Rheinisch-Westfälische Technische Hochschule Aachen

Faculty Computer Science

Autonomous Driving Lab - Documentation

Winter Term 2016/17

Topic: Autonomous Driving Lab –
Documentation Controller

submitted by: Stefan Erlbeck - Matr. Nr. 344628
Xiangwei Wang - Matr. Nr. 353370
Christoph Grüne - Matr.-Nr. 349070

submitted on: 20.03.2017

Advisor: Evgeny Kusmenko

Contents

1 Introduction

An autonomous car is a vehicle that is capable of sensing its environment and navigating without human input.It uses a variety of techniques, such as radar, laser light, GPS and a series of computer vision technologies to detect their surroundings and collect the information. Controller system analyses and interpret those information to identify appropriate navigation paths, deal with emergencies and in the end drive to the desired destination safely.
Our Controller consists of five main components. DataBus, which provides a mechanism for information exchange between different groups. MainControlBlock, which initialize the main sub components, input all needed data to its sub components and output steering, engine, brake, gear values for actuators. InnerControlBlock, which consists of SteeringLogic and VelocityLogic, is the central processing unit of Controller. Besides, NavigationBlock takes a map, current position and destination as input and calculate a path for the car. TrajectoryBlock calculates the velocity we can drive by foreseeing the trajectory. Both those two components provide the auxiliary information for the SteeringLogic and VelocityLogic. In the end, we have PID controllers for steering angle and velocity, which continuously calculate an error value as the difference between a desired setpoint and a measured value and applies a correction based on the proportional, integral, and derivative terms.

2 Function Blocks

2.1 Navigation Block

Inputs:

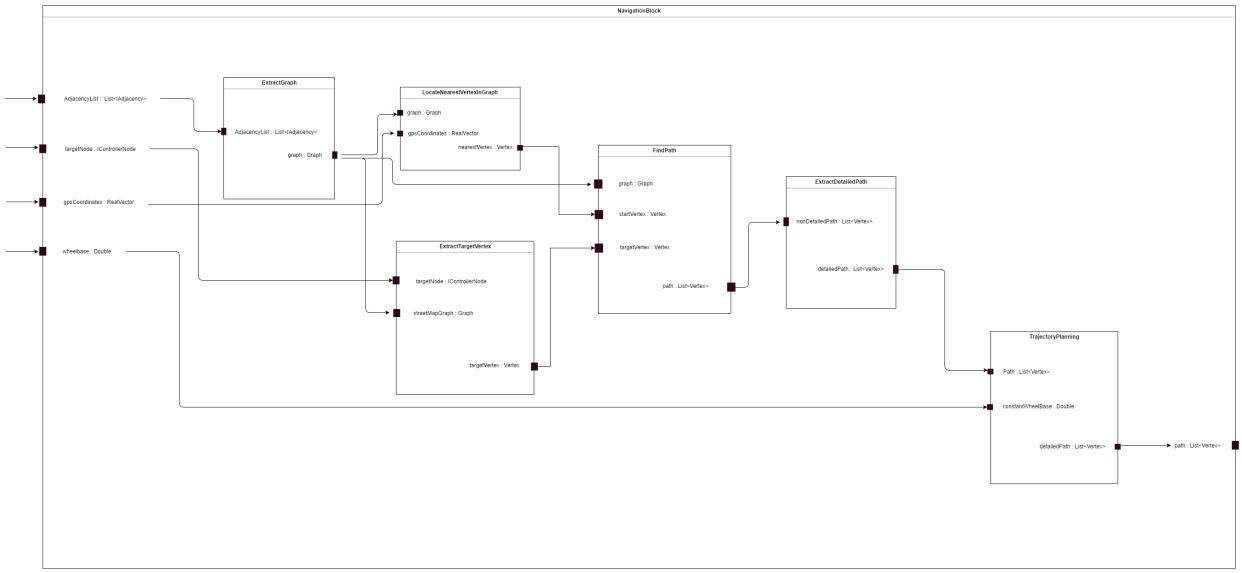
- **AdjacencyList** The street map. It is a list of adjacent vertex pairs provided by environment, each IAdjacency has two adjacent vertices.
- **TargetNode** The Node the car should navigate to.
- **gpsCoordinates** The GPS coordinates of the current position of the car.
- **wheelbase** The wheelbase of the car.

Outputs:

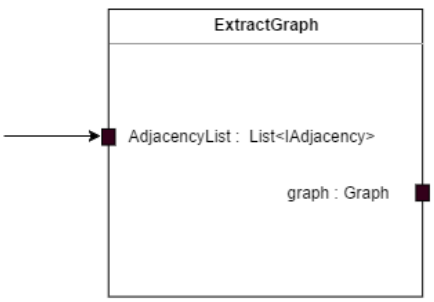
- **path** The shortest path to TargetNode with the calculated maximal steering angles.

Purpose. The NavigationBlock extracts the shortest path to the point the car should navigate to out of the street map and calculates for every vertex the maximal steering angle the car drives, so that the MainControlBlock can use this data to calculate the maximal velocity in every vertex.

Implementation. The shortest path is calculated by a modified version of Dijkstra's algorithm. Furthermore, the maximal steering angle is calculate with the algorithm that is specified in Trajectory Planning Block.
The overall structure of the NavigationBlock is as follows:



2.1.1 ExtractGraph



Inputs:

- **AdjacencyList** List of adjacent vertex pairs provided by environment, each IAdjacency has two adjacent vertices.

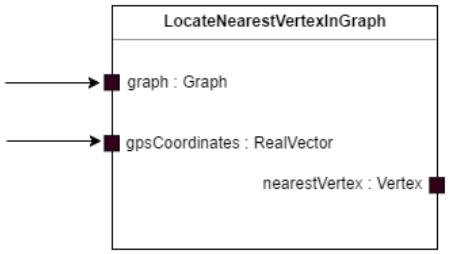
Outputs:

- **graph** a graph with vertexes and edges between two adjacent vertexes.

Purpose. to generate a graph for further calculating.

Implementation. for each IAdjacency with two nodes, get the id, osmid and coordinates of them, to calculate the position of them and assign them to two new vertexes and check if there already exists vertex in the list by id, if not, add it to the list, if yes, get the vertex with same id and generate an edge and add it to the edge map.

2.1.2 LocateNearestVertexInGraph



Inputs:

- **graph** A Graph with vertices and edges.
- **gpsCoordinates** The GPS coordinates of the position of the car

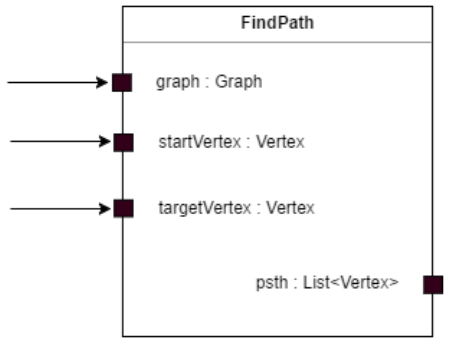
Outputs:

- **nearestVertex** the nearest vertex in the graph to the car

Purpose. LocateNearestVertexInGraph finds the nearest vertex in the graph to the car, it is also the start vertex of the path to be calculated.

Implementation. Calculate the distance between car and the distance of all vertices of the graph and select the closest one.

2.1.3 FindPath



Inputs:

- **graph** A Graph with vertices and edges as specified in Graph.
- **startVertex** start vertex of the path.
- **targetVertex** end vertex of the path, specified by simulation group in simulation main function.

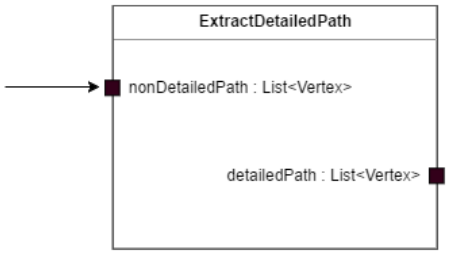
Outputs:

- **path** list of vertices which represents the path the car should drive along to the destination.

Purpose. FindPath calculates the path, which are represented by a list of vertices one after other.

Implementation. Use Dijkstra's algorithm to find the shortest path from the start vertex to the end vertex in a graph.

2.1.4 ExtractDetailedPath



Inputs:

- **nonDetailedPath** the list of vertices, calculated by FindPath.

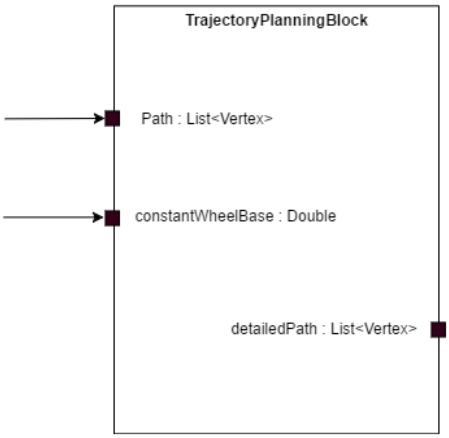
Outputs:

- **detailedPath** a path with more vertices along the nondetailedPath, it means that a the list of vertices is interpolated linearly between each adjacent pair of vertices along the nondetailedPath.

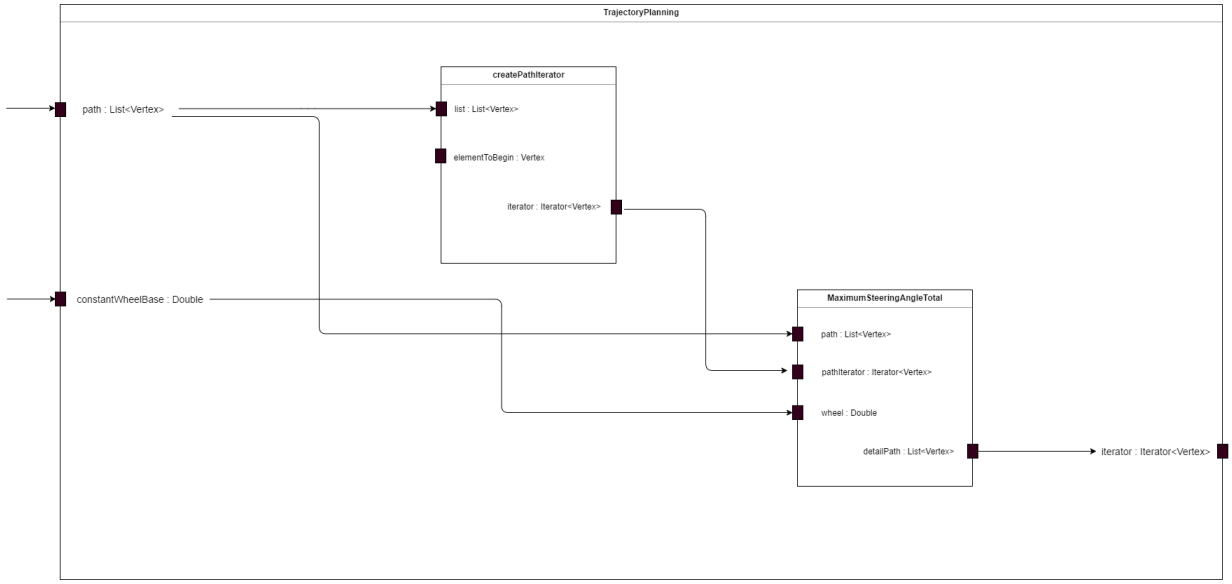
Purpose. Improves the resolution of the path, so that the MainControlBlock can use this for controlling issues.

Implementation. If there exists more than two vertices in the nondetailedPath, then insert (distance/interpolation_distance) vertices in between.

2.1.5 TrajectoryPlanningBlock



2.2 Trajectory Planning Block



Inputs:

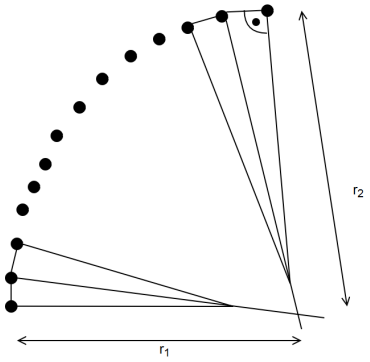
- **path** the path, for that all maximal steering angles should be calculated
- **constantWheelbase** the wheelbase of the car

Outputs:

- **detailedPath** the original path with calculated maximal steering angle for every vertex

Purpose. The TrajectoryBlock calculates for every vertex in the path the maximal steering angle one can steer at that vertex.

Implementation. See also in figure below. For that, firstly the path is partitioned in sections of the length of LENGTH_OF_SECTIONS meters, which one can find in the FunctionBlockParameter enum as tunable parameter.
Secondly, for every section of the path the first three vertices are determined. These are called in the same order beginningInFront, beginning and beginningBehind and are DISTANCE_OF_MEASURING_POINTS meters apart. Moreover, the same is done for the last three vertices, they are called endInFront, end and endBehind.
Thirdly, the perpendicular vectors on beginning and end are calculated approximately by using the vectors that are perpendicular to the vectors from beginningInFront to beginning and beginningBehind to beginning. Now, these perpendicular vectors are build up to a line with beginningInFront and beginningBehind as position vectors. The vector from beginning to the intersection point of those two lines is the perpendicular vector to beginning regarding the path. Analogously, this is done for end, too.
Fourthly, the minimal distance of the line through the "middle point" of the curve is the minimal radius of the curve. So, what we do is to approximate the curve with a part of the circle to calculate the maximal steering angle.

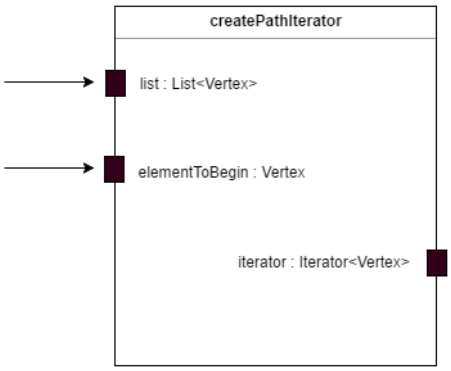


Fifthly, with this minimal radius $r = min(r_1, r_2)$ one can determine the maximal steering angle for this curve by using the formula:

$$\alpha = arcsin(\frac{w}{r}),$$

where α is the steering angle in the circle, w is the wheelbase of the car, and r the radius of the circle.

2.2.1 CreatePathIterator



Inputs:

- **list** A List
- **elementToBegin** The element, where the iterator should begin

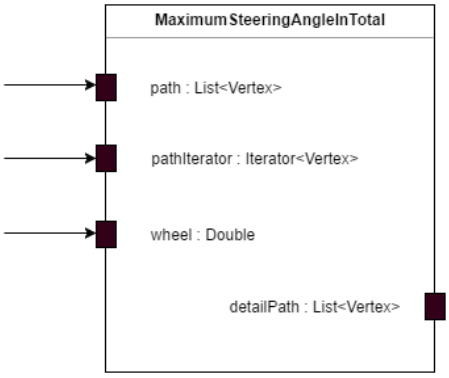
Outputs:

- **iterator** The iterator of the list which begins at the specified element

Purpose. CreateIterator creates an iterator for a list.

Implementation. If elementToBegin is null, then the iterator begins at the start of the list.

2.2.2 MaximumSteeringAngleInTotal



Inputs:

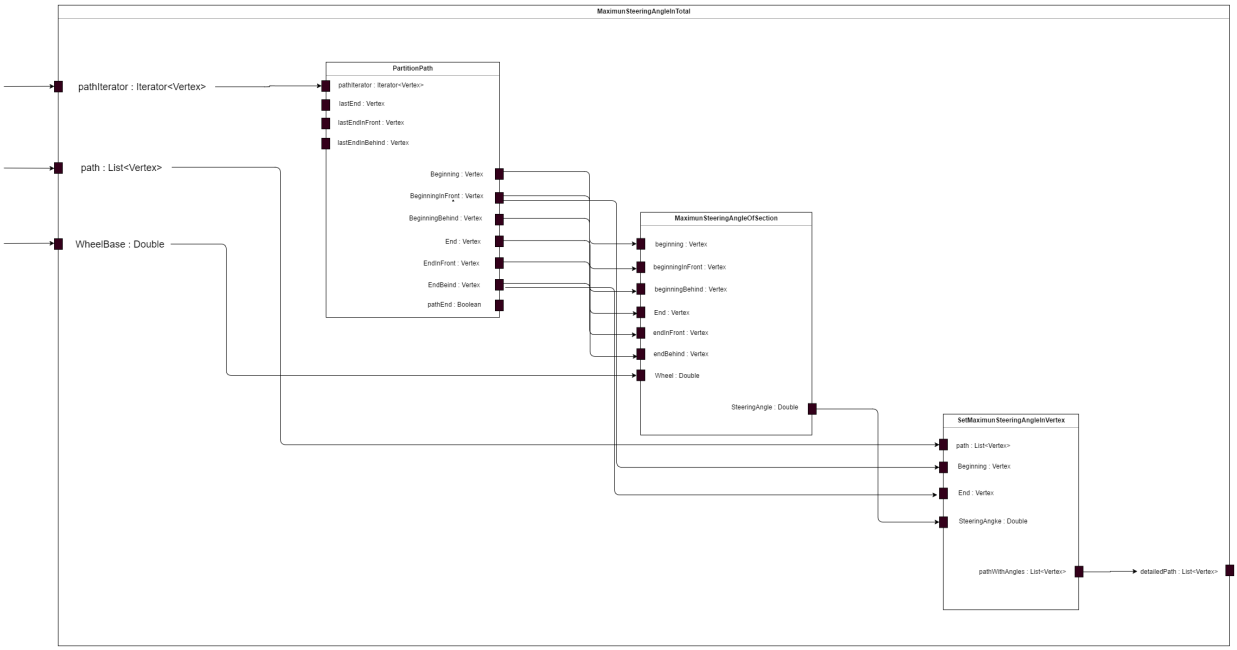
- **path** List of vertices, which represents a path.
- **pathIterator** Iterator of the path with the beginning vertex.
- **wheel** The wheelbase of the car.

Outputs:

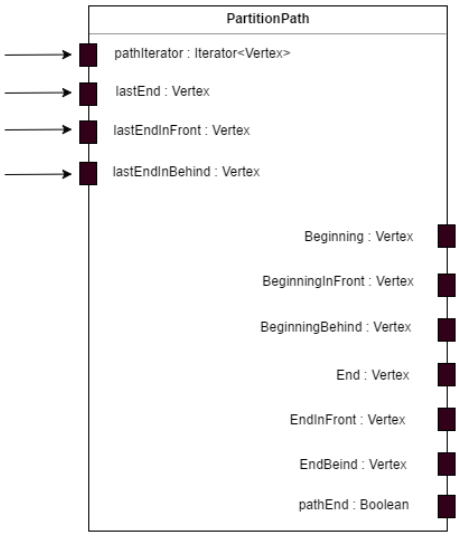
- **detailedPath** The detailed path with maximal steering angle of each vertex.

Purpose. MaximumSteeringAngleInTotal calculates the maximal steering for each vertex that is contained in the path.

Implementation. The idea is presented in the trajectory planning block.



2.2.3 PartitionPath



Inputs:

- **pathIterator** Iterator of the path.
- **lastEnd** End vertex from last section
- **lastEndFront** The vertex directly in front of end vertex from last section.
- **lastEndBehind** The vertex directly behind end vertex from the last section.

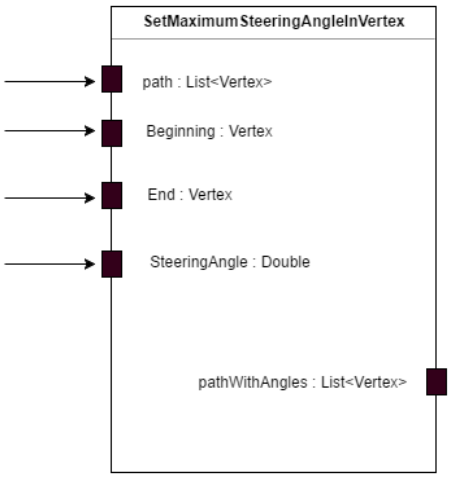
Outputs:

- **Beginning** the beginnging vertex of this section.
- **BeginninginFront** the vertex directly in front of the beginning vertex of this section.
- **BeginningBehind** the vertex directly behind the beginning vertex of this section.
- **End** end vertex of this section.
- **EndinFront** the vertex directly in front of the end vertex of this section.
- **EndBehind** the vertex directly behind the end vertex of this section.
- **pathEnd** a flag check whether reaches to the end of the path.

Purpose. PartitionPath partitions the path into sections and returns 6 vertices for later steering angle calculation.

Implementation. These 6 vertices for every section in PartitionPath in a while-loop, which is to check if reach to the end of the path, you can find it in MaximumSteeringAngleInTotal block. After get 6 vertexes for section 1, the execute() of MaximumSteeringAngleInTotal will call partitionPath again until the end.

2.2.4 SetMaximumSteeringAngleInVertex



Inputs:

- **path** a list of Vertexes, represent a path.
- **Beginning** the beginning vertex of this section.
- **End** the end vertex of this section.
- **SteeringAngle** the maximal steering angle of this section, which is calculated by MaximumSteeringAngleOfSection, will be explained later.

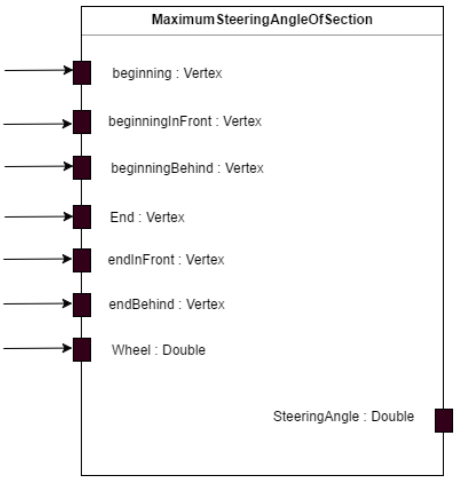
Outputs:

- **pathWithAngle** partial or all vertexes(last section finished) of path with maximal steering angle.

Purpose. set the calculated maximal steering angle of this section to each vertex in this section.

Implementation. find all vertexes between beginning vertex and end vertex in the path and assign them with this angle.

2.2.5 MaximumSteeringAngleOfSection



Inputs:

- **Beginning** the beginnging vertex of this section.

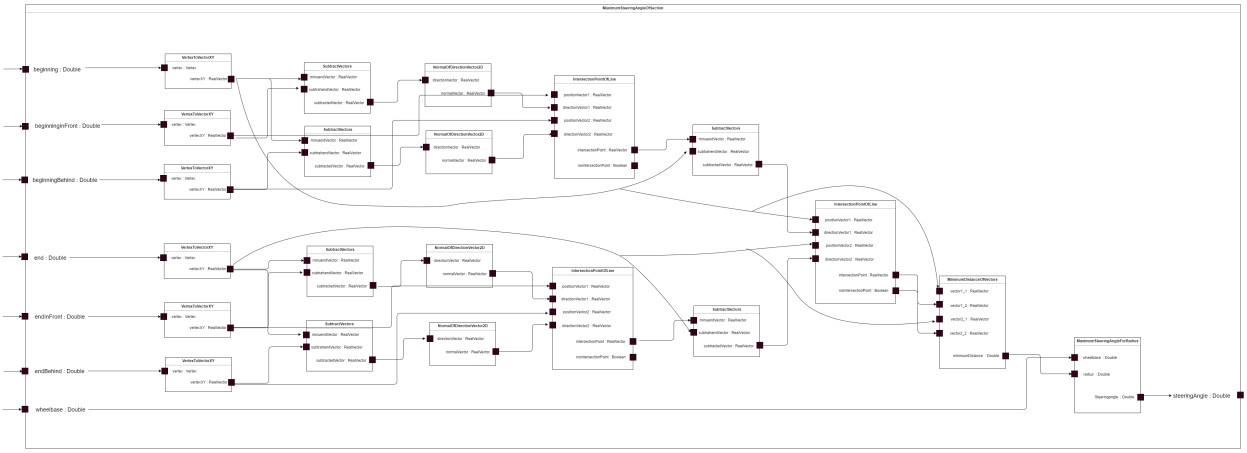
- **BeginninginFront** the vertex directly in front of the beginning vertex of this section.
- **BeginningBehind** the vertex directly behind the beginning vertex of this section.
- **End** end vertex of this section.
- **EndinFront** the vertex directly in front of the end vertex of this section.
- **EndBehind** the vertex directly behind the end vertex of this section.
- **pathEnd** a flag check whether reaches to the end of the path.
- **wheelbase** the wheelbase of the car.

Outputs:

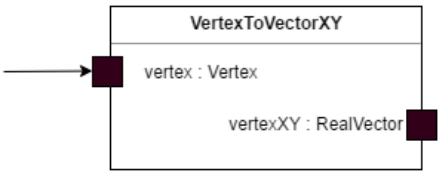
- **steeringAngle** the maximal steering angle for this section

Purpose. MaxmumSteeringAngleOfSection calculates the maximal steering angle for every section, whereby it gets the control points from PartitionPath.

Implementation. The exact implementation can be found in Trajectory Planning Block.



2.2.6 VertexToVectorXY



Inputs:

- **vertex** a vertex with id, osmID, position, maximal steering angle.

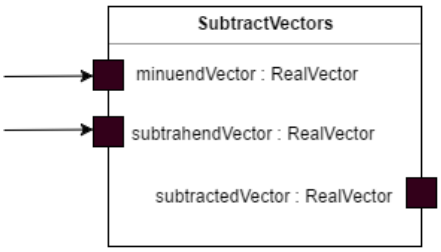
Outputs:

- **vectorXY** a RealVector with the position coordinates from vertex.

Purpose. VertexToVectorXY converts a vertex into a RealVector that contains the x and y coordinate of the position of the vertex.

Implementation. constructs corresponding RealVector by getting the position from the vertex.

2.2.7 SubtractVectors



Inputs:

- **minuendVector** the first RealVector
- **subtrahendVector** the second RealVector

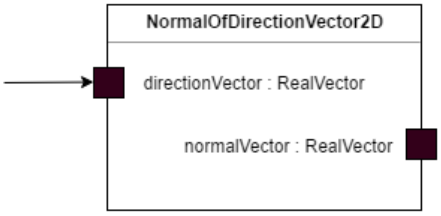
Outputs:

- **subtractedVector** the result of the subtraction of both vectors.

Purpose. SubtractVectors subtracts two vectors.

Implementation. By invoking subtract() of the RealVector interface.

2.2.8 NormalOfDirectionVector2D



Inputs:

- **directionVector** direction vector.

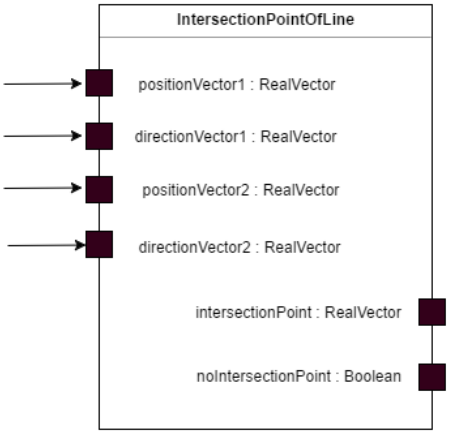
Outputs:

- **normalVector** normal vector of the direction vector.

Purpose. NormalOfDirectionVector2D calculates the (unified) normal vector of the direction vector in 2D.

Implementation. By assigning values to make the normal vector cross product direction vector to be 0.

2.2.9 IntersectionPointOfLines



Inputs:

- **positionVector1** the position vector of the first line
- **directionVector1** the direction vector of the second line
- **positionVector2** the position vector of the second line
- **directionVector2** the direction vector of the second line

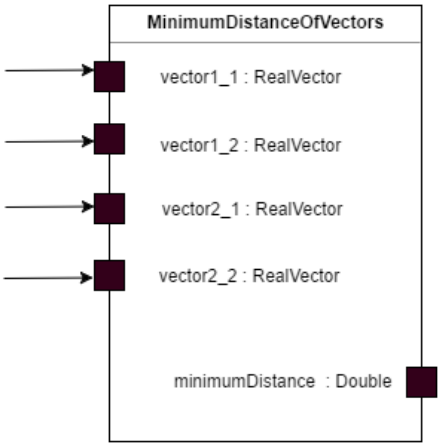
Outputs:

- **intersectionPoint** the intersection point of the two lines, if an intersection point exists
- **nointersectionPoint** iff no intersection point, return true.

Purpose. IntersectionPointOfLines calculates the intersection point of two lines in 2D.

Implementation. The intersection point is calculated by solving a linear equation system by using a matrix decomposition (LU decomposition).

2.2.10 MinimumDistanceOfVectors



Inputs:

- **vector1_1** beginning vector

- **vector1_2** intersection vector 1
- **vector2_1** end vector
- **vector2_2** intersection vector 2

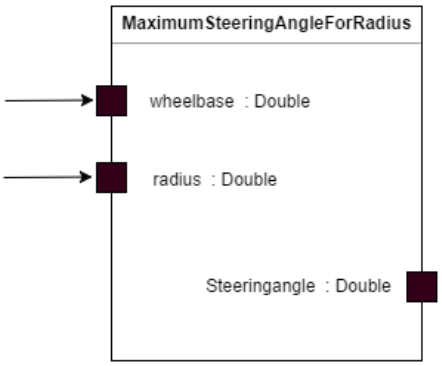
Outputs:

- **minimumDistance** the smaller one of the distance between beginning vector and intersection vector 1 and the distance between end vector and intersection vector 2.

Purpose. MinimumDistanceOfVectors calculates the minimal radius of the path section.

Implementation. By using Math.min() function.

2.2.11 MaximumSteeringAngleForRadius



Inputs:

- **wheelbase** the wheelbase of the car.
- **radius** the minimal radius calculated before.

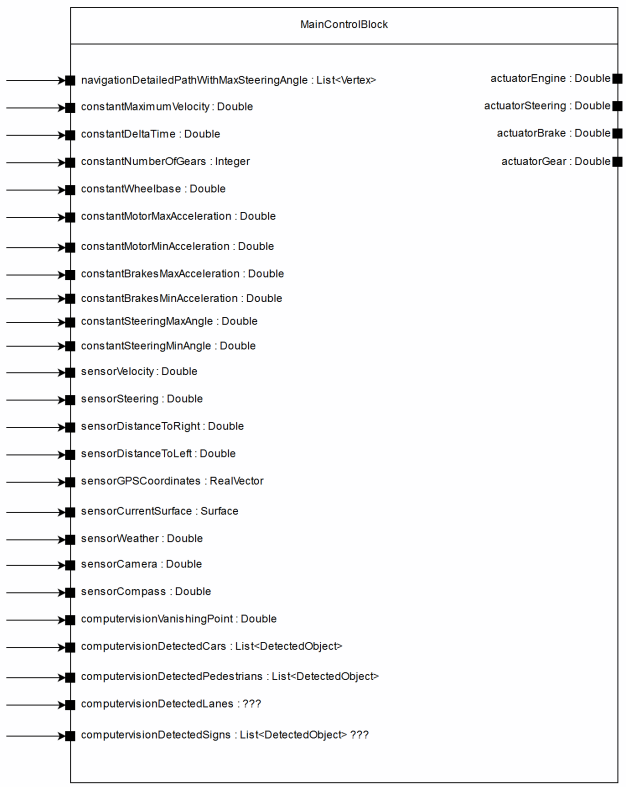
Outputs:

- **steeringAngle** the maximal steering angle the car could drive dependent on the wheelbase and radius.

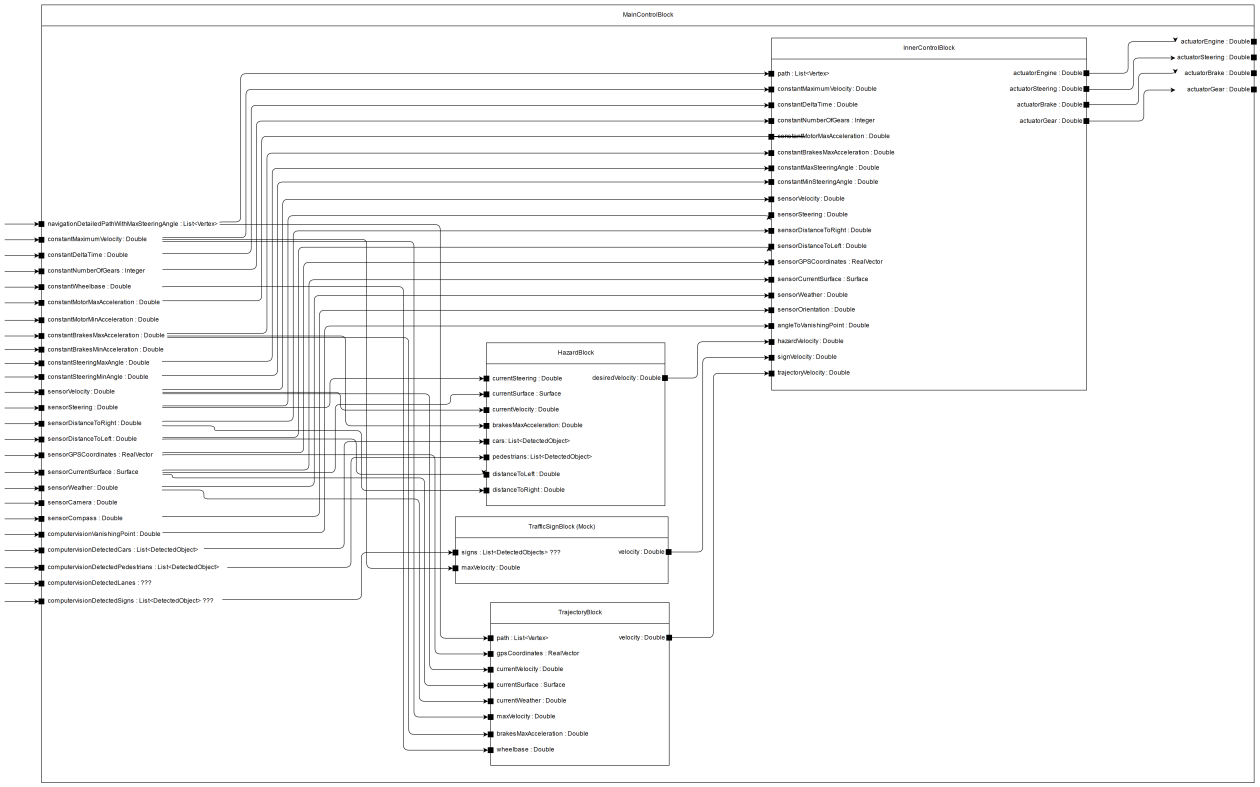
Purpose. MaximumSteeringAngleForRadius calculates the maximal steering angle the car could drive in this path section.

Implementation. By using Math.asin(wheelbase/radius).

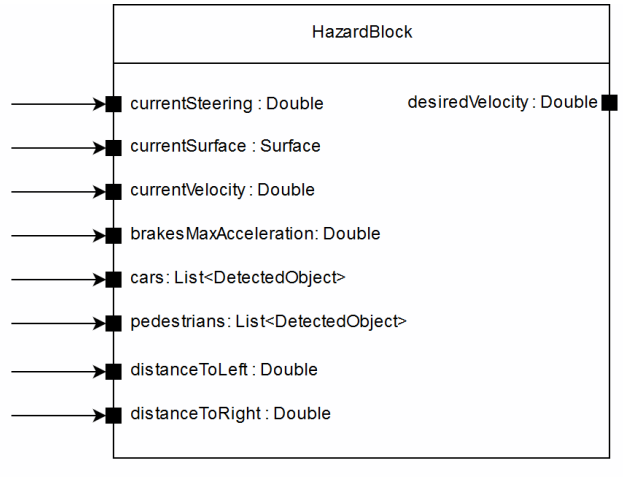
2.3 Main Control Block



The MainControlBlock is the block that provide all control functionality. It gets all Inputs from the Bus and gives all outputs to the Bus. The Implementation of all sub blocks can be found in this and the following chapter.
The connections are as follows:



2.4 HazardBlock



Inputs:

- **currentVelocity** current velocity measured by sensor
- **brakesMaxAcceleration** constant value, one feature of the brake system in the car.
- **cars** list of detected cars with depth to our car and relative angle to our car provided by CV group.
- **pedestrians** list of detected pedestrians with depth to our car and relative angle to our car provided by CV group.
- **distanceToLeft** distance to left mark lane in meter provided by sensor.
- **distanceToRight** analogous to above.
- **maximumVelocity** constant value, one feature of the car.
- **widthOfCar** constant value, one feature of the car.
- **safeDistance** constant value, defined by me, equals to 2 meters.

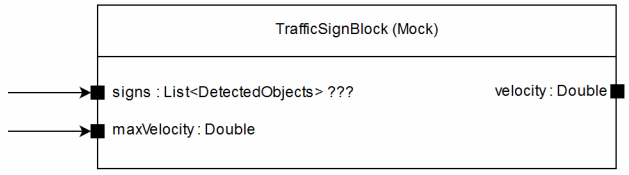
Outputs:

- **desiredVelocity** the velocity calculated by hazard logic and return to velocity logic to calculate the maximal velocity in current frame.

Purpose. consider the car and pedestrians in front and calculate the velocity we should drive in next frame.

Implementation. the basic idea is first check if there exists a pedestrian in road and if there exists a car in front with conflict with our car, if our car could overtake with current velocity. If yes, then calculate the nearest pedestrian or car, which makes this conflict.

2.5 TrafficSignBlock



Inputs:

- **signs** the velocity shows in the traffic sign.

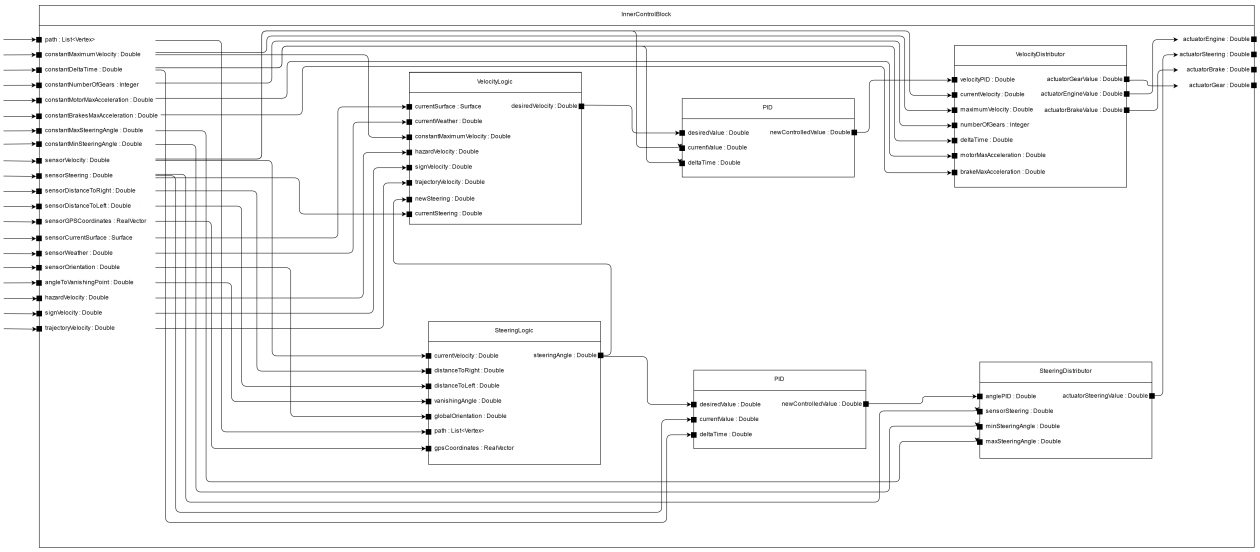
Outputs:

- **velocity** the velocity equals to the velocity shows in the traffic sign.

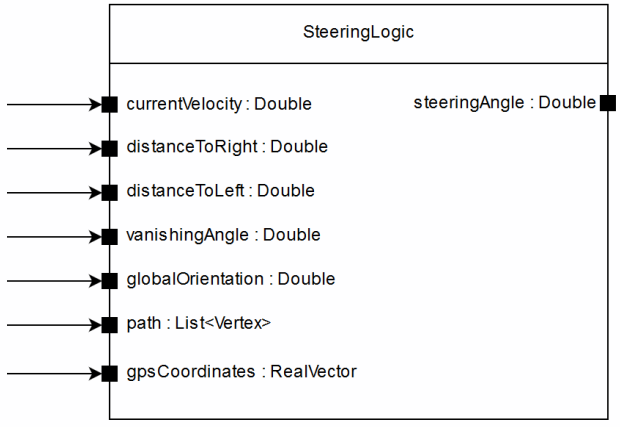
Purpose. to obey the traffic rule.

Implementation. set the current maximal velocity to the sign detected velocity.

2.6 Inner Control Block



2.6.1 SteeringLogic



Inputs:

- **currentVelocity** the velocity of the car measured by the according sensor in $[\frac{m}{s}]$
- **distanceToRight** the nonnegative distance to the side stripe on the right side in $[m]$
- **distanceToLeft** analogue to previous
- **vanishingAngle** the angle to the vanishing point of the picture from the camera sensor in radians, range is $[-\pi, \pi]$ (e.g. an angle of $-\frac{\pi}{2}$ means that the car has to steer 90° to the right in order to head directly to the vanishing point)
- **globalOrientation** the angle from the y-axis of the global coordinate system to the projection of the car's direction on the xy-plane in radians, range is $[0, 2\pi]$ (e.g. an angle of $\frac{3}{2}\pi$ means that the car has to steer 270° to the right in order to drive along the y-axis)
- **path** the path which is calculated by the controller
- **gpsCoordinates** the coordinates in the global coordinate system in $[m]$

Outputs:

- **steeringAngle** the inverted desired steering angle in radians, range is $[-\pi, \pi]$ (e.g. an angle of $-\frac{\pi}{2}$ means that the car has to steer 90° to the left in order to head to the desired direction)

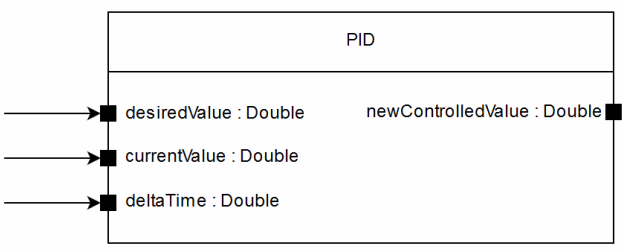
Purpose. The purpose of the SteeringLogic is to calculate what steering is needed to keep the car safely on the street and follow the path to the destination of the car.
Internally, a helper function block is used to do the computations. This may be changed in the future due to efficiency.

Implementation. This function block calculates two angles: the first angle is the steering correction needed to stay on the street. The algorithm uses the side stripe distances to determine the error of the car's position to the middle of the street. The corresponding angle tries to correct this error on the next two meters assuming the car is still somewhere close to the middle of the street. The algorithm could also use the angle to the vanishing point for this purpose or a combination of both. Furthermore, the precision of the side stripe distance could be improved with the help of computer vision. However, it is not known in which form this data is present, so the SteeringLogic does not use this improvement.
The second angle is the steering correction needed to stay on the path. In order to so, the next two points are extracted and the car tries to steer onto the line through those points. Again, the error is calculated and the velocity is used to calculate an angle which corrects the error on the next two meters making the same assumption. The following formula is used to calculate the distance and on with side of the line the car is:

$$d = \frac{(y_2 - x_2) \cdot c_1 - (y_1 - x_1) \cdot c_2 + y_1 \cdot x_2 - y_2 \cdot x_1}{\|x - y\|}$$

with x being the point on the trajectory the car has just passed, y being the point on the trajectory the car is heading for and c being the car's position. If and only if d is negative, then the car will be on the left side of the trajectory. The absolut value of d is the distance to the trajectory.
For both angles, the arc tangent can be used with the corresponding distance as the opposite side of the triangle and two meters as the adjacent side of the triangle. However, you also have to add the difference of the directions of trajectory and car to the second angle.
The final angle is the arithmetic mean of both computed angles. Thus, the algorithm tries to minimize both errors at the same time giving them the same priority. Future testing could involve finding out which angle is more important.
Helper methods throw "IllegalArgumentException" when called with incorrect parameters (see javadoc for more information). An "IllegalStateException" means that the parameters were correct but the method failed to do its job because of unknown reasons. If this occurs, it will most likely be caused by a bug in this method.
Currently, only the angle for staying on the trajectory is used to simplify the tuning. In the future, the angle for staying on the street should be included and tuned.

2.6.2 PID for Steering



Inputs:

- **desiredValue** the desired steering angle calculated by SteeringLogic in radians in the range $[-\pi, \pi]$
- **currentValue** the current steering angle from the sensor in radians in the range $[-\pi, \pi]$
- **deltaTime** the duration since the last call of the controller in $[s]$

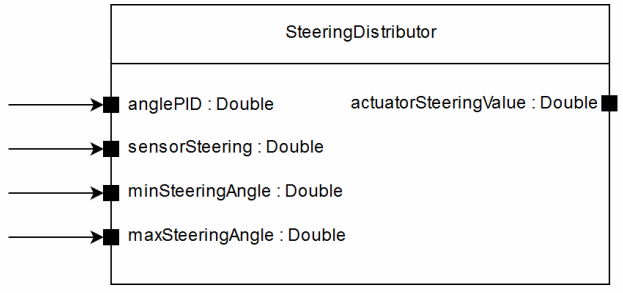
Outputs:

- **newControlledValue** the controlled steering angle in radians in the range $[-\pi, \pi]$

Purpose. This FunctionBlock controls the input of the steering actuator in order to allow smooth transitions while having a quick reaction to changes.

Implementation. The controller is implemented as a PID controller. This means, that the difference between desired value and current value is used to calculate a correction. Testing is required to decide whether a PD controller yields better results for steering. As the actuator and sensor use a clockwise rotation, the inputs and outputs are all clockwise.

2.6.3 SteeringDistributor



Inputs:

- **anglePID** the correction of the steering angle from the PID controller in radians in the range $[-\pi, \pi]$
- **sensorSteering** the current steering angle from the sensor in radians in the range $[-\pi, \pi]$
- **minSteeringAngle** the smallest angle which can be given to the actuator
- **maxSteeringAngle** the biggest angle which can be given to the actuator

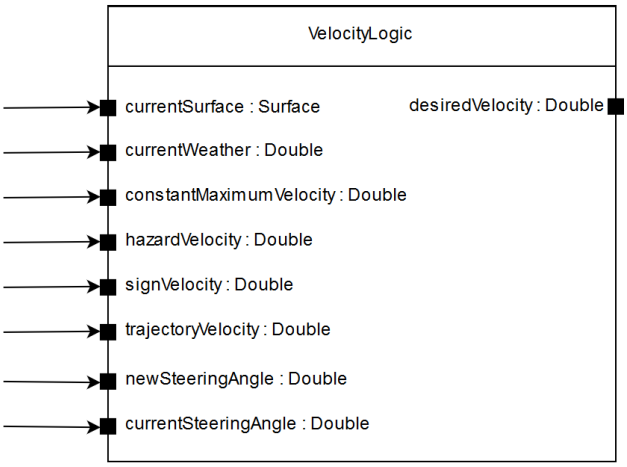
Outputs:

- **actuatorSteeringValue** the steering angle given to the actuator in radians in the range $[minSteeringAngle, maxStee$

Purpose. The SteeringDistributor passes the new steering angle to all devices which need the know this angle. Currently, this is only the actuator itself.

Implementation. The correction from the controller is added to the sensor value. If the sum does not violate a boundary, the sum will be returned. Otherwise, the according boundary is returned. As the actuator and sensor use a clockwise rotation, the inputs and outputs are all clockwise.

2.6.4 VelocityLogic



Inputs:

- **currentSurface** current surface parameter, used to calculate the maximal velocity by steering.
- **currentWeather** current weather condition, used to calculate the maximal velocity by steering.
- **constantMaximumVelocity** one specific feature of car 100m/s
- **signVelocity** the maximal velocity allowed in current street, sign detection is not implemented by CV, mocked by Christoph.
- **hazardVelocity** consider the position of the detected pedestrians and cars by CV, decide if need brake and thus return the desired maximal velocity
- **trajectoryVelocity** maximal velocity determined by TrajectoryBlock, which calculate the maximal velocity we can drive by using the maximal steering angle by pass a crossing.
- **newSteeringAngle** the new steering angle which was calculate by the steeringLogic before
- **currentSteeringAngle** the current steering angle of the car

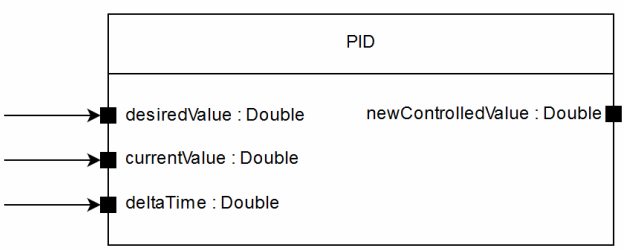
Outputs:

- **desiredVelocity** combine those different factors, return the velocity we can drive and pass it to the PID for velocity.

Purpose. So far consider the most important factors, and calculate the maximal velocity we can drive in the current environment and driving condition.

Implementation. Get all maximal velocities of different factors and select the smallest one.

2.6.5 PID for Velocity



Inputs:

- **desiredValue** the desired velocity calculated by VelocityLogic in m/s, $[0, ConstantMaximalVelocity]$
- **currentValue** the current velocity from the sensor in m/s.
- **deltaTime** the duration since the last call of the controller in $[s]$

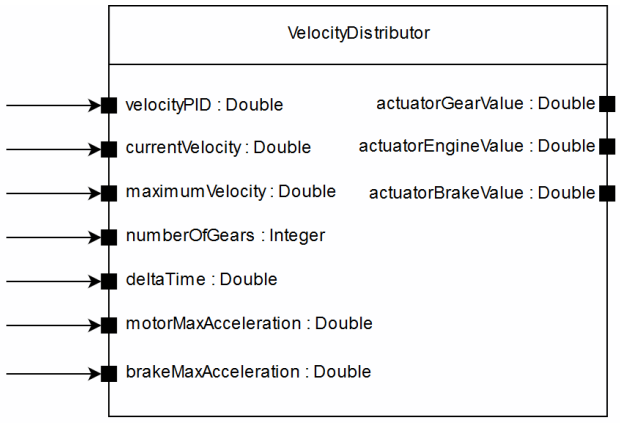
Outputs:

- **newControlledValue** the controlled velocity in m/s.

Purpose. This FunctionBlock controls the input of the velocity actuator in order to allow smooth transitions while having a quick reaction to changes.

Implementation. The controller is implemented as a PID controller. This means, that the difference between desired value and current value is used to calculate a correction. Testing is required to decide which kind of controller yields better results for velocity.

2.6.6 VelocityDistributor



Inputs:

- **velocityPID** the correction of the steering angle from the PID controller in radians, counter-clockwise in the range $[-\pi, \pi]$
- **currentVelocity** current velocity measured by sensor, used to calculate GearValue.
- **maximumVelocity** constant maximal velocity of car, used to calculate GearValue.
- **numberOfGears** constant value, one feature of car, used to calculate GearValue.
- **deltaTime** the duration since the last call of the controller in $[s]$
- **motorMaxAcceleration** constant value, one feature of motor in the car.
- **brakeMaxAcceleration** constant value, one feature of brake system in the car.

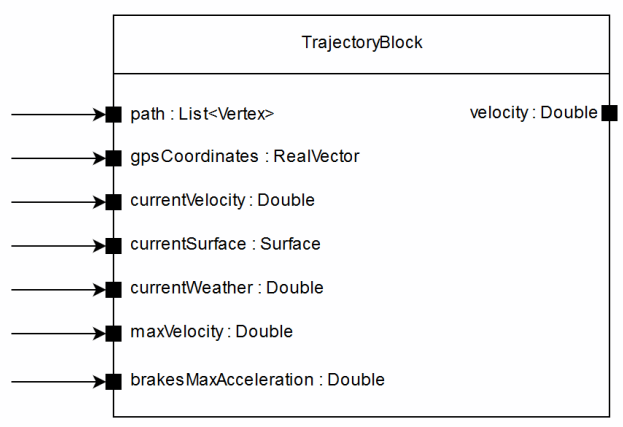
Outputs:

- **actuatorGearValue** if the velocity changes above the velocity limitation of certain gear, the gear value should be changed, calculate by the function as follow:
  GearValue = currentVelocity / (maximumVelocity + 0.00001 / numberOfGears) + 1
- **actuatorEngineValue** if the PID velocity large then the current velocity, then we calculate the acceleration of the car by velocityPID / deltaTime, and compare with motorMaxAcceleration, select the smaller one.
- **actuatorBrakeValue** analogous to actuatorEngineValue

Purpose. The VelocityDistributor passes the new velocity to all devices which need the know this velocity. Currently, this is only the actuator itself.

Implementation. The correction from the controller is added to the sensor value. If the sum does not violate a boundary, the sum will be returned. Otherwise, the according boundary is returned.

2.7 Trajectory Block



Inputs:

- **path** the path the car should follow
- **gpsCoordinates** the GPS coordinates of the current position of the car
- **currentVelocity** the current velocity of the car in $[\frac{m}{s}]$
- **currentSurface** the current surface the car drives on
- **currentWeather** the current weather coefficient in $[0, 1]$
- **maxVelocity** the maximal velocity the car can drive in $[\frac{m}{s}]$
- **brakesMaxAcceleration** the maximal brake acceleration in $[\frac{m}{s^2}]$

Outputs:

- **velocity** the maximal velocity for the trajectory in $[\frac{m}{s}]$

Purpose. The TrajectoryBlock uses the path with the maximal steering angles in every vertex constructed by the NavigationBlock to calculate the maximal velocity the car can drive in the considered section.

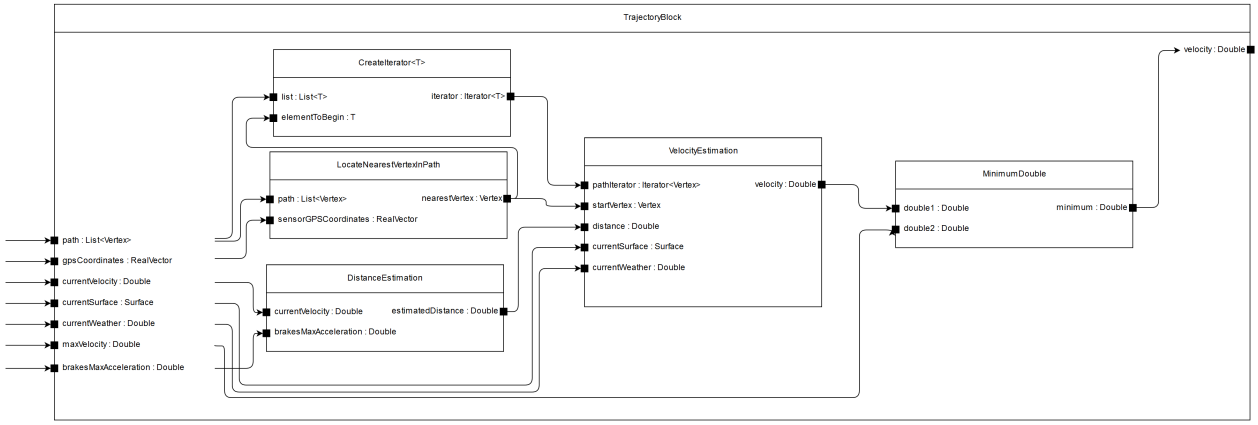
Implementation. The considered section has the nearest vertex to the car as start point and has a length of the distance calculated by using the formula

$$s = \frac{v^2}{2a} \cdot 1.5,$$

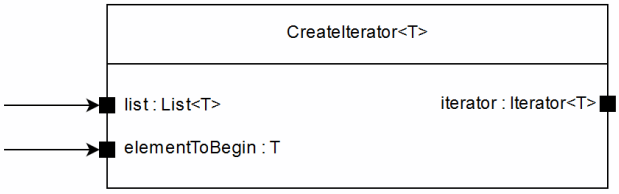
whereby a is the maximal brake acceleration, v is the current velocity and s is the distance. The multiplication by 1.5 is for safety, so that the car is able to brake in front of the critical point.
The following function blocks are used:

- CreateIterator
- LocateNearestVertex
- DistanceEstimation
- MinimumDouble
- VelocityEstimation

The connections are as follows:



2.7.1 CreateIterator



Inputs:

- **list** the list for which an iterator should created
- **elementToBegin** the element where the iterator should begin

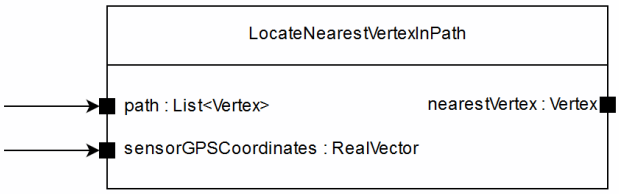
Outputs:

- **iterator** the iterator

Purpose. CreateIterator creates an iterator for the input of list.

Implementation. If the input of elementToBegin is not null the iterator begins at the position of that element. If elementToBegin is not contained in the list the iterator begins at the beginning of the list.

2.7.2 LocateNearestVertexInPath



Inputs:

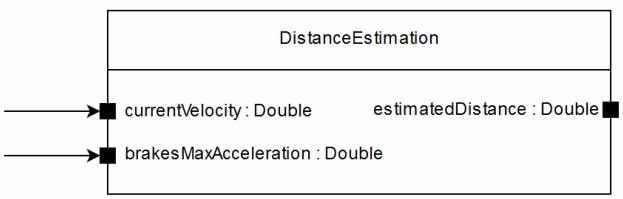
- **path** the path the car should follow
- **gpsCoordinates** the GPS coordinates of the current position of the car

Outputs:

- **nearestVertex** the nearest vertex to the position

Purpose. LocateNearestVertexInPath returns the nearest vertex to the current position of the car to the list of vertices you put in.

2.7.3 DistanceEstimation



Inputs:

- **currentVelocity** the current velocity of the car in $[\frac{m}{s}]$
- **brakesMaxAcceleration** the maximal brake acceleration in $[\frac{m}{s^2}]$

Outputs:

- **estimatedDistance** the calculated distance in $[m]$

Purpose. DistanceEstimation calculates the safety braking distance for a specific car.

Implementation. It uses the formula

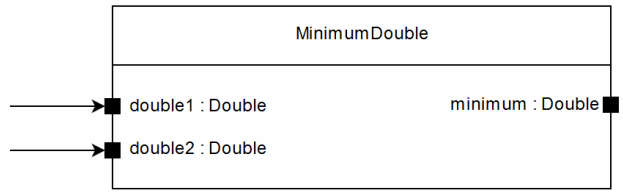
$$s = \frac{v_{cur}^2 - v_{new}^2}{2a},$$

where a is the maximal brake acceleration, v_cur is the current velocity, v_new the desired velocity and s is the distance. The formula is modified to

$$s = \frac{v_{cur}^2}{2a} \cdot 1.5,$$

so that there is a safety factor of 1.5 and $v_new = 0$ in order to calculate the braking distance.

2.7.4 MinimumDouble



Inputs:

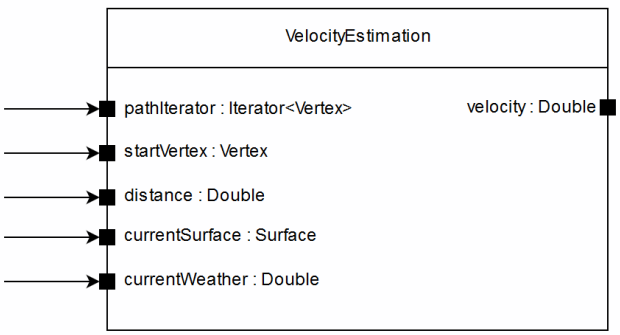
- **double1** the first double
- **double2** the second double

Outputs:

- **minimum** the minimum of both doubles

Purpose. MinimumDouble calculates the minimum of two doubles.

2.7.5 VelocityEstimation



Inputs:

- **pathIterator** the iterator of the considered path section
- **startVertex** the start vertex of the considered path section
- **distance** the length of the considered path section in $[m]$
- **currentSurface** the surface the car drives on
- **currentWeather** the coefficient of the current weather in $[0, 1]$

Outputs:

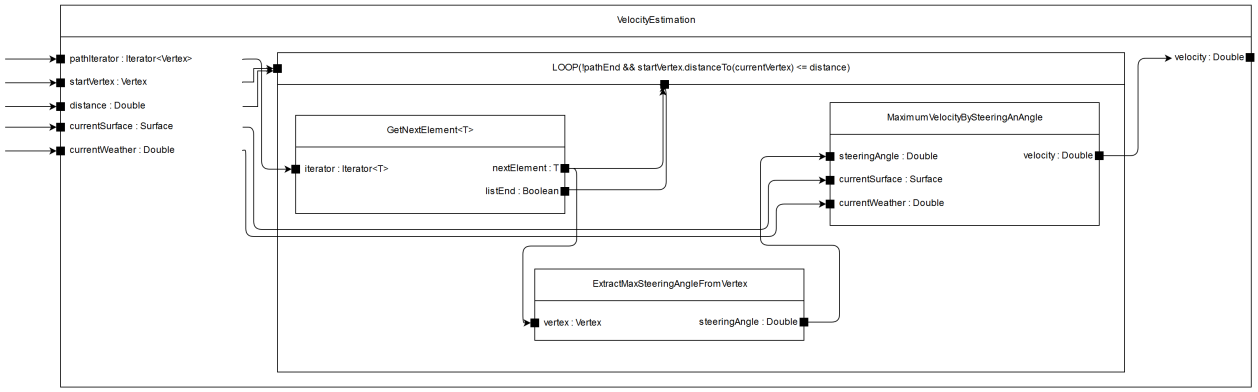
- **velocity** the calculated velocity of the car in $[\frac{m}{s}]$

Purpose. VelocityEstimation calculates the maximal velocity for every vertex gettable by the iterator starting from startVertex and ending if the distance is reached.

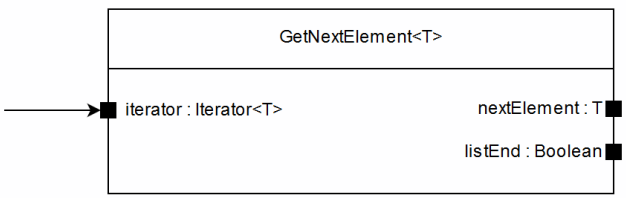
Implementation. The following function blocks are used:

- GetNextElement
- ExtractMaxSteeringAngleFromVertex
- MaximumVelocityBySteeringAnAngle

The connections are as follows:



2.7.6 GetNextElement



Inputs:

- **iterator** the iterator of the considered list

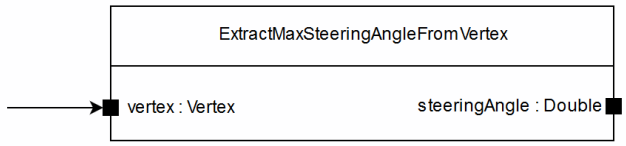
Outputs:

- **nextElement** the next element of the iterator
- **listEnd** contains if the iterator has a next element

Purpose. GetNextElement gets an iterator as input.

Implementation. It returns the the next element and if the iterator has no next element either before or after getting the last element it returns listEnd = true.

2.7.7 ExtractMaxSteeringAngleFromVertex



Inputs:

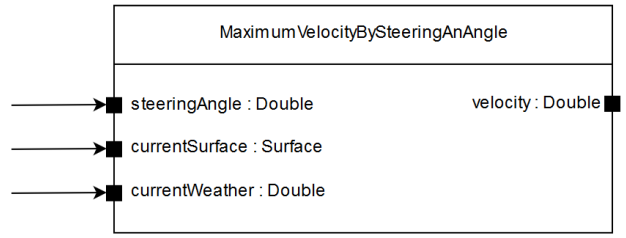
- **vertex** the vertex from which the maxSteeringAngle shpuld be extracted.

Outputs:

- **steeringAngle** the maximal steering angle one can drive in the vertex in $[rad]$

Purpose. ExtractMaxSteeringAngleFromVertex gets the maxSteeringAngle from a vertex.

2.7.8 MaximumVelocityBySteeringAnAngle



Inputs:

- **steeringAngle** the steering angle in $[rad]$
- **currentSurface** the surface the car drives on
- **currentWeather** the coefficient of the current weather in $[0, 1]$

Outputs:

- **velocity** the maximal velocity the car can drive dependent on the steering angle, surface and weather.

Purpose. MaximumVelocityBySteeringAnAngle calculates the maximal velocity you can drive by steering a certain angle under the conditions of weather and surface.

Implementation. It uses the Surface enum which stores and calculates the coefficients for every weather and surface. The following formula is used:

$$v = a \cdot exp(b \cdot |\phi|) + c,$$

where a is the first parameter, b is the second parameter and c the third of the surface/weather and ϕ is the steering angle.

3 Tuning

3.1 Tuning of PIDs

For tuning the PID controllers, the Ziegler-Nichols method was used. This methods works in the following way [PML]:
During the process of tuning, only the proportional coefficient k_p is not zero. The tuning consists of several iterations each starting in the same manner except for k_p.
The value of the controlled variable x starts at zero. The desired value of x is chosen as a fixed, positive value throughout the tuning. In the first iteration, k_p is set to a small positive value. The control loop is executed for some time and the values of x are measured. Depending on the result, k_p is increased or decreased and the next iteration starts with the $x = 0$.
There are a four different cases:

1. The value of x converges to the desired value but never exceeds it. Increase k_p in this case.
2. The value of x surpasses the desired value (overshoot) and then converges to the desired value. Increase k_p in this case.
3. The value of x oscillates consistently around the desired value. Set $k_{krit} = k_p$.
4. The amplitude of the oscillation of x increases with time. Decrease k_p in this case.

Once k_{krit} is found, the period t_{krit} of the oscillation needs to be measured. The final, tuned coefficients can be calculated from k_{krit} and t_{krit} with the help of different rules.
We used

$$k_p = 0.2 \cdot k_{krit}$$

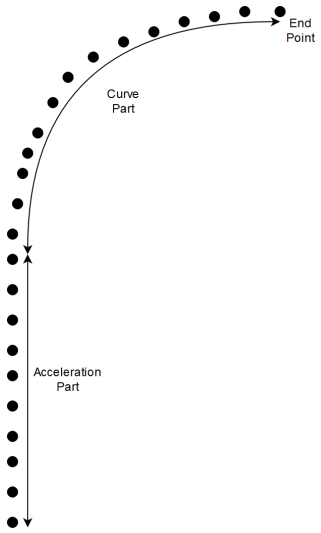
$$k_i = \frac{k_p}{t_{krit} \div 2}$$

$$k_d = k_p \cdot (t_{krit} \div 3)$$

for the velocity controller to lower the overshoot [Unk]. The results were $k_{krit} = 3$ and $t_{krit} = 2[s]$.
During tuning, there did not seem to be any correlation between the parameters of the steering controller and the course of the car. The parameters $k_p = 0.2$, $k_i = 0$ and $k_d = 0.5$ are currently used.
The tuning was done with precise sensor data because distortions seemed to make a tuning unpredictable.

There are several mock implementations of tuning classes in "montiarc4-features\features\autonomousdriving\controller\control\src\test\java\tuning". Notable are **TestVelocityHighResolution** and **TestZieglerNicholsTuning**. The latter executes Ziegler-Nichols Tuning automatically by using a binary search for k_{krit} as soon as an upper bound is found and linear search until then. The first allows to do this manually. Both print the measured values in a document.
The final tuning implementations are in "montiarc4-features\features\autonomousdriving\application-layer\src\test\java\tuning". The automatic version failed to find a stable control of the velocity. The reason is that the algorithm that analyses the course of the controlled variable is too simple. It simply searches for extrema by comparing consecutive values and calculates whether those extrema are in a certain neighbourhood of the arithmetic mean of the according extrema. However, the sensor values fluctuate quite heavily resulting in the algorithm to find local extrema outside of the range of the extrema of the precise value. The manual version was used to calculate the results currently used.
These tests seem to crash Jenkins for unknown reasons despite working locally. This is why most of the code is written as comments.

3.2 Tuning of Surface



For determining the velocity which can be driven in a curve, the maximal necessary steering angle of this curve is calculated. This steering angle limits the possible velocity in this curve. Thus, in the acceleration part the car is braked to this velocity depending on the maximal steering angle. The car can then safely drive through the curve part at this velocity.
Tuning is needed to determine the relation between velocity and steering. The current approach is to use:

$$v^* = a \cdot exp(b \cdot |\phi|) + c$$

where ϕ is the maximal steering angle and a and b c are tuning parameters. The value of a determines the velocity for straight streets and the value of b determines the rate of reducing the speed for sharper curves. The value of c determines the velocity in the sharpest curves. These parameters itself depend on the surface (mostly asphalt) and the weather, e.g. the amount of rain.
In order to execute the tuning, a trajectory describing a curve (figure above) is given to the car. The car shall drive on this trajectory with different parameters for a and b and c. For the evaluation of certain parameters, the positions of the car are compared to the positions of the trajectory. Furthermore, different approaches for changing the parameters at a certain amount of rain have to be tested. Currently, the amount of rain only influences the parameter b.
The tuning was done without rain. The most reliable result of the car seemed to be for $a = 2$ and $b = c = 0$. The car then tries to drive at $2[\frac{m}{s}]$ constantly. Different values were tested, good results were also achieved with $a = -15$, $b = -10$ and $c = 2$. However, the tuning is not finished. Bigger values for b should be tested for a safer behaviour of the car. Furthermore, there should be future improvements to the logics (see Prospect). Afterwards, tuning needs to be repeated.

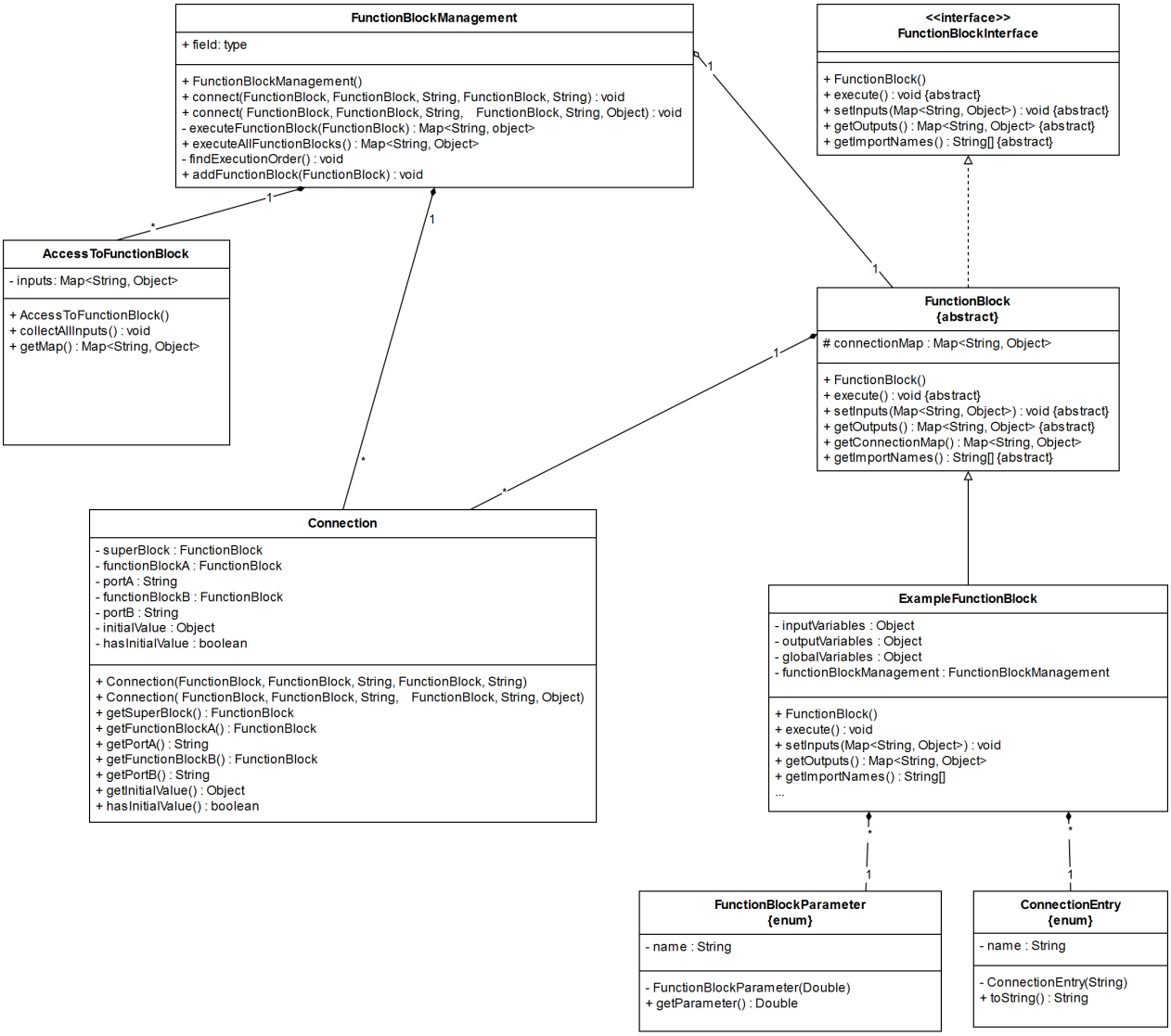
3.3 Tuning of TrajectoryPlanningBlock

The tuning of the TrajectoryPlanningBlock is simple. What you can tune is LENGTH_OF_SECTIONS and DISTANCE_OF_MEASURING_POINTS.
LENGTH_OF_SECTIONS defines the distance in meters between the beginning vertex and the end vertex of every section. What you want to do is to minimise this parameter to get a better solution for every section and therefor every curve.
DISTANCE_OF_MEASURING_POINTS is the distance between the beginning vertex and beginning in front as well as beginning behind. The same is true of the end vertex. See also in section PartitionPath. Tune this value balanced, because of the numerical extinction and the structure of the street. That is, the extinction should be in a lower order as 10^{-5} and all curves should be detected. A value in the range of $[0.2, 2]$ should be good. This range should be checked in terms of the curve detection.

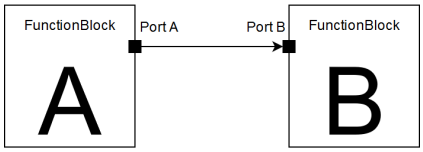
4 Modules And Data Structures

4.1 Function Block

The structure of every function block is specified below.



The class **FunctionBlockManagement** manages all sub blocks, connections and execution order for every function block. Sub blocks can be added to the super block via the method addFunctionBlock(FunctionBlock). Connections can be added via the method connect(FuctionBlock, FunctionBlock, String, FunctionBlock, String). The connection exists within a super block and begins at port A of function block A and ends at port B of function block B. Be sure to connect only function blocks that were added before, otherwise a IllegalFunctionBlockConnectionException is thrown.



To execute all sub blocks within the super block one calls executeAllFunctionBlocks(). This function returns a Map<String, Object> that contains all variables of the connections to the output ports of the super block, that one has lain, stored in the key which is specified in the connection. In short, if you called

connect(superBlock, functionBlockA, portA, superBlock, portB) you find the variable of portA of functionBlockA in the map in portB. Furthermore, executeAllFunctionBlocks() sorts all function block in terms of the execution order. Take note, that if a circular usage of function blocks occurs, to initialise at least the connection of one of those function blocks, so that a execution order can be found. To initialise a connection call connect(superBlock, functionBlockA, portA, superBlock, portB, initialValue), whereby initialValue is an object.
The execution order is determined as follows: We specify two sets S and $\overline{S}$. In the beginning in S contains only the super block. This is the block that has logically all ports initialised and assigned. $\overline{S}$ contains all sub blocks, that are not initialised and assigned, because we want to find an execution order. Now, for every loop iteration a sub block is found that has only connections from blocks in S. Then, we can add this block to S because then all connections are initialised and assigned. If such an block is found, it will removed from $\overline{S}$ and added to S. The loop runs until all blocks are in S. In that case, an execution order can be found because all connections can be assigned. If there are remaining blocks in $\overline{S}$ and there is no block one can add to S, then the algorithm states that it is an incorrect connection. That is, all remaining blocks are not connected such that all ports have a connection to blocks out of S and we know it will not be assigned. Then, an IllegalConnectionOfSubBlocksException is thrown.
The interface **FunctionBlockInterface** has the purpose to access a function block out of the project, it has all necessary methods for this.
The abstract class **FunctionBlock** realises the access to the connectionMap, so that the FunctionBlockManagement can access the inputs of the function block. This connectionMap is used to pass the inputs of a function block to their FunctionBlockManagement, so that AccessToFunctionBlock can assemble all inputs. For this, it is important to call super.setInputs(inputs) in every ExampleFunctionBlock. Also, there is checked whether any entry is null. If an entry is null, then an EntryIsNullException is thrown. You can prevent that a variable is checked if it is null by calling super.setInputs(inputs, String...).
The **ExampleFunctionBlock** can be every implementation of a function block e.g. MainControlBlock. **AccessToFunctionBlock** exists for every function block. It collects all inputs from all ports that are connected to the function block by calling collectAllInputs() to give the Map to the FunctionBlock.
ConnectionEntry is an enumeration that manages all connections and **FunctionBlockParameter** manages parameters for function blocks, see both also in chapter Entry Enumerations for more details.

4.2 Runtime Exceptions

4.2.1 EntryIsNullException

This exception is used to denote that an entry of the input map of a FunctionBlock is null. Therefor, it is thrown if an entry is null and is not marked that it can be null.
Solution:
Either check if the connections to the block are done correctly or mark the variable as it that can be null by calling setInputs(inputs, String...) with the name of the port as one part of the vararg.

4.2.2 IllegalConnectionOfSubBlocksException

This exception is used in findExecutionOrder() in FunctionBlockManagement. It is thrown, if no execution order can be found because of not assignable ports.
Solution:
Check the connections of the blocks. Maybe you did not connect all ports or a circular execution has no initialised ports, such that the order can be found.

4.2.3 IllegalFunctionBlockConnectionException

This exception is used in the function connect() in FunctionBlockManagement. It is thrown if non-added function blocks are connected.
Solution:
Add the function blocks you want to ceonnect before you connect them.

4.2.4 IllegalPathiteratorExceptionException

This is an Exception for the blocks PartitionPath and Velocity Estimation. It is thrown if an incorrect path iterator was passed.

4.2.5 IllegalTargetNodeExceptionException

This is an Exception for the FindPath block. It is thrown if the TargetNode is not present in the graph respectively adjacency list.
Solution:
Check if the target node that was given to the NavigationBlock is correct and present in the IAdjacency list that is passed to the NavigationBlock.

4.2.6 UnreachableVertexExceptionException

This is an Exception for the Dijkstra algorithm in the FindPath block. It is thrown if a vertex is not reachable.
Solution:
Check the node or if the street map is correct. If it is correct, what you did there, then catch this exception and show the "user" that the car can not drive to this point.

4.3 Entry Enumerations (enum)

4.3.1 ConnectionEntry

This enum administrates all ports of all function blocks which are used in this project. Every entry stands for one port of a function block. The signature of the entries is as follows:

- The name of the block is in uppercase letters and separated by underscores.
- The name of the variable the port assigned to is in lowercase letters and separated per underscore as well

Furthermore:

- non-indented entries are inputs
- tab-indented entries are outputs
- double tab-indented entries are global variables

4.3.2 BusEntry

This enum administrates all ports of the bus. Every entry stands for one port in the MainControlBlock. The signature of the entries is as follows:

- The name of the first word is the category of the variable (e.g. constant for a constant, sensor for a sensor value)
- The name of the variable

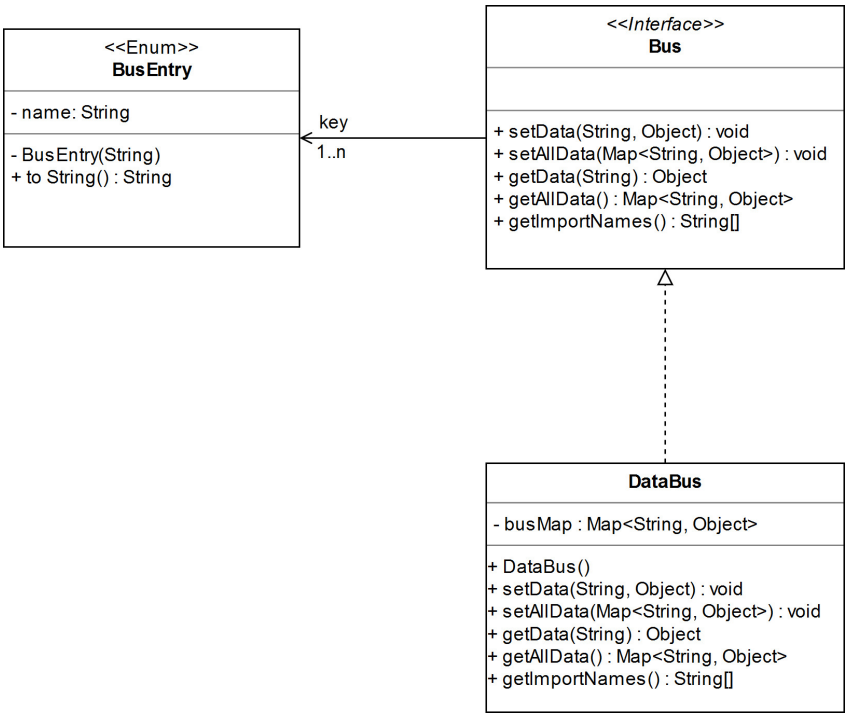
4.3.3 NavigationEntry

This enum administrates all ports of the NavigationBlock. Every entry stands for one port in the NavigationBlock. The signature of the entries is as follows:

- The name of the first word is the category of the variable (e.g. constant for a constant, sensor for a sensor value)
- The name of the variable

4.4 Bus

The structure of the Bus is specified below.



The interface **Bus** is the simulation of a real world bus (e.g. a CAN bus) which is used for the communication between controllers, sensors and actuators. The Bus resembles a map with keys of type String and values of type java.lang.Object. This was done because the address of a component linked to a real world bus could be interpreted as the key of data needed by this component. Furthermore, the methods of the function blocks then match those of the classes in \montiarc4-features\features\verification\montiarc-executionOrder\src\main\java\de\ma2cfg\simulator\atomic_blocks. The interface is known to every class which needs to communicate with the controller. Data can be written to the bus via setData(key, value) and read from the bus via getData(key).
The enum **BusEntry** acts as keys or addresses for the bus. Every enum constant represents one key and thus one type of data needed by one component. The String can be extracted from the constant via toString(). There is no need to hardcode the keys this way. Furthermore, corrupted data written to the bus is never accessed as every access uses a predefined enum constant.
The class **DataBus** is a straight forward implementation of Bus using LinkedHashMap from Java Platform, Standard Edition.

4.5 Graph

The class **Graph** is a data structure for a directed or undirected graph. This data structure is able to store all information needed for navigating the car, e.g. at crossings and forks. Furthermore, it can store the maximal necessary steering angle per location (per vertex) which is extracted by statically analysing the course of the road at that location.
Internally, Graph consists of a list of vertices and a function $f : V \times V \to E$ with V being the set of vertices and E the set of edges. The function is implemented with the help of nested maps. The Graph can either be constructed from an existing Graph or from a list of adjacencies using the class **IAdjacency** from the package "de.rwth.autonomousdriving.commons.environment.map.IAdjacency". Each adjacency corresponds to an edge and consists of the combination of two nodes, each holding the vertex information, and the distance between them.
An object of class **Vertex** is a vertex used in the graph. Each Vertex has an ID, a second ID originating from OpenStreetMap, a three dimensional position in space and the maximal necessary steering angle of the car around this position.
An object of class **Edge** is an edge used in the graph. Each Edge stores a distance and a capacity. The

capacity is currently unused but allows more generalized use of the graph data structure. An Edge does not depend on its incident vertices and therefore is not able to access them.

5 Prospect

There are lots of possibilities to improve the controller. Most of the constants which were used in formulas are assumed to be realistic. Some do not seem to have an impact, others had to be changed several times during tuning. This made tuning difficult.
Here are several suggestions with the most important ones first:
The behaviour of the controller has to be tested with more different test cases. Currently, the car is able to drive along a line and through a curve without losing control in or behind the car. However, longer tests have suggested that the car starts to drift several meters after stably leaving the curve and driving on a straight line which normally works fine. Whether this has to do with steering, velocity, the trajectory, simulation or tuning has to be tested still. Furthermore, the tuning was done with exact sensors because distortions seemed to have weird effects on the controller.

Currently, both logics can compute values which are much higher than what the actuator can handle. It has to be tested whether the steering logic should only give admissible angles to the controller as well as whether the velocity logic should test whether the desired value can be reached in the near future with the current velocity and the limitation of the motor. Furthermore, an acceleration controller could be implemented which takes the output of the velocity controller and smoothens the values for motor and brakes. All these steps would require a new tuning. However, it could be that some or even all of these steps could improve the driving behaviour significantly.

In the future, the functionalities of the car should be enlarged. For example, the controller does not distinguish between city and motorway. Thus, the car is either too slow or it starts to drift and crash. With a distinction, the car could use a broader range of velocities. Furthermore, the car does only try to stay on the trajectory at the moment. However, it also has to try to stay on the street by using the side distance sensors and computer vision. Apart from that, the trajectory is calculated in a high resolution with GPS which is unrealistic. Future implementations should use computer vision and laser sensors to calculate a high resoluted trajectory.

References

[Cor11] P. Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer Tracts in Advanced Robotics. Springer Berlin Heidelberg, 2011.

[PML] H. Papp, R. Moros, and F. Luft. Grundlagen regelung. `http://www.chemgapedia.de/vsengine/vlu/vsc/de/ch/7/tc/regelung/grundlagen/regelung_grundlagen.vlu/Page/vsc/de/ch/7/tc/regelung/grundlagen/regparam/regparam.vscml.html`. Last accessed on 20th March, 2017.

[Unk] Unknown. Ziegler-nichols method. `https://en.wikipedia.org/wiki/Ziegler%E2%80%93Nichols_method`. Last accessed on 20th March, 2017.