SPOT Manual

(Version 2017a)

Markus Koschi and Matthias Althoff

Technische Universität München, 85748 Garching, Germany

Abstract

This manual describes the architecture and methods of our MATLAB tool *SPOT*, which predicts the occupancies of traffic participants in a set-based fashion. Predicting the set of all possible behaviors is required to ensure safe motion plans of an automated vehicle. *SPOT* is a MATLAB tool to efficiently compute the future occupancy of other traffic participants based on reachability analysis. We consider physical constraints and assume that the traffic participants abide by the traffic rules. However, we remove assumptions for each traffic participant individually as soon a violation is detected.

Contents

1 Philosophy and Architecture

We introduce our tool Set-Based Prediction Of Traffic Participants (*SPOT*), which is available at spot.in.tum.de. Please note that *SPOT* is prototypical and this manual is still a rough documentation, which will be continiously improved. This manual focuses on presenting the

capabilities of *SPOT* without providing the mathematical background. For details and proofs of our over-approximative occupancy prediction based on reachability analysis, please see [1–3].

SPOT provides the following key features:

- The tool is open source so that parts of the code can be used for one's own purposes.
- Uncertainties in measurements (position, orientation, dimensions, velocity, and acceleration) and the future behavior of traffic participants are explicitly considered.
- Even though *SPOT* is implemented in MATLAB (generally slower than compiled code), we achieve computation times within a fraction of the predicted horizon (typically less than 1.0 %).
- *SPOT* only has a few lines of code (less than 5000) to easily understand its working and extend its features.
- *SPOT* is designed to be fed by different types of input. As a stand-alone tool, the user defines the time horizon for the prediction and a traffic scenario in an XML format. One can either download traffic scenes from our website or create new ones (see our XML specification for details). It is also possible to embed *SPOT* in a motion planner and feed it with environment data.

The architecture of *SPOT* is presented in Fig. 1 using the class diagram of UML[1]. Different packages[2] contain several classes as visualized with colored background. We emulate the perception of the ego vehicle (class `Perception`, package `globalPck`, Section 5) to consider sensor range limitations among others. Our model of the environment is stored as a map (class `Map`) and contains lanes (class `Lane`), obstacles (class `Obstacle`), and the ego vehicle itself (class `Vehicle`). A hierarchical class structure behind the superclass `Obstacle` allows us to distinguish between static and dynamic obstacles (class `StaticObstacle` and `DynamicObstacle`) and to represent different types of traffic participants, like passenger cars, trucks, and bicycles[3]. All these classes are part of the package `world` (Section 6). For predicting the occupancy of an obstacle, each object of the class `Obstacle` holds its occupancy (class `Occupancy`, package `prediction`, Section 7) as a property. Please note that if you prefer to employ your own or another environmental model, you can extract the methods from the class `Occupancy` to benefit from our implementation.

2 Installation

The software does not require any installation, except that the path for *SPOT* has to be set as the current folder in MATLAB. To this end, the *Mapping Toolbox* of MATLAB is needed for polygon computations, i.e. intersection and difference of two polygons. *SPOT* checks whether the Mapping Toolbox is installed and throws an error if it cannot find a license.

3 Definition of Traffic Scenes

As a stand-alone tool, only two inputs are required for the occupancy prediction. The user must define (1) the time of the prediction horizon, i.e. the time t_s (start) to t_f (final) and the step size dt, which defines the length Δt of the consecutive time intervals $\tau_k = [t_k, t_{k+1}]$, and (2)

[1] uml.org

[2] mathworks.com/help/matlab/matlab_oop/organizing-classes-in-folders.html

[3] So far, we only do occupancy prediction for road vehicles.

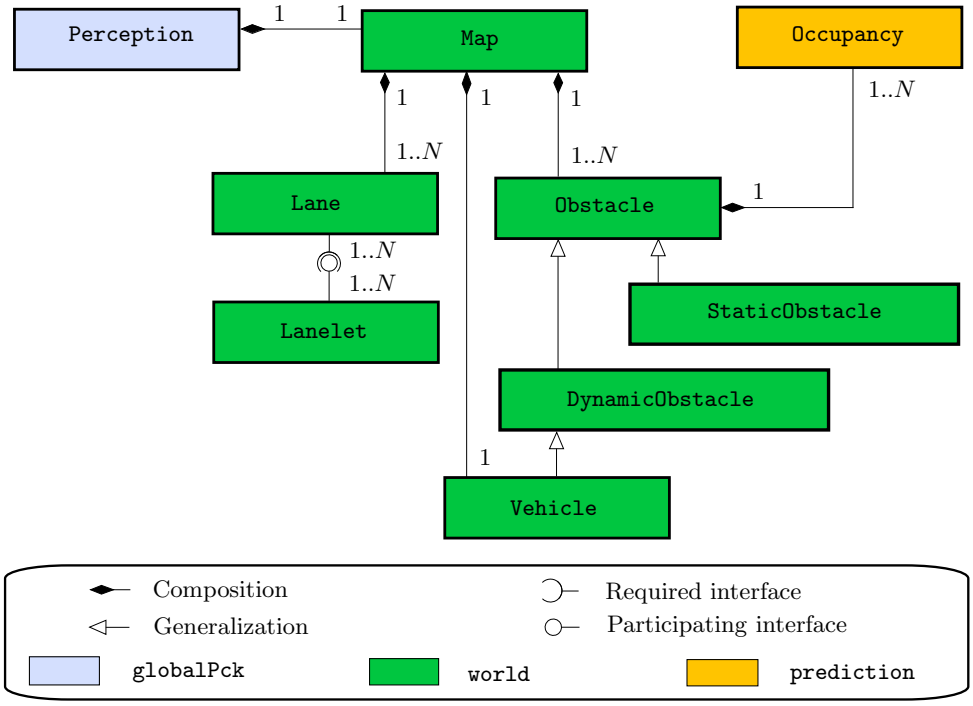

Figure 1: Unified Modeling Language (UML) class diagram of *SPOT*.

the traffic scene as an XML file (see the XML Documentation on our website for details on the traffic scene model). In *SPOT*, a set of examples of traffic scenes are included (see the folder scenarios). Further scenarios can be downloaded from the CommonRoad benchmark repository[4]. Beyond that, we encourage you to create new traffic scenes by importing an open-source map (e.g. *OpenStreetMap*[5]) into the free editor *JavaOpenStreetMap*[6] (JOSM). *SPOT* can also parse the osm files created by JOSM.

4 Package `geometry`

The package `geometry` contains utility functions required for geometric computations and are not explained in detail here.

5 Package `globalPck`

The package `globalPck` contains all global required classes for the occupancy prediction.

- `Dynamic` – Abstract class for dynamic objects.
- `Perception` – emulates the perception of the ego vehicle.
- `TimeInterval` – describes time intervals for prediction horizons.
- `Trajectory` – contains the discrete states of other dynamic traffic participants.
- `PlotProperties` – globally defines properties for plotting the occupancies. Here, the user can specify its requirements for plotting objects.

[4] commonroad.in.tum.de
[5] openstreetmap.org
[6] josm.openstreetmap.de

6 Package World

The structure of the package world is shown in Fig. 1. It contains several classes, which are displayed in their hierarchy in Fig. 2. Further dynamic obstacles, e.g. pedestrians and bicycles, will be added later.

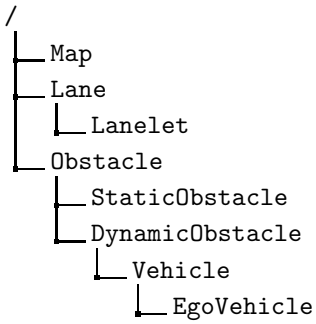

Figure 2: Structure of the package world

For details of our models of traffic participants and their constraints, please see our publications.

The constraints for the traffic participants, e.g. $v_{\max}$ and v_S, are defined in the corresponding classes DynamicObstacle and Vehicle. To adapt the default values during runtime, use the method setset(obj, propertyName, propertyValue).

7 Package Prediction

The package Prediction is structured in the following functions:

- computeOccupancyCore – invokes all relevant prediction functions to compute the future occupancy of one obstacle.
- manageConstraints – adapts the prediction parameter as soon as a constraint has been violated.
- findAllReachableLanes – searches the road graph for the reachable lanes of the current obstacle.
- M1_accelerationBasedOccupancy – computes $\mathcal{O}_1$, see Sec. 7.1.
- M2_laneFollowingOccupancy – computes $\mathcal{O}_2$, see Sec. 7.2.
- M3_safeDistance – computes $\mathcal{O}_3$, see Sec. 7.3.

The class Occupancy is used to compute and describe the occupancy polygons. By setting the Boolean parameters COMPUTE_OCC_M1 and COMPUTE_OCC_M2 to true/false, one can individually activate/deactivate the abstractions M_i, which are explained next.

7.1 M1_accelerationBasedOccupancy

The method accelerationOccupancyLocal over-approximates the occupancy of a point mass with a convex polygon $Q(q_1, \ldots, q_6)$ as shown in Fig. 3.

Next, the dimension of the traffic participant is added to the occupancy of our point mass (method addObjectDimensions). We enclose its shape by a rectangle of length l and width

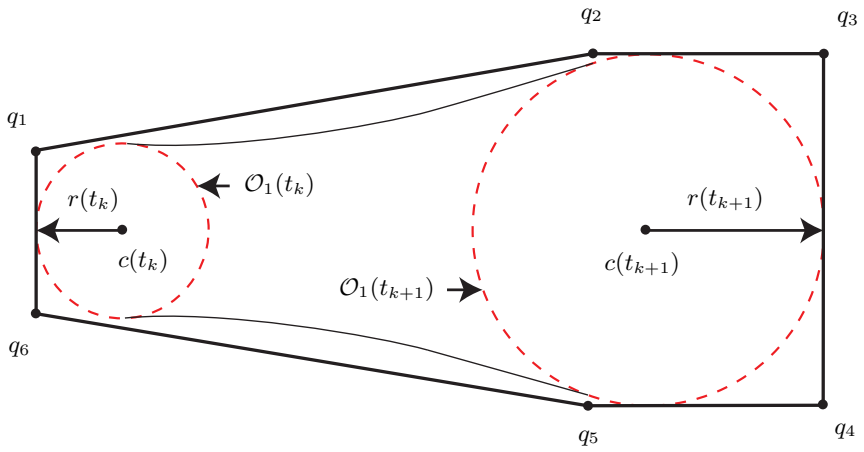

Figure 3: Polygon $Q(q_1, \ldots, q_6)$ encloses the occupancy of a point mass for the time interval $\tau_k = [t_k, t_{k+1}]$.

w, which includes the obstacle dimension ($\tilde{l}$ and $\tilde{w}$) and measurement uncertainties. Fig. 4 illustrates how polygon Q is enlarged by half of l and w in each direction to obtain polygon P, which is an over-approximation for the occupancy $\mathcal{O}_1(\tau_k)$. Finally, the occupancy $\mathcal{O}_1(\tau_k)$ is obtained by rotating and translating polygon P according to the initial position and orientation (method `rotateAndTranslateVertices`).

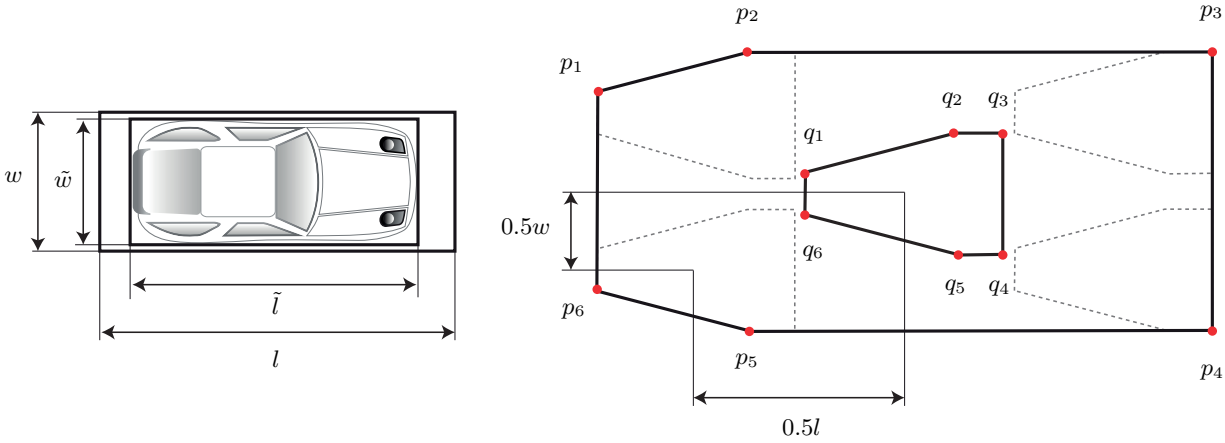

Figure 4: Polygon $P(p_1, \ldots, p_6)$ is the occupancy $\mathcal{O}_1(\tau_k)$ of an obstacle, here a passenger car. The vertices q_1-q_6 are taken from Fig. 3.

7.2 M2_laneFollowingOccupancy

The computation for abstraction M_2 is divided into the problem of the shortest path of a lane (computed prior to the prediction, see Sec. 7.2.1), finding the minimum and maximum position along this path (see Sec. 7.2.1), and constructing the occupancy polygon (see Sec. 7.2.3).

7.2.1 Shortest Path through a Lane

Since lateral acceleration is not restricted in abstraction M_2, the traveled distance along a path is irrespective of its shape and it suffices to consider a 1D path, i.e. a straight line, which represents a path through the current lane. In order to over-approximate the occupancy, we must choose the shortest path. As exactly solving this optimization problem of finding the shortest path through a lane is too time consuming, we use an abstraction for the shortest path.

When always following the inner bound of the lane, while jumping across to the other lane border at an inflection point, i.e. a border point where the sign of the well-known signed curvature changes, we obtain an under-approximation for the shortest path through this lane. A proof is provided in [2], where this method is described as *inflection-point segmentation*. We denote the path variable of the shortest path by ξ, which is obtained by the method `findShortestPath` prior to the prediction algorithm.

7.2.2 Closest and Furthest Position along a Path

Along this path, we compute the minimum and maximum position which can be reached by the traffic participant.

When applying full deceleration, the traveled distance in a time interval $\tau_k = [t_k, t_{k+1}]$ is

$$\xi_{\text{closest}}(\tau_k) = \min(\xi_{\text{closest}}(t_k), \xi_{\text{closest}}(t_{k+1})) \tag{1}$$

in which $\xi_{\text{closest}}(t) = -\frac{1}{2}a_{\max}t^2 + v_0 t$ is the well-known solution for constant acceleration. If driving backwards is not allowed ($C3$), t is limited by $\frac{v_0}{a_{\max}}$, which is the time needed to stop. (See method `closestLongitudinalReach`.) Similarly, full acceleration is applied to find the furthest position along ξ. Due to the different acceleration limits, the traveled distance in time t depends on the velocity v and is

$$\xi_{\text{furthest}}(t) = \begin{cases} \frac{1}{2}a_{\max}t^2 + v_0 t, & v \le v_S \wedge v < v_{\max}, \\ \frac{(2a_{\max}v_S t + v_0^2)^{\frac{3}{2}} - v_0^3}{3a_{\max}v_S}, & v_S < v < v_{\max}, \\ v_{\max}t, & v \ge v_{\max}. \end{cases}$$

Since the longitudinal dynamics are monotonic, the furthest position in a time interval τ_k is

$$\xi_{\text{furthest}}(\tau_k) = \xi_{\text{furthest}}(t_{k+1}). \tag{2}$$

(See method `furthestLongitudinalReach`.)

7.2.3 Start and End Bound of the Occupancy $\mathcal{O}_2$

To obtain a set which over-approximates the occupancy $\mathcal{O}_2(\tau_k)$, the closest and furthest position along ξ are mapped back to the lane in the global coordinate frame as shown in Fig. 5. As ξ_{closest} and ξ_{furthest} are defined on the shortest path along the inner border of the curent lane, they yield the support points μ_{start} and μ_{end}. Then, we construct the start bound h_{start} and end bound h_{end} such that they are perpendicular to the inner lane bound. Finally, the occupancy polygon for the current lane is defined by the vertices on the left and right lane border from the start bound h_{start} until the end bound h_{end}. See [2] for a detailed mathematical description and the method `constructBound` for our implementation.

Occupancy $\mathcal{O}_2$ is not only computed for all current lanes, but for all reachable lanes $\mathcal{O}_{\text{road}}$. The occupancy in the left adjacent lane is exemplarily drawn in Fig. 5. By constructing a perpendicular line at the shared border point of the end bound σ_{end} to the other lane border, we obtain the end bound $\tilde{h}_{\text{end}}$ in the adjacent lane. The start bound $\tilde{h}_{\text{start}}$ is constructed analogously based on the shared point μ_{start}. Thus, the occupancy polygon for the reachable adjacent lanes can be obtained as for the current lane.

7.3 M3_safeDistance

Please note that this method will be released shortly.

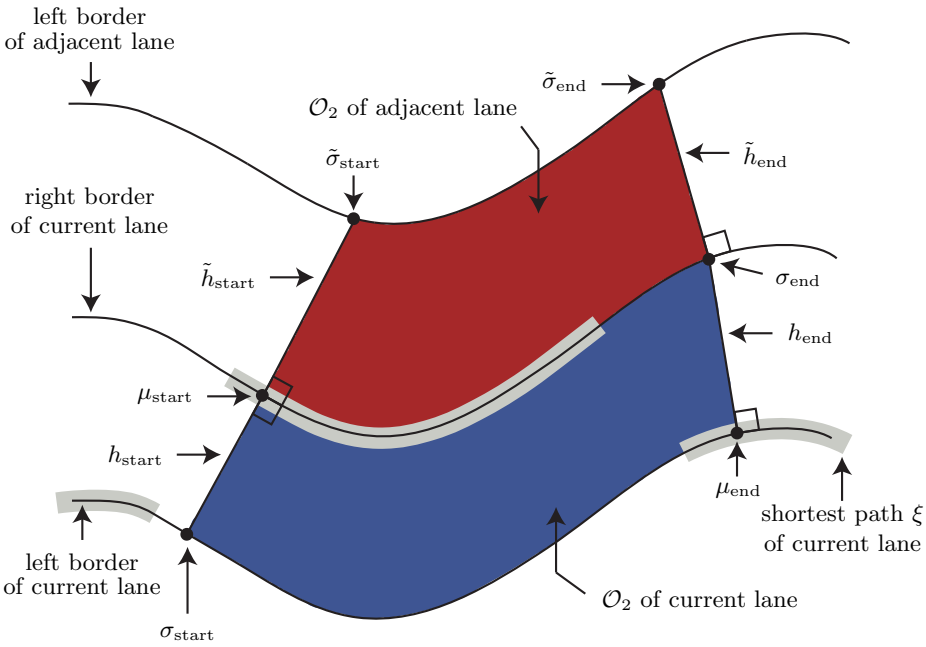

Figure 5: Constructing h_{start} and h_{end} to obtain the occupancy $\mathcal{O}_2$ of an obstacle.

8 Example

We demonstrate the workflow of *SPOT* by giving a short example.

First, we define the inputs and create a perception object:

```
% inputFile = someFile.ext;
inputFile = 'scenarios/GER_Muc_3a.xml'

% time interval in seconds for prediction of the occupancy
ts_prediction = 0;
dt_prediction = 0.1;
tf_prediction = 3.0;
timeInterval_prediction = globalPck.TimeInterval(ts_prediction,
   dt_prediction, tf_prediction);

% create perception from input (holding a map with all lanes,
% adjacency graph and all obstacles)
perception = globalPck.Perception(inputFile);
```

Then, we run the prediction and plot the perception object including the occupancies.

```
% do occupancy calculation
perception.computeOccupancyGlobal(timeInterval_prediction);

% plot
perception.plot(timeInterval_prediction)
```

SPOT computes the future occupancy of all three obstacles independently. They are saved in a a $n \times m$ matrix for n reachable lanes and m time intervals. You can obtain the coordinates of the polygon points for obstacle i from the workspace variable `perception.map.obstacles(i).occupancy(n,m).vertices`.

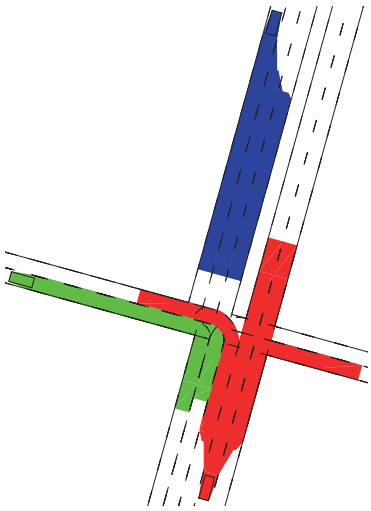

Figure 6: *SPOT* Output: Occupancies for $t \in [t_s, t_f]$.

The occupancies for the whole prediction horizon are plotted in a figure as shown in Fig. 6.

Please see the provided method `main` for the basic functionality of *SPOT*. One can also specify a time interval for the scenario and run the prediction for several initial states. In addition, one can add a trajectory for the ego vehicle, compute the occupancy of the ego vehicle along its trajectory and then try to verify the trajectory by a collision check with the occupancies of the surrounding traffic participants.

9 Disclaimer

Please note that *SPOT* is primarily for research. We do not guarantee that the code is bug-free.

One needs expert knowledge to obtain optimal results. This tool is prototypical and not all functions that exist in the software package are explained.

If you have questions or suggestions, please contact us through www6.in.tum.de.

10 Conclusions

We present *SPOT*, the first tool for set-based prediction of other traffic participants, which is available as open source software at spot.in.tum.de and can be easily adapted to one's own needs. Based on reachability analysis, we compute the set of future occupancies of each surrounding traffic participant for arbitrary road networks. These traffic scenes can be specified in XML files, which are based on the CommonRoad format. Several applications can benefit from our tool, where most importantly, *SPOT* can be used to verify intended trajectories, since our approach is inherently safe. We have introduced six constraints to obtain a tight over-approximation, but remove them individually as soon as one is violated, which results in larger occupancies and thus a smaller drivable area for the ego vehicle.

Acknowledgment

The authors gratefully acknowledge financial support by the BMW Group within the CAR@TUM Project.

References

[1] M. Koschi and M. Althoff, "SPOT: A tool for set-based prediction of traffic participants," in *Proc. of the IEEE Intelligent Vehicles Symposium*, 2017, [to appear].

[2] M. Althoff and S. Magdici, "Set-based prediction of traffic participants on arbitrary road networks," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 2, pp. 187–202, 2016.

[3] M. Althoff and J. M. Dolan, "Online verification of automated road vehicles using reachability analysis," *IEEE Transactions on Robotics*, vol. 30, no. 4, pp. 903–918, 2014.