SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Interdisciplinary Project

Adapting OSI for the driveBOT: Challenges and Solutions

Authors:	Marvin Heinzelman, Raphael Schilling
Supervisor:	Prof. Dr.-Ing. Eckehard Steinbach
Advisor:	Markus Hofbauer
Submission Date:	July 24, 2023

We confirm that this interdisciplinary project is our own work and we have documented all sources and material used.

Munich, July 24, 2023 Marvin Heinzelman, Raphael Schilling

Abstract

Having realistic simulations is important for testing autonomous cars. For this purpose, cogniBIT develops the so-called *driveBOT*, which steers traffic participants like humans would steer them. This is vital since autonomous cars must be able to drive safely next to human drivers. DriveBOT needs to interact with different simulator tools. We implemented an adapter called *OsiExtractor* that extracts data from the OSI protocol and provides them to the driveBOT. OsiExtractor can receive OSI messages from simulators like *esmini* but also allows driveBOT to send information back to the simulator. In particular, OsiExtractor calculates more complex information from the data represented by OSI. This report describes how OsiExtractor performs these calculations and presents the reasoning behind the chosen approaches. Additionally, we show how the OsiExtractor package can be integrated into other projects like driveBOT.

Contents

1 Introduction

Simulation plays an important role in the development of autonomous vehicles: It provides the only feasible way of testing the safety of autonomous driving systems in complex scenarios. At the same time, self-driving cars must be able to safely interact with human traffic participants. The startup cogniBIT develops the so-called driveBOT. This bot mimics human drivers using a cognitive model which acts based on human-like perception. Since the driveBOT should interact with other simulators, common interfaces need to be implemented. The current standard to exchange data between driving simulators is OSI [1]. The goal of this project is to implement an adapter that provides driveBOT with the information it needs. One challenge with this is, that OSI is a general-purpose simulation interface which provides data in a way that is not directly usable for driveBOT. For example, driveBOT needs the current position on a road, but OSI only provides the current position on a lane. By merging lanes into roads that represent the road network, we can derive the current position on the road. Also, other information like the curvature of the road, the distance to the next highway exit, the current speed limit, etc. has to be calculated. This project is tailored to the OSI data that is sent via UDP by *esmini*, which is an open-source traffic simulator that can run scenarios of the format *OpenSCENARIO*. The contributions of this project are:

- create an OSI adapter with a data model that fits the needs of driveBOT
- calculate complex properties by combining basic information from OSI
- implement functionality to send information back to esmini

This report is structured as follows: In the second chapter, we will give background on OSI and esmini. Chapter 3 explains how to install and use our implementation, followed by chapter 4 which shows the properties/signals that we provide. Chapter 5 explains how we calculate the complex properties from the basic OSI information. The report ends with a conclusion.

2 Background

As mentioned above, traffic simulations typically rely on several interacting software components. This chapter provides an overview of different standards for communication between these components and introduces the simulator esmini.

2.1 OpenDRIVE

OpenDRIVE [2] is a standard for describing roads, their connection, and their geometry. Also objects like road markings, traffic signs, and traffic lights are represented by the specification. The file format, called "xodr", is based on XML. OpenDRIVE was developed to provide an input format for simulation environments. It does not contain vehicles or other traffic participants.

2.2 OpenSCENARIO

For the dynamic part of the environment, OpenSCENARIO [3] was developed. It describes vehicles and other traffic participates by specifying their trajectories or defining other controllers. The corresponding file format "xosc" is based on XML.

2.3 Open Simulation Interface

The Open Simulation Interface (OSI) [1] is a standardized interface for exchanging information about traffic situations. OSI does share some similarities with OpenDRIVE and OpenSCENARIO, as it can encode static information about road geometry etc. However, these details are typically only included when necessary. Most OSI messages rather focus on information about the current situation, e.g. the position of vehicles.

2.4 esmini

esmini [4] is a simple simulation program designed to work with OpenSCENARIO files. At its core, esmini is a library that can be used by other software, but it also provides a standalone scenario player with a graphical interface. While running a simulation, it can record OSI data in a file or send it to other software via UDP. If a scenario requires it, esmini can also receive commands via UDP, allowing other software to control the actions of certain vehicles.

3 The Python Package "osi_extractor"

The core of our IDP was the development of the python package osi_extractor. This chapter shows how to install and use osi_extractor. We further show how to use it in combination with esmini.

3.1 Installation and Setup

The package is available for download on the "Releases" page of our GitHub repository.[1] It is delivered as a wheel file with the file name extension ".whl" and can be installed using pip or other python package managers like conda.

```
pip install osi_extractor.whl
```

It is also possible to simply clone the git repository and build the package using pip:

```
pip install .
```

In both cases, all necessary python dependencies will be automatically installed. However, for reasons described in the next subsection 3.1.1, the protobuf compiler `protoc` must already be installed beforehand.

3.1.1 Dependencies

Our implementation uses `numpy` [5] to optimize certain calculations. This package is available via pip and can be installed automatically.

Additionally, `osi-extractor` depends on `open-simulation-interface` to process OSI messages. This package is only available on GitHub[2] and must be built locally. When installing `osi-extractor` as described above, the build process will be triggered automatically. However, this requires the protobuf compiler `protoc`[3] to already be installed on the system. Otherwise the build will fail.

3.2 Interaction with esmini

During a simulation, osi_extractor needs to receive simulation state updates from esmini and parse them. It should then produce its own representation of the current simulation state

[1] `https://github.com/cognibitgmbh/asam-osi-idp/releases`
[2] `https://github.com/OpenSimulationInterface/open-simulation-interface`
[3] `https://github.com/protocolbuffers/protobuf`

and make it accessible to the calling Python code. Finally, osi_extractor should also allow for sending messages back to esmini.

3.2.1 Receiving OSI Messages

The OSI specification describes how individual messages should be encoded using protobuf. However, there exists no general specification on how a stream of these messages should be sent over a network. esmini therefore introduces its own format for sending OSI messages to a preconfigured IP address and UDP port. While this format is rather simple, we did not find any documentation on it but had to reverse engineer it from network dumps.

Because OSI messages can be quite large, esmini may split each individual message into several UDP packets. A simple header is then added to the start of each packet, providing information on how to reconstruct the complete message. Hence, each UDP packet has the following format:

1. Counter: A 4 byte signed integer using two's complement, that specifies the position of the current payload. This is positive if the current payload is not the last piece of an OSI message. It is negative if it the current payload is the last piece of an OSI message. For each new OSI message, the counter starts again with 1.
2. Payload size: A 4 byte unsigned integer, that specifies the length of the payload in bytes.
3. Payload: The current OSI message piece.

Generally, esmini only sends OSI messages of type `GroundTruth`. While other message types might focus on the perspective of certain sensors, a `GroundTruth` message represents the ground truth of a simulated situation. This also means that positions and orientations are expressed in terms of a global coordinate system, not relative to any sensor or vehicle.

In principle, each `GroundTruth` message could contain all available information about the road network, stationary objects, traffic lights, moving objects and other details. However, this would result in unnecessarily large update messages because a significant part of this information does not change over the course of a simulation. esmini addresses this by only encoding most of this information into the first `GroundTruth` message. Following messages will only include information that has changed, e.g. the state of moving objects.

As a consequence, osi_extractor must be able to store static information for later use. Among other data, this includes information about the road network as well as traffic signs.

3.2.2 Sending Vehicle Updates to esmini

To simulate human driving behaviour it is not sufficient to simply receive data from esmini. There must also be some way to send messages back to esmini, informing it about the driver's decisions. Esmini allows two different modes for receving such vehicle updates:

asynchronous esmini simulates the scenario in real time and processes vehicle updates whenever they arrive.

synchronous Before each simulation step, esmini waits for a vehicle update. However, there is a timeout after which esmini will simulate the next step, even if no update arrived.

This mode can be configured directly in the OpenScenario definition of a vehicle entity. The following example is based on code from the esmini GitHub repository[4]:

```
<Entities>
    <ScenarioObject name="Car0">
      <Vehicle> <!-- details omitted for clarity --> </Vehicle>
      <ObjectController>
          <Controller name="UDPDriverController">
                <Properties>
                    <Property name="basePort" value="53995" />
                    <Property name="port" value="0" />
                    <Property name="execMode" value="synchronous" />
                </Properties>
            </Controller>
      </ObjectController>
  </ScenarioObject>
</Entities>
```

3.2.3 Types of Vehicle Updates

osi_extractor contains functionality for sending vehicle updates back to esmini via UDP. It supports three different types of update messages:

- `send_driver_input_update`: Overrides current value of steer, brake and throttle.
- `send_xyh_speed_steering_update`: Overrides current x, y, h, speed and steering wheel angle.
- `send_empty_update`: Does not override values, but triggers next step, if vehicle is synchronous.

The last type is only useful if Esmini runs synchronously and waits for an update to calculate the next time-step. The user guide of esmini[5] provides more information about the *UDPDriverController* in chapter 6.6.8.

3.3 Code Example

The following code snippet is intended to give an overview of how the osi_extractor package can be used:

[4] `https://github.com/esmini/esmini/tree/master/scripts/udp_driver`

[5] `https://github.com/esmini/esmini/blob/master/docs/user_guide.adoc#668-udpdrivercontroller`

```python
 1 from osi_extractor import SynchronOSI3Extractor
 2 from driver import DriverController
 3 
 4 RECEIVE_ADDRESS = "0.0.0.0"
 5 RECEIVE_PORT = 48198
 6 EGO_VEHICLE_ID = 0
 7 ESMINI_ADDRESS = "127.0.0.1"
 8 ESMINI_PORT = 53995
 9 
10 def main():
11     osi_extractor = SynchronOSI3Extractor(rec_ip_addr=RECEIVE_ADDRESS,
12                                           rec_port=RECEIVE_PORT,
13                                           ego_id=EGO_VEHICLE_ID,
14                                           esmini_ip_addr=ESMINI_ADDRESS,
15                                           esmini_port=ESMINI_PORT)
16     driver_controller = DriverController(EGO_VEHICLE_ID)
17 
18     with osi_extractor:
19         while True:
20             state = osi_extractor.get_next_state()
21             driver_update = driver_controller.process_next_state(state)
22             osi_extractor.send_driver_update(driver_update)
23 
24 if __name__ == "__main__":
25     main()
```

After importing the necessary classes, lines 4 through 8 introduce several constants to define IP addresses, ports and other parameters.

Inside the `main` function, line 12 then initializes a `SynchronOSI3Extractor` instance. This can be used to communicate with esmini using the sychronous mode. The object needs to be configured with IP addresses and ports as well as the vehicle ID of the "ego vehicle", i.e. the vehicle that should be controlled by this instance.

Line 17 then instantiates a `DriverController`, which is essentially a placeholder for driveBOT or any other code that processes simulation state and returns vehicle updates.

The `SynchronOSI3Extractor` class implements Python's notion of a *context manager*. Therefore, the `with`-statement in line 19 can be used to automatically manage the network sockets etc. used by osi_extractor.

The main simulation loop then contains three steps: First, it gets the next simulation state from the `SynchronOSI3Extractor` object. Because we are using synchronous mode, this method call will block until a new state is actually available. After that, the state object is passed to the driver controller, which processes it and returns a new vehicle update. Finally, this vehicle update is sent back to esmini.

4 Available Signals

For every moving object, we instantiate `MovingObjectState` and `RoadState`. Most signals that are related to a moving object are directly stored in the `MovingObjectState` object. Some of the related signals are stored in the `RoadState` object, which is referenced by an attribute in `MovingObjectState` called `road_state`.

4.1 MovingObjectState

attribute name	attribute type	description
simulator_id	int	unique id of the object in the simulator
object_type	int	type of the object according to OSI MovingObject.type
dimensions	Dimension3d	height, length, width of the object
location	Vector3d	x,y,z of the object
velocity	Vector3d	x,y,z velocity of the object
acceleration	Vector3d	x,y,z acceleration of the object
orientation	Orientation3d	roll, pitch, yaw of the object
lane_ids	list[int]	list of lane ids the object is on
road_id	int	id of the road the object is on
road_s	tuple[float, float]	current and total distance on the road
indicator_signal	int	state of the indicator signal
brake_light	int	state of the brake light
front_fog_light	int	state of the front fog light
rear_fog_light	int	state of the rear fog light
head_light	int	state of the head light
high_beam	int	state of the high beam
reversing_light	int	state of the reversing light
license_plate_illumination_rear	int	state of the license plate illumination rear
emergency_vehicle_illumination	int	state of the emergency vehicle illumination
service_vehicle_illumination	int	state of the service vehicle illumination

4.2 RoadState

attribute name	attribute type	description
curvature	float	curvature of the lane at the current position
curvature_change	float	first derivative of curvature
lane_width	float	width of current lane
lane_position	float	relative horizontal position of object in lane
distance_to_lane_end	float	distance to end of driving lane
distance_to_ramp	float	distance to the start of the next deceleration lane (highway exit)
distance_to_next_exit	Optional[float]	distance to end of deceleration lane. Only set if object on deceleration lane
lane_type	tuple[LaneType, LaneSubtype]	OSI LaneType and LanesSubtype
left_lane_marking	LaneBoundary MarkingType	left LaneBoundaryMarkingType (OSI)
right_lane_marking	LaneBoundary MarkingType	right LaneBoundaryMarkingType (OSI)
road_z	float	road elevation at objects position
road_angle	float	absolute road angle at objects position
relative_object_heading_angle	float	vehicle heading angle relative to road
road_on_highway	bool	is the road a highway?
road_on_junction	bool	is object on a junction?
same_road_as_ego	bool	is the object on the same road as the ego vehicle?
speed_limit	Optional[int]	speed limit on the current road
traffic_signs	list[RoadSignal]	list of all traffic signs on the current road

5 Implementation Details

5.1 Projecting Positions onto (OSI) Lanes

Most of the signals described in chapter 4 depend on a vehicle's relative position in its current lane. When driving along a road, for example, lane width and curvature can vary significantly over time. But while OSI does directly provide the absolute position of each vehicle, the representation of lane geometry is more complex: Each lane is associated with a sequence of straight line segments that approximate its center line [1]. A similar format is used to describe the left and right boundaries of a lane [1].

This presents a challenge regarding vehicle positions as they are unlikely to be exactly on the center line. In order to compute signals like the lane width, it is therefore necessary to project potentially abitrary positions onto a sequence of line segments. Ideally, this projection should of course minimize the distance between any actual position and its corresponding projected point.

5.1.1 Projecting onto a Straight Line

To address this problem, we first consider the most simple case: Projecting a position p onto a center line with only a single segment, i.e. a straight line. As illustrated in Figure 5.1, this can be solved by analyzing the vector v which connects p to its corresponding projected point q on the line: The angle between v and the center line is a right angle if and only if q minimizes the distance to p.

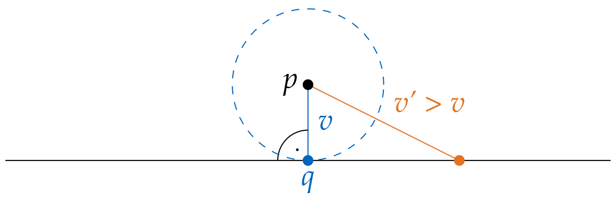

Figure 5.1: The shortest projection distance corresponds to a right angle

Finding q is therefore equivalent to solving an equation where the scalar product of v and the center line direction l is 0:

$$(p - q) \cdot l = v \cdot l = 0$$

Because OSI represents this center line as a line segment with a start point l_1 and an end point l_2, any point on this segment can be described as follows:

$$q = l_1 + t \cdot l = l_1 + t \cdot (l_2 - l_1)\ ,\ t \in [0, 1] \tag{5.1}$$

After combining these two equations, it is possible to solve for the value t, which should be a real number between 0 and 1. This value can be directly inserted into equation 5.1 to determine the actual projection point q . However, it also allows for a more intuitive interpretation: With l_1 as the start of the center line and l_2 its end, t relates the distance between q and the lane start to the total distance of the lane. For example, a value of $t = 0$ indicates that the projected point is exactly at the start of the lane and a value of $t = 0.6$ corresponds to a point 60% along the distance of this lane segment.

It is also possible that the value of t is not between 0 and 1. This indicates that the ideal projected point is not located between the start and end of a line segment. In this case, it typically makes more sense to project the vehicle position onto a different section of the road.

5.1.2 Choosing the Closest Line Segment

Naturally, most center lines in realistic simulation scenarios can not be described by simple straight lines. But for more complex roads, the above process can simply be applied to every single segment of the center line. This can be implemented efficiently by combining all start and end points into respective matrices which are then processed using numpy [5]. The result is a vector with one value t_i for each segment i of the centerline.

In the next step, it is necessary to choose the optimal line segement to minimize the projection distance. As explained above, if a value t is not between 0 and 1 it does not make much sense to project onto the corresponding segment because the car position is most likely on a different section of the road. Therefore, it is only necessary to consider line segments i where $0 <= t_i <= 1$.

In many cases, this condition will only be satisfied by one line segment and the projected position can immediately be calculated. If there is more than one such segment, one projection position can be calculated for each of them. The final result can then be determined by choosing the projection with minimum distance to the actual vehicle position.

However, in some cases it is possible that no t_i is between 0 and 1. As a consequence, the vehicle position can not be directly projected on any segment of the center line.

While this is a rather unlikely scenario which typically only lasts for a short time, it is still necessary to return some reasonable projected position. To address this issue, we decided to simply consider the start and end points of all center line segments and return the point which is closest to the vehicle position. This operation is also optimized by using numpy on the same matrix representation as above.

5.2 Curvature

The curvature κ of a curve is the inverse of the radius, that is: $\kappa = \frac{1}{r}$. In the context of driving, the curvature of the road is important, because the driver must turn the steering wheel according to it. OSI only provides a list of points to describe the street. Hence we need to calculate an approximation of the curvature based on these points. To calculate a meaningful estimate we consider how road trajectories are typically created.

5.2.1 Clothoids

Bachmann [6] explains why clothoids are used in road construction to define the transition between different road segments. The defining property of clothoids is that the curvature changes proportional to the arc length. In the context of driving, this helps to avoid sudden steering wheel movements. However OSI won't deliver clothoids, but only a list of points. One solution would be to interpolate the clothoid parameters based on the points. We are only aware of one python-library that does clothoid-interpolation. The python-library is named "pyclothoids" and is based on Bertolazzi et al. [7, 8]. However the function `SolveG2` that pyclothoids provides does not fit our needs. `SolveG2` does not interpolate a list of points, but only two points. It also requires the curvature and tangent of both points.

5.2.2 Piecewise Linear Curvature

To avoid clothoid-interpolation, we came up with a simpler solution. The solution consists of 2 steps:

1. Calculate the curvature κ_i for each OSI point P_i, based on the neighboring points
2. Weighted sum of the two curvatures of the points that are closest to the position of interest.

1. Curvature of an OSI Point

We take the predecessor P_{i-1} and successor P_{i+1} of OSI point P_i. The z-axis is neglected for the following. If the three points are in a straight line, we assume that the circle-radius is $r_i = \infty$, hence we set the curvature to $\kappa_i = 0 \approx \frac{1}{\infty}$. If that is not the case, we imagine a circle that touches P_{i-1}, P_i and P_{i+1}. Since the three points are not in a straight line, this circle exists and is unambiguous. We assume now, that the curvature κ_i at P_i is the curvature of the circle. To calculate the curvature of this circle, we need to first calculate the area A_i of the triangle, that spans between the points. We use Heron's formula [9] which can be rearranged to

$$A_i = \frac{1}{4}\sqrt{4a_i^2 b_i^2 - (a_i^2 + b_i^2 - c_i^2)^2}$$

where a_i, b_i and c_i are the edge lengths of the triangle.

With the area A_i and the edge lengths a_i, b_i and c_i, we can calculate the curvature κ_i using the Menger curvature [10]:

$$\kappa_i = \frac{4A_i}{a_i b_i c_i}$$

We only need the direct neighbors of each point for calculating the curvature. This enables us to calculate the curvatures for each lane in one pass using numpy. For the beginning and end of a lane, however, we can not directly calculate the curvature with this method as they only have one neighbor. We therefore simply assume that the curvature at these points is zero, i.e the road is straight at its start and end.

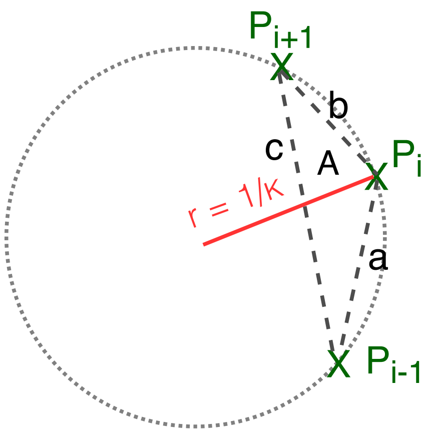

Figure 5.2: Curvature calculation using Haron and Menger

2. Weighted Sum of Two Curvatures

Our approach in section 5.1 projects the vehicle onto the line segment between two OSI points. The approach emits the value t which represents the progress between the start (P_i) and end (P_{i+1}) of this segment. We simply use this t_i, as well as curvatures κ_i and κ_{i+1} to calculate a weighted sum:

$$\kappa = \kappa_i \cdot (1 - t) + \kappa_{i+1} \cdot t$$

This linear curvature interpolation is motivated by the linear curvature change of clothoids.

5.3 Lane Graph

In some cases it is necessary to consider an entire highway lane. Example signals where this is relevant are the distance to the end of a lane or the beginning of an offramp. While OSI does describe roads in terms of "lanes", these are not directly suitable for calculating such signals: A single continuous *driving lane* can be split into several *OSI lanes*. Therefore it is not sufficient to simply consider OSI lanes by themselves. Often, we also need to consider their successors and predecessors. Additionally, it can also be helpful to consider neighboring lanes. For example, when determining the distance to the next highway exit, the actual exit lane will not be one of the regular driving lanes but it should neighbor at least one of them.

Again, OSI does already provide ways to encode information about an OSI lanes's neighbors, predecessors and successors. However, when working with OSI data generated by esmini, we found this information to be rather unreliable. In particular, the successor and predecessor relations between OSI lanes were quite inconsistent. Sometimes, there even where situations where lane 2 was listed as a successor of lane 1 but lane 1 was not encoded as a predecessor of lane 2.

To address this, we decided to create our own representation of a *lane graph*: Each *lane graph node* corresponds to a particular OSI lane and uses lists to store its left and right neighbors as well as its predecessors and successors. To determine the neighbors of a given OSI lane, we simply rely on the data provided by OSI. But for the reasons described above, we use a different method to determine predecessors and successors.

The underlying idea is very simple: If an OSI lane succeeds another one, then their respective start and end points should be very close to each other. We therefore simply consider all possible pairings of OSI lanes and compute the distances between their start and end points. Should this distance is below a certain small threshold, we can determine their successor relationship. If e.g. the end point of lane 1 is very close to the start point of lane 2, then lane 1 is considered the predecessor of lane 2. Conversely, lane 2 must then also be the successor of lane 1. To prevent inconsistent results, we also included checks that raise an exception if a lane could have more than one predecessor or successor.

Because this computation has to iterate over all possible pairings of OSI lanes, it is not very efficient and should not be executed too frequently. However, it is sufficient to only run this computation when new information about lane information is received via OSI. In typical use cases this only happens once at the start of the simulation, minimizing the performance impact of the lane graph computation.

5.4 Roads

For some signals we need to define roads. One example is *road_s*: If one lane of a three-lane highway ends, *road_s* should not depend on the length of the ending lane but on the length of the entire road. The signals *road_id, same_road_as_ego, traffic_signs* and *speed_limit* also require a definition of roads.

5.4.1 OpenDrive Roads

OpenDRIVE roads are a sequence of one or more geometric shapes that define the road course as explained in [2, Chapter 7&8]. Nevertheless for the following two reasons we cannot use OpenDRIVE roads:

- According to [2, Chapter 8] every road is described by one or more OpenDRIVE road elements. Hence, it is possible for an OpenDRIVE road to end, even though the highway continues.
- Esmini doesn't send OpenDRIVE data but OSI data. OSI does not contain a definition for roads. Even if a natural road is described by exactly one OpenDRIVE road, we lose this information.

As a consequence we establish road definitions based on lane graph, lane type and lane subtype. Since the scope of this IDP is limited to highway scenarios, our approach does not target rural roads.

5.4.2 Overview: Turn Lane Graph to Roads

The algorithm to assign lanes to roads can be broken down into the following steps:

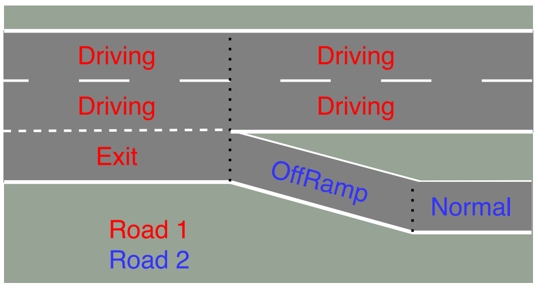

Figure 5.3: Highway exit with assigned roads

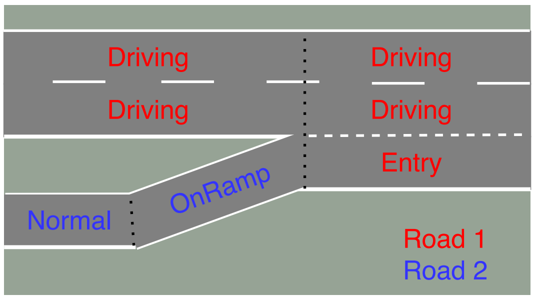

Figure 5.4: Highway entry with assigned roads

- Infer if two consecutive lanes belong to same road. (Used by the other two steps)
- Identify the beginning lane of each road
- Enlarge roads by transitively adding successor and neighbor lanes

The following subsections explain these 3 steps.

5.4.3 Do Consecutive Lanes Belong to Same Road?

For the other two steps it is important to decide if two consecutive lanes belong to the same road. In case they don't, we draw a conceptual border between these lanes. As a result we don't treat them as consecutive lanes anymore. In the context of highways, this happens on highway exits, since there are lane successors which are not part of the highway any more. To detect highway exits we look at how they are built in OpenDRIVE (see [2, Chapter 9.5.3.]):

- A lane of type *Exit* is a parallel neighbor of the normal *Driving* lanes.
- The *Exit* lane is followed by either *OffRamp* or *ConnectingRamp*.
- *OffRamp*/*ConnectingRamp* are not neighbours of any highway lane.

We consider the type change from *Exit* to *OffRamp*/*ConnectingRamp* as a border between roads. Since highway entries are created conversely, we also treat the type change from *OnRamp*/*ConnectingRamp* to *Entry* as a border between roads. To visualize this we created Figure 5.3 and 5.4, which contain the OpenDRIVE types and our road-assignment.

5.4.4 Identify Beginning Lanes

With *beginning lane* we describe the first and rightmost lane of a road. It can be seen as a representative for this road. This lane is the starting point, from which our algorithm adds the neighbouring and succeeding lanes. We identify road beginnings by searching for lanes in our lane graph that meet the following criteria:

- The lane is not part of an earlier detected road.

- If there exist a right neighbour lane, this neighbour is already part of a earlier detected road.
- For the lane and all its left neighbours holds:
  - They have no predecessor lane, or the predecessor is already part of another road, or the predecessor is not considered to be part of the same road according to the criterion in 5.4.3

5.4.5 Add Lanes to Roads

Since we now have a way to find beginning lanes, we can create all roads as follows.

1. Iterate through lanes in the lane graph until a beginning lane *L* is found.
2. Create new road starting with *L*
   a) Add all left neighbours of *L* to new road if they are not already part of a another road.
   b) Find next rightmost lane among the successor lanes of the lanes added in a).
      - only consider lanes that are not already part of a road
      - only consider lanes that are of the same road as *L* according to the criterion in 5.4.3
   c) If the rightmost lane was identified, add it to the road. Set *L* to this new rightmost lane and repeat from a). If no new rightmost lane was identified, the creation of the new road is finished.
3. repeat from 1. until no new road gets identified.

After this algorithm terminates, all lanes from the lane graph should be assigned to a road. One consequence of this approach is that opposing highway lanes get assigned to individual roads. If a highway splits up into two highways, the left arm will be assigned to a new road. The right arm will have the same road assigned as the highway before the split. If two highways merge, the road that is first created will continue after the merge, while the later created road will end at the merge. We need such heuristics, since OSI does not provide specific road information. Since we tell highway splits and highway exits apart by looking at the lane types, it is crucial that those types are set properly.

5.5 Traffic Signs and Speed Limits

Certain signals listed in chapter 4 are related to traffic signs. These include the current speed limit as well as `traffic_signs`, which provides a list of signs for the current road, including potential warning signs etc.

5.5.1 Assigning Signs to Roads

With the OSI representation of traffic signs, it is not immediately possible to decide which signs belong to which road or lane. Our approach for assigning signs to roads uses two criteria:

- the distance between an OSI lane's center line and a sign
- the angle between driving direction and a sign's orientation

We use this information to assign each sign to one specific OSI lane. To find the most suitable lane, we first analyze the orientation of the sign. If the angle with respect to driving direction is greater than 45 degrees, the respective OSI lane will not be considered any further. After that, we focus on the distance between center line and sign. All OSI lanes where this distance is above a certain threshold are immediately excluded. (Currently this threshold is set to 10 meters.) From the remaining OSI lanes, we simply choose the one with the smallest distance to the sign. We then add this sign to the corresponding road, using the road mapping from section 5.4.

Because every OSI lane needs to be considered for every traffic sign, this assignment process is rather time consuming. Similar to other complex computations, it is therefore only executed when OSI lane information changes.

5.5.2 Current Speed Limit

After mapping signs to roads, we can use this information to determine the speed limit at a position. We implemented this by iterating over all road signs. Any sign which appears later than the current vehicle position is skipped. The speed limit sign which is closest to the vehicle determines the speed limit. If the vehicle is not on an exit lane, all signs are ignored that stand next to exit lanes. Otherwise, if the vehicle is on an exit lane, we include speed limits that are next to this exit. If there is no applicable speed limit sign, `speed_limit` is set to None. Similarly, if the last applicable sign marks the end of a speed limit, `speed_limit` is also set to None.

6 Conclusion

During this project we encountered several challenges. Some of them where questions that we had to answer, like "What is a road" or "How do we know, if a speed sign belongs to a road". Another challenge was the lack of documentation of protocols. We needed to reverse engineer how esmini wraps OSI when it's sent via UDP. Also OSI is sometimes not specific on how certain information is formated. The same applies to OpenScenario. In these cases we relied on examples.

OsiExtractor was developed specifically for highways. Accordingly, it can only be used in these cases. The behavior in urban road networks with, for example, intersections is not defined. To distinguish between highway junctions and exits, OsiExtractor uses lane types. It is important to pay attention to these lane types, because in some of the test scenarios we received they were set incorrectly.

With our implementation of *OsiExtractor* we have illustrated how complex signals can be inferred from simple OSI data. These complex signals represent the road environment from the point of view of the individual vehicles. This is a necessary step to enable simulators like *driveBOT* to work with OSI.

An important feature of our implementation is throughput. For the scenarios we tested, OsiExtractor can process OSI streams from esmini on a current consumer PC without creating a backlog. This works since we used vectorized calculations using numpy and precalculate some data.

Overall OsiExtractor is not only an adapter for OSI generated by esmini, but can be used in the future as a starting point for connecting other OSI simulators to driveBOT.

List of Figures

Bibliography

[1] Open Simulation Interface Contributors. *ASAM OSI v3.5.0 Reference Documentation*. Version 3.5.0. July 2022. URL: https://www.asam.net/static_downloads/ASAM_OSI_reference-documentation_v3.5.0 (visited on 03/12/2023).

[2] ASAM. *OpenDRIVE*. Version 1.6. 2020. URL: https://releases.asam.net/OpenDRIVE/1.6.0/ASAM_OpenDRIVE_BS_V1-6-0.html (visited on 03/15/2023).

[3] ASAM. *OpenSCENARIO*. Version 1.2.0. 2022. URL: https://www.asam.net/standards/detail/openscenario/ (visited on 05/12/2023).

[4] esmini Contributors. *Environment Simulator Minimalistic (esmini)*. URL: https://github.com/esmini/esmini (visited on 05/23/2023).

[5] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[6] E. Bachmann. "Die Klothoide als Übergangskurve im Straßenbau". In: *Schweizerische Zeitschrift für Vermessung, Kulturtechnik* 49.6 (1951).

[7] E. Bertolazzi and M. Frego. "On the G2 Hermite Interpolation Problem with clothoids". In: *Journal of Computational and Applied Mathematics* 341 (2018), pp. 99–116. ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2018.03.029. URL: https://www.sciencedirect.com/science/article/pii/S0377042718301924.

[8] E. Bertolazzi and M. Frego. "G1 fitting with clothoids". In: *Mathematical Methods in the Applied Sciences* 38.5 (2015), pp. 881–897. DOI: https://doi.org/10.1002/mma.3114. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/mma.3114.

[9] W. Dunham. *Journey through Genius: Great Theorems of Mathematics*. John Wiley & Sons, 1991, p. 119.

[10] P. Mattila, M. S. Melnikov, and J. Verdera. "The Cauchy Integral, Analytic Capacity, and Uniform Rectifiability". In: *Annals of Mathematics* 144 (1996), p. 129. URL: http://www.jstor.org/stable/2118585.