A tutorial on External Driver Models in PTV VISSIM®

-

Gopikrishnan Nair Suresh Kumar
Georgia Institute of Technology

Table of Contents

1 Introduction

PTV VISSIM® is a traffic microsimulation platform with the capability to model traffic flows and analyze microscopic behaviors in a network. The simulation models can be enhanced by using an External Driver Model (EDM). The External Driver Model provides users with a way to customize driving behaviors and execute certain vehicle maneuvers at will. With careful manipulation of vehicle parameters, the modeling of anticipatory and intelligent driver models can be made possible on VISSIM®. The EDMs also give users freedom on situational control of vehicles, such as executing lane changes to make way for emergency vehicles etc. Such flexibility in modeling greatly expands the scope and potential of VISSIM® simulations. In this regard, the following report is intended to provide users with a background on the workings of EDMs, their capabilities and limitations. Also accompanying the tutorial is a folder containing the code template and project files for the EDMs. This tutorial is recommended to be consulted alongside the External Driver Model Documentation as provided by PTV.

2 Software and Platform Requirements

The software required for the EDM are the following:

1. PTV VISSIM®:
   The tutorial was developed and tested on VISSIM® 21. The necessary files and documentation for External Driver Models should also be available with the VISSIM® installation in the following directory: "*YourDrive:\Program Files\PTV Vision\PTV Vissim 2021\API\DriverModel_DLL".*

2. Microsoft Visual Studio IDE
   MS Visual Studio provides a platform for editing, building, and deploying applications with convenience. The user must ensure that the C++ development tools are also installed as the EDMs are built with C++ programming language.

A brief guide to setting up the C++ project for building custom models is described below:

1. The *.vcxproj* file is an XML file linking the C++ code, the necessary headers and other information needed to build the project. The file is to be opened using the latest *Microsoft Visual Studio IDE* for making any changes to the driver model code.

2. Opening the project file will launch the Solution Explorer window which displays the contents of the project including the codebase and the header files necessary (Fig 1). The "*DriverModel.cpp*" file contains the C++ code necessary to build a driver model and will be the primary focus of this tutorial. Double-clicking the file will open the code in a new tab.

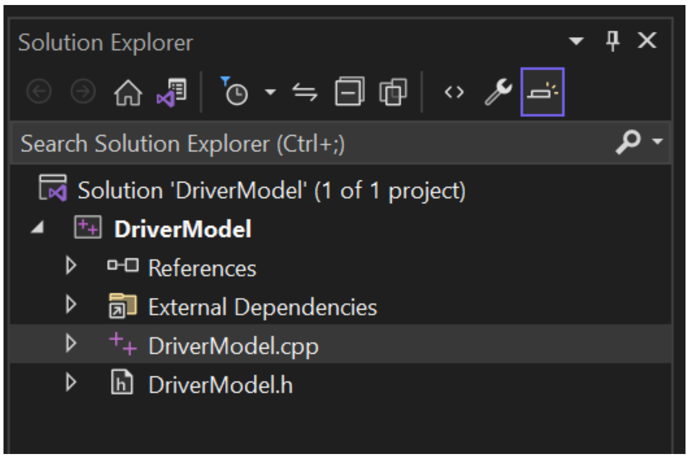
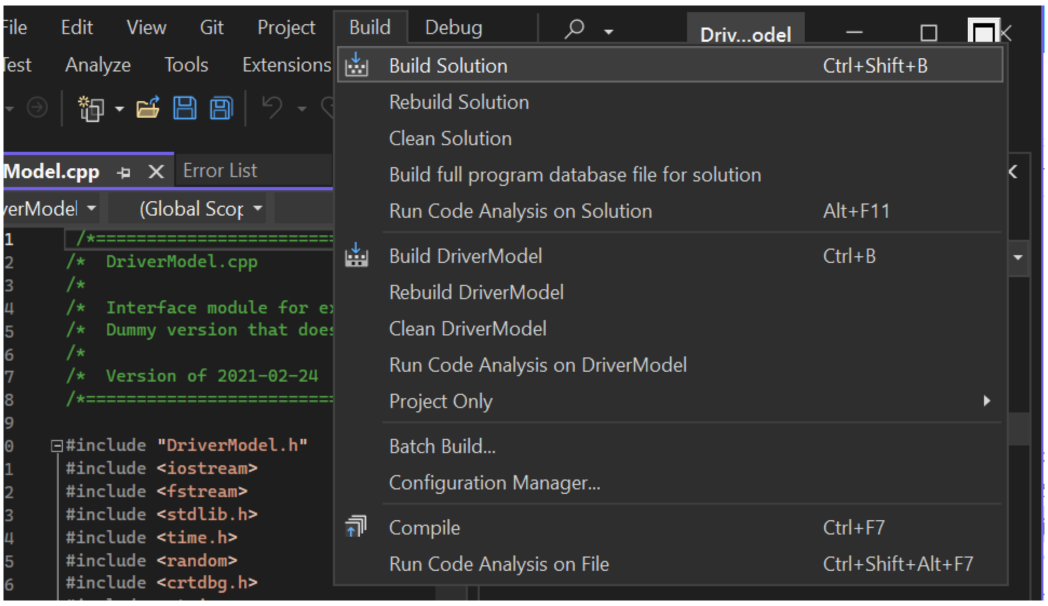


Figure 1. Solution Explorer window in Visual Studio

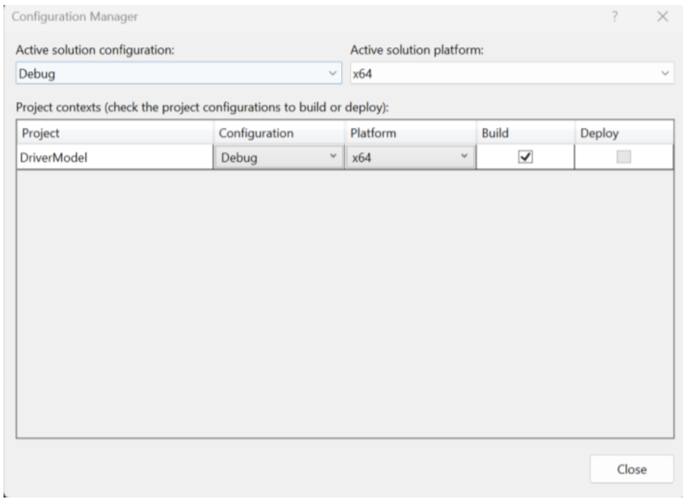

File Edit View Git Project Build Debug
Test Analyze Tools Extensions
Driv...odel
Model.cpp
Error List
erModel
(Global Scop
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Build full program database file for solution
Run Code Analysis on Solution Alt+F11
Build DriverModel Ctrl+B
Rebuild DriverModel
Clean DriverModel
Run Code Analysis on DriverModel
Project Only
Batch Build...
Configuration Manager...
Compile Ctrl+F7
Run Code Analysis on File Ctrl+Shift+Alt+F7

```
 /*========================
/*  DriverModel.cpp
/*
/*  Interface module for e
/*  Dummy version that doe
/*
/*  Version of 2021-02-24
/*========================

#include "DriverModel.h"
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <time.h>
#include <random>
#include <crtdbg.h>
```

Figure 2. The Build Solution option under the Build menu

3. For implementing any custom driver models, the project must be rebuilt after every change to the code using Visual Studio. To do this, the user must select the "*Build Solution*" option which can be found under the "*Build*" menu on the main toolbar. (Fig 2)
4. The user must also ensure the build platform matches their PC processor architecture (x86, x64 etc.), which can be specified in the configuration manager window under the *"Build"* options in Visual Studio (Fig 3). The rebuilt solution is in the form of a dynamically linked list (DLL) which would be available in the directory specified in the output window in Visual Studio. An example of the output window after a successful build is shown in Figure 4.

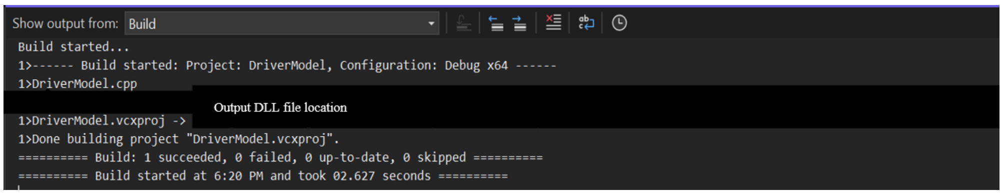


Active solution configuration:	Active solution platform:
Debug	x64

Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build	Deploy
DriverModel	Debug	x64	✓	

Figure 3. Build Platform Configuration Window

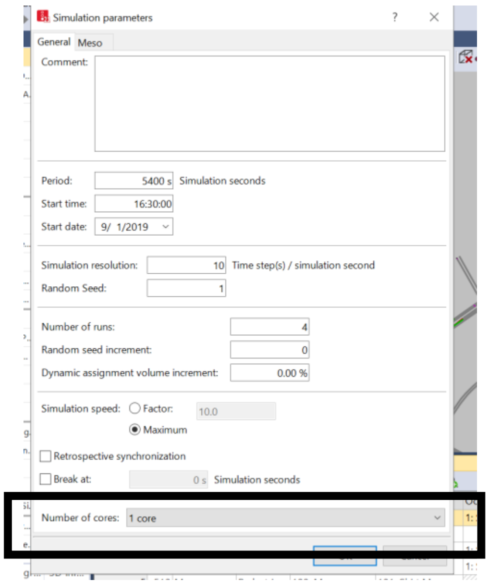

Show output from: Build

```
Build started...
1>------ Build started: Project: DriverModel, Configuration: Debug x64 ------
1>DriverModel.cpp
                              Output DLL file location
1>DriverModel.vcxproj ->
1>Done building project "DriverModel.vcxproj".
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
========== Build started at 6:20 PM and took 02.627 seconds ==========
```

Figure 4. Output Window

3 VISSIM® Model Parameters

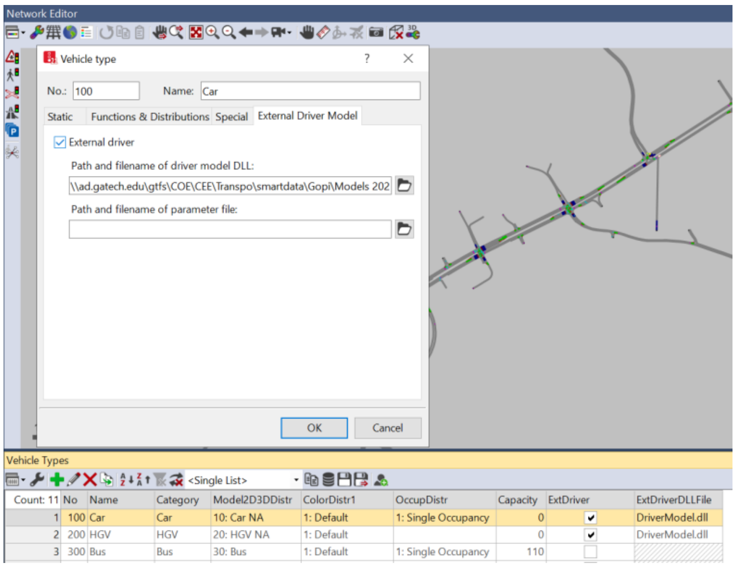
As recommended by PTV, the use of global variables in the code limits the simulation to be run only on a single core. The use of multiple cores speeds up the model, but for the data to be assigned correctly, thread local storage would be needed. This usage has not been explored in this tutorial, and the simulation has been constrained to a single core (Fig 5).



Comment:		
Period:	5400 s	Simulation seconds
Start time:	16:30:00	
Start date:	9/ 1/2019	
Simulation resolution:	10	Time step(s) / simulation second
Random Seed:	1	
Number of runs:	4	
Random seed increment:	0	
Dynamic assignment volume increment:	0.00 %	
Simulation speed:	Factor: 10.0 [✓ Maximum]	
☐ Retrospective synchronization		
☐ Break at:	0 s	Simulation seconds
Number of cores:	1 core	

Figure 5. Simulation Cores set to Single

The DLL file is linked to each of the appropriate vehicle types within VISSIM® as seen in Figure 6. The use of External Driver must also be ensured by filling in the checkbox. It is recommended that prior to running the model on any new directory, this DLL link process must be specified again. The directory does not update automatically with the transfer of models.



No.: 100 Name: Car

[✓] External driver

Path and filename of driver model DLL: \\ad.gatech.edu\gtfs\COE\CEE\Transpo\smartdata\Gopi\Models 202

Path and filename of parameter file:

Count: 11	No	Name	Category	Model2D3DDistr	ColorDistr1	OccupDistr	Capacity	ExtDriver	ExtDriverDLLFile
1	100	Car	Car	10: Car NA	1: Default	1: Single Occupancy	0	✓	DriverModel.dll
2	200	HGV	HGV	20: HGV NA	1: Default		0	✓	DriverModel.dll
3	300	Bus	Bus	30: Bus	1: Default	1: Single Occupancy	110	☐	

Figure 6. Linking the DLL to the vehicle type

4 Data Availability

The External Driver Model has access to numerous levels of data but with certain constraints on them. The data is not universal and is mostly limited to what the vehicle currently experiences or what the vehicle may perceive up to a limited range They are discussed in brief in this section and a few examples are also provided.

4.1 Global Data

The specifics of the simulation run can be accessed via EDM. This includes the current simulation time and timestep length.

4.2 Vehicle Specific Data

Vehicle parameters such as vehicle ID, type, position, speed, acceleration etc. are readable by the EDM. Any user defined attributes can also be accessed for each vehicle. Besides this a vehicle can also retrieve the same data for vehicles in its proximity. This allows the user to keep a track of its neighboring vehicles and their actions.

```
#VEH – Current Vehicle; NVEH - Nearby vehicle)
case DRIVER_DATA_VEH_LATERAL_POSITION
case DRIVER_DATA_VEH_VELOCITY
case DRIVER_DATA_VEH_ACCELERATION
case DRIVER_DATA_NVEH_LATERAL_POSITION
case DRIVER_DATA_NVEH_VELOCITY
case DRIVER_DATA_NVEH_ACCELERATION
```

4.3 Data of the current link/lane

The global coordinates of the current roadway link, and its physical characteristics are also available to the EDM. The width and end distance of the lanes can be referred with the following functions.

```
case DRIVER_DATA_LANE_WIDTH
case DRIVER_DATA_LANE_END_DISTANCE
```

4.4 Data of Upcoming Environment

Depending on the roadway design, information such as curve radius, the slope etc. are also made available to the user with the following commands.

```
case DRIVER_DATA_MIN_RADIUS
case DRIVER_DATA_DIST_TO_MIN_RADIUS
case DRIVER_DATA_SLOPE
```

4.5 Data of upcoming Infrastructure

Data regarding the next signals, priority rules, stop heads are also observable by the vehicles. Actions can be suggested ahead of time by analyzing the upcoming road rules and conditions.

```
case DRIVER_DATA_SIGNAL_STATE
case DRIVER_DATA_SIGNAL_STATE_START
case DRIVER_DATA_SPEED_LIMIT_DISTANCE
case DRIVER_DATA_SPEED_LIMIT_VALUE
case DRIVER_DATA_PRIO_RULE_DISTANCE
```

5 User Defined Attributes

VISSIM® and EDM share only a select subset of information directly. These include coordinates, speeds, etc. The full list is available in the "***Interface_Description.pdf***" file published by PTV and enclosed in the Driver Model folders. Aside from these, any other information regarding the simulation model is essentially invisible to the EDM. Moreover, the EDM is designed to run every timestep, and as a result does not retain any memory. It is not possible to store any data for future uses within the code itself.

To circumvent both these limitations, we propose using User Defined Attributes (UDAs) within VISSIM® to bring more flexibility to the modeling process. The EDMs have the capability to read and write the values of previously defined UDAs. So, any user could use these UDAs as placeholders for storing any data across timesteps, or even pass data regarding the network that would otherwise be inaccessible to the EDM. Each UDA which is used must be initialized with the following command, where the index1 specifies the ID of the UDA that is to be used.

```
case DRIVER_DATA_USE_UDA:
        if (index1 == 1)
                {
                        return 1;
                }
```

6 Lane Changes

A primary task of an EDM is to control the lane changes that may happen as part of the custom driving behavior. The EDMs allow two modes of lane change behavior: Simple and Full Control. Once a user requests a lane change, the simple mode allows VISSIM® to take control over the vehicle and hands the control back after completion of the lane change. The Full Control gives the user the ability to define every parameter of the maneuver. This includes the target lane and the angle of approach.

The modes can be selected by specifying the value of the commands below.

```
case DRIVER_DATA_SIMPLE_LANECHANGE :
  *int_value = 1; (1- Simple Lane Change; 0 - Full Control)

case DRIVER_DATA_WANTS_SUGGESTION :
  *int_value = 1; (0 – Will not accept suggestion from EDM
```

A lane change to the right or left can be then initiated with the command:

```
active_lane_change = 1;(1 to the right Or -1 to the left)
```

7 Code Structure

The EDM code environment is defined as follows:

7.1 Global Variables

All necessary variables names are to be declared globally. Separate variables are to be defined for each vehicle that any other vehicle can see (By default it can see two vehicles upstream and downstream and two lanes to either side). The data types to be used (*double, float, long etc)* for the variables are specified in the **DriverModel.h** file, and are to be defined as shown here:

```
double  current_velocity;
double  current_acceleration;
double  vehicle_length;
```

7.2 SetValue()

The declared variables are assigned values that are passed along from VISSIM®. They are linked to their data values and types under each relevant switch case. An example of this is described below. The previously defined variables are assigned to their respective data from VISSIM®. The data types should always match the definitions and the types in the **DriverModel.h** file

```
case DRIVER_DATA_VEH_TYPE              :
          current_vehicle_type = int_value;
          return 1;
case DRIVER_DATA_VEH_COLOR             :
     vehicle_color = int_value;
     return 1;
```

7.3 GetValue()

The GetValue function serves as a return function from the DLL to VISSIM®. The actions and data as suggested by the DLL is retrieved using the GetValue function. As the value is being sent back to VISSIM®, the variables and their types should be linked as given below

```
case DRIVER_DATA_VEH_TURNING_INDICATOR :
*int_value = turning_indicator;
return 1;
case DRIVER_DATA_VEH_DESIRED_VELOCITY   :
*double_value = desired_velocity;
return 1;
```

7.4 ExecuteCommand()

The ExecuteCommand function encompasses all the processes involved in the movement of a vehicle in the simulation. The function is responsible for initializing the DLL, the creation of a vehicle in the network, the movement of a vehicle during the simulation, and the memory deallocation when a vehicle leaves the network. The custom calculations, condition checks and the maneuver suggestions are to be specified within this function. As the code is executed every timestep, the condition checks are also repeated with no memory of the prior timestep.

When referencing nearby vehicles and their data, it is important to correctly specify their relative position to the vehicle under observation. That vehicles "see" two vehicles in front of and behind, and two lanes to either side (i.e., expected behaviors will not be visible instantly further downstream or upstream). The indices which are used to refer to each visible vehicle are as shown in Fig 2. This area of effect can be expanded by defining new indices and variables referencing these values.

V(2,2)	V(1,2)	V(0,2)	V(-1,2)	V(-2,2)
V(2,1)	V(1,1)	V(0,1)	V(-1,1)	V(-2,1)
		🟥		
V(2,-1)	V(1,-1)	V(0,-1)	V(-1,-1)	V(-2,-1)
V(2,-2)	V(1,-2)	V(0,-2)	V(-1,-2)	V(-2,-2)

Figure 7. Area of effect with indices as reference

A way to reference the vehicles nearby a target vehicle by specifying relative indices is given below, with separate distance variables already defined globally for each position.

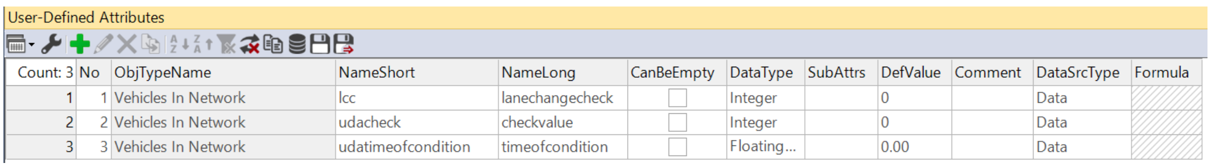
```
case DRIVER_DATA_NVEH_DISTANCE      :
        if (index1 == 0 && index2 == -1)
        {
                behind_vehicle_distance = double_value;
        }
        else if (index1 == 0 && index2 == -2)
        {
                behind2_vehicle_distance = double_value;
        }
        else if (index1 == 0 && index2 == -3)
        {
                behind3_vehicle_distance = double_value;
        }
```

8 Pullover Implementation

In this section, a driver model design utilizing several different features of VISSIM®'s EDM is described. The objective is to simulate a pullover behavior for vehicles when they are in proximity of an emergency vehicle. The emergency vehicle type is specified as 630 in the model and that can be modified. Several distance checks are also implemented and a custom delay is also introduced in the behavior by using UDAs.

8.1 VISSIM® Model Changes

With regards to the modeling of the pullover behavior, three different UDAs have been created to facilitate storing certain variables and values that are necessary for the algorithm. The UDAs were all created with the same object type of "Vehicles in Network." The chosen type does not impact the algorithm, as it is simply a placeholder. The details of the UDAs are shown in Figure 8. The EDM allows the user to refer to the UDA by name or index number, and the code in reference uses index numbers for convenience.



User-Defined Attributes

Count: 3	No	ObjTypeName	NameShort	NameLong	CanBeEmpty	DataType	SubAttrs	DefValue	Comment	DataSrcType	Formula
1	1	Vehicles In Network	lcc	lanechangecheck	☐	Integer		0		Data	
2	2	Vehicles In Network	udacheck	checkvalue	☐	Integer		0		Data	
3	3	Vehicles In Network	udatimeofcondition	timeofcondition	☐	Floating...		0.00		Data	

Figure 8. User Defined Attributes for Pullover Algorithm

```
case DRIVER_DATA_VEH_UDA:
    if (index1 == 1)
    {
        udalanechange = int_value;
    }
    else if (index1 == 2)
    {
        udacheck = int_value;
    }
    else if (index1 == 3)
    {
        udatime = double_value;
    }
                return 1;
```

8.2 Proximity of vehicles

The primary trigger for initiating pullover behavior is the proximity of vehicles to the ERV. At every timestep, the vehicles check behind them to spot the presence of an ERV. If an ERV is detected, and the distance conditions are met (within 150 ft), the vehicle is directed to perform a lane change to the right Similarly, the action of rejoining the lane will only be executed once a minimum headway distance of (330 ft) is satisfied (Fig 9).

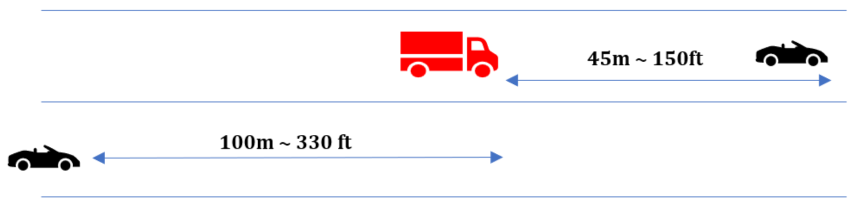


Figure 9. Distance limits between vehicles

```
if (leftbehind_vehicle_type == 630 && leftbehind_vehicle_distance  > -100)
{
        active_lane_change = 0; //Prevents the executing of any lane changes
}


if (behind_vehicle_type == 630 && behind_vehicle_distance > -45 && vehicle_current_lane != 1)
{
        udacheck = 1;//Stores the value 1 for triggering a lanechange after a delay

        udatime = current_time;//Stores the time of triggering to calculate the delay

        udalanechange = 1;//Stores the value of 1 to indicate that the vehicle has already executed a
        lanechange
}
```

8.3 Executing a Lane change

The trigger values are checked at every timestep, and if the values are equal to one a lane change maneuver is initiated. A delay of 2.5 seconds has also been implemented for vehicles slower than 5 mph. The conditions and the action calls are defined below:

```
if (current_velocity < 1.4 && udacheck == 1 && vehicle_current_lane != 1) //Checks for the trigger and
speed limits

{
        if ((current_time - udatime) > abs(2.5) && udacheck == 1) // Wait for current time to reach the
        suggested delay of 2.5 seconds (As speed <5mph or 1.4 kmph)
       {
         udacheck = 0;  // Reset the trigger for lanechange
                active_lane_change = -1; //Execute the Lane change to the right (-1 = right, +1 =left)
       }
}
```

8.4 Observing model behavior

There exist several ways to confirm the successful working of the implemented logic:

1) The user can write into an output file of their choice from within the C++ code to indicate a successful lane change in response to an ERV. Although this allows the user great flexibility and improves the readability of the result, the process of opening and writing into an external file may affect the performance of the simulation. Depending on the size of the model, the simulation speed may be affected severely.
2) Visual observation of the lane change maneuvers during a model run is a quick way to confirm if the conditions and actions are being followed as implemented. The run can be paused or slowed down to see the effects every timestep.
3) Several user defined attributes which were created for the logic can be observed during a visual run. Their values should be either 0 or 1 depending on present conditions.

In most cases, it is recommended to follow a combination of (2) and (3) as it is the easiest to do so. The custom output file method (1) is useful for more descriptive or complex logic which may be hard to confirm visually. Moreover, if the VISSIM® model was launched via a COM script, the run cannot be paused easily.

9 Conclusion

The External Driver Model is a very powerful tool with great potential in microscopic modeling. With the capability of modeling complex custom behaviors any user may require, it is very important to understand the development of the models. Although it does have limitations, it is possible to work around a lot of them as detailed in the tutorial. The tutorial has been developed on the VISSIM® 21 version and it is also advised to refer to the "DriverModel_changes.txt" and "Interface_description.pdf" published by PTV for the future releases of VISSIM®.