instance loads the network file and starts the simulation run automatically. During the initialization of the simulation the DrivingSimulatorProxy.dll tries to connect with Vissim. At the end of the vehicle record file, the simulation run is stopped and Vissim is closed automatically.

5.2 Unity

A simple Unity application using the DrivingSimulatorProxy.dll for a co-simulation with Vissim is provided in the subdirectory Unity\, including specific documentation.