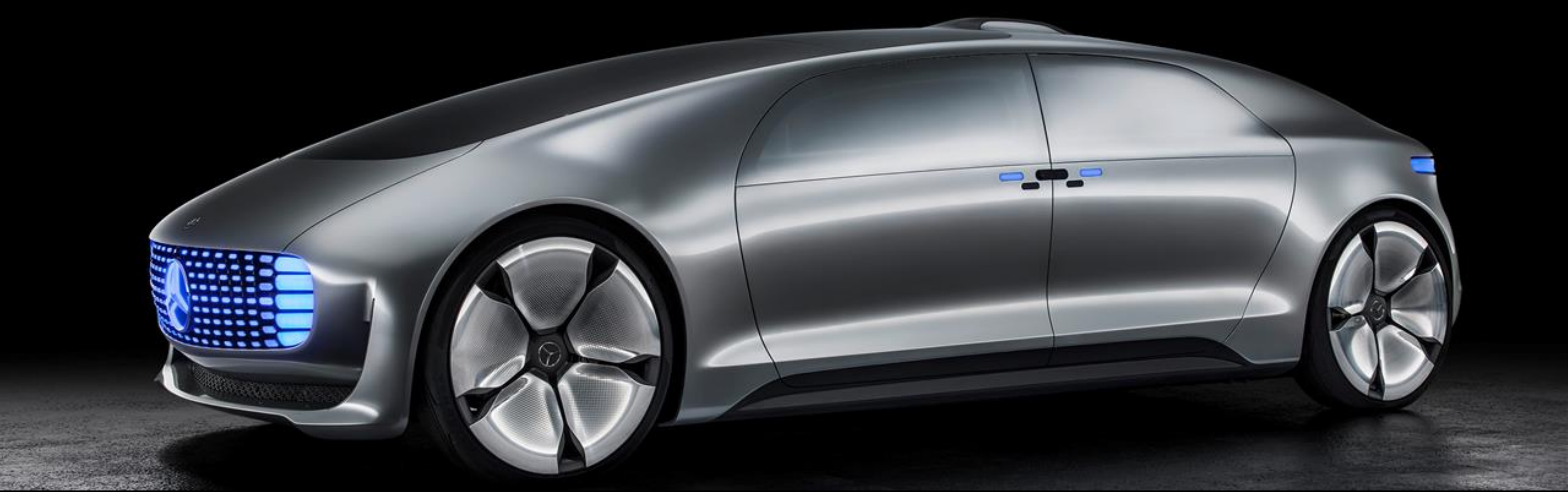


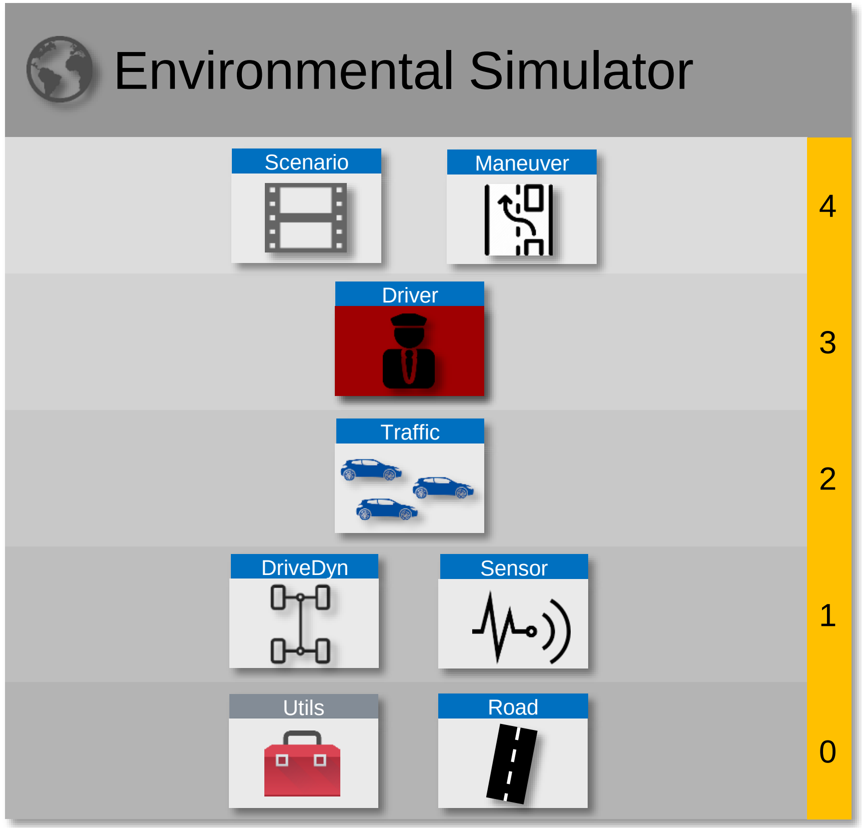
Abstract Environment API

Requirements for components *Map* and *VehicleController*

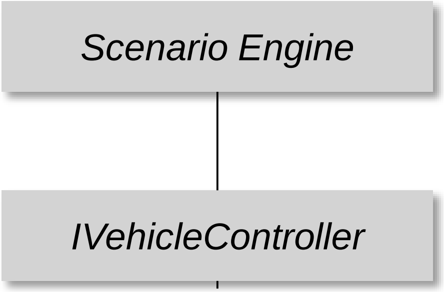
jupp.tscheak@daimler.com, Gärtringen, 2021-04-12

Mercedes-Benz
The best or nothing.

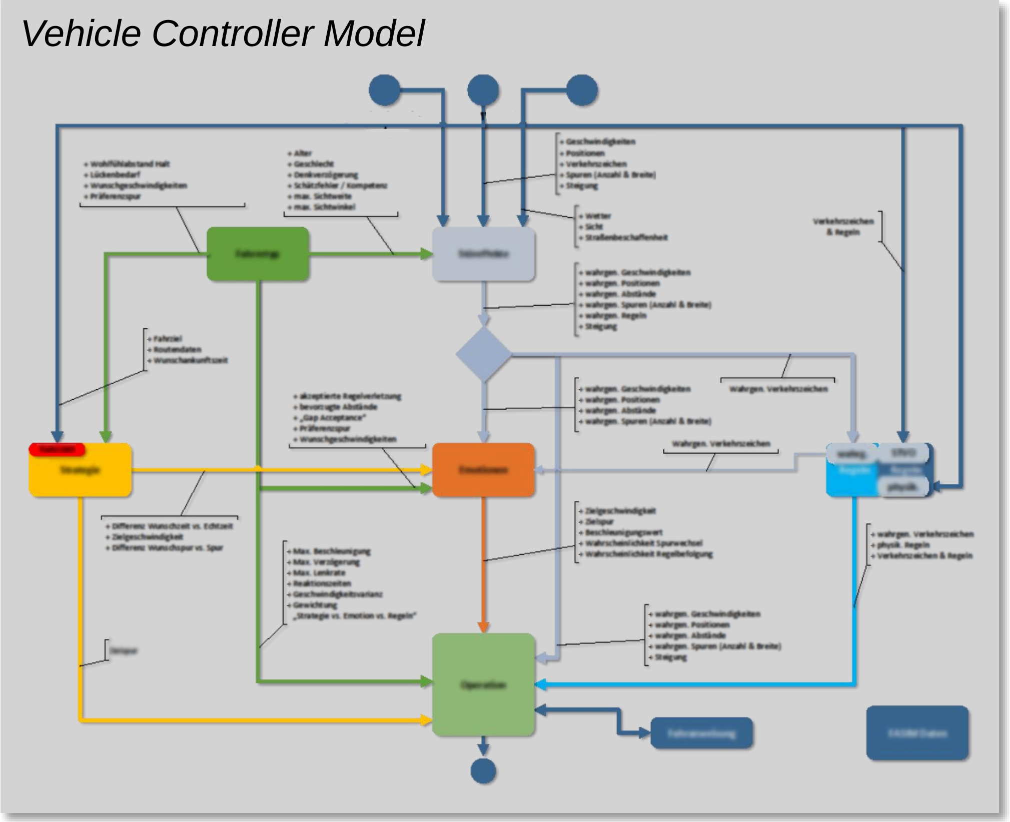
Requirements Of Components Map And VehicleController



A General Overview



The *VehicleController* interface should make as few assumptions of the underlying model implementation as possible.



VehicleController: Features

C++

```
virtual void EnableFeature(const std::string& name) = 0;
virtual void DisableFeature(const std::string& name) = 0;
virtual std::vector<std::string> GetFeaturesNames() const = 0;
```

A feature is a specific functionality provided by the underlying *VehicleController* model that can be enabled/disabled.

Example:
A model may implement the emotional state of a human driver which is influencing the desired velocity based on statistical calculations. In certain situations this behavior is unwanted and can be disabled explicitely by calling:
`DisableFeature(„Emotion“);`

VehicleController: Routes

C++

```
virtual void SetRoute(IRoute* route) = 0;
virtual const IRoute* GetRoute() const = 0;
```

A *VehicleController‘s* route is representing the navigational information about the destination and how it shall be reached.
In order to reach the destination, the *VehicleController* needs to perform navigational lane changes in front of intersections based on its parameters.

Question:
Do we also need to have a mode where the *VehicleController* is trying to follow a specific trajectory?

VehicleController: Vehicle

C++

```
virtual void SetVehicle(IVehicle* vehicle) = 0;
virtual IVehicle* GetVehicle() = 0;
```

Setter/Getter for querying the controlled *IVehicle.* The query of an *IVehicleController* of a *IVehicle* is important as well since instances managing the world entities will most probably store instances to *IVehicles.* In order to achieve “God” knowledge to provide an accident free simulation, a VehicleController model needs to have access to the state/properties of the surrounding *IVehicleControllers*.

VehicleController: Pause

C++

```
virtual bool IsPaused() const = 0;
virtual void SetPaused(bool paused = true) = 0;
```

Influences the property “pause” which stops/resumes the calculation of the underlying VehicleController model.

Example:
In certain situations the Scenario System wants to overtake the control of a vehicle e.g. to apply an unrealistic acceleration. After the desired effect has been achieved, control can again be returned to the vehicle controller model. The state of *IVehicleController* is preserved.

VehicleController: Lateral commands

C++

```
virtual void ChangeLane(LaneChangeDirection lane_change_direction) = 0;
virtual void Overtake() = 0;
```

Question: Do we need to command certain actions where lane changes are introduced?

Example:
A common use case in a scenario is to command an *IVehicleController* to overtake a certain vehicle. This should be done as fast as possible but respecting traffic rules not introducing unrealistic accelerations. Please note that the same behavior might be achieved by simply modifying parameters of an *IVehicleController* (e.g. “urge to overtake”).

VehicleController: Car Following / Lane Change Models

C++

```
virtual void SetCarFollowingModel(ICarFollowingModel* car_following_model) = 0;
virtual const ICarFollowingModel* GetCarFollowingModel() const = 0;
virtual ICarFollowingModel* GetCarFollowingModel() = 0;
virtual void SetLaneChangeModel(ILaneChangeModel* lane_change_model) = 0;
virtual const ILaneChangeModel* GetLaneChangeModel() const = 0;
virtual ILaneChangeModel* GetLaneChangeModel() = 0;
```

C++

```
class ICarFollowingModel {
 public:
  virtual units::acceleration::meters_per_second_squared_t GetAcceleration(IVehicle* veh_self, IVehicle* veh_lead,
                                                                            units::length::meter_t ds_lead) const = 0;

  virtual units::length::meter_t GetAdjustedHeadwayDistance(IVehicle* veh_self, IVehicle* veh_lead) const = 0;
};
```

C++

```
class ILaneChangeModel {
 public:
  virtual LaneChangeDecision GetLaneChangeDecision(IVehicleController* vehicle_controller,
                                                   IEnvironmentSimulator* environment_simulator) = 0;
};
```

Setter/Getter for accessing or providing car following and lane change models. Please note that this might be a contradiction to the previously mentioned premise to make as few assumptions of the model as possible.
However, it is a common use case to be able to provide a scenario specific model.

VehicleController: Parameters

C++

```
virtual void SetParameters(IVehicleControllerParameters* parameters) = 0;
virtual IVehicleControllerParameters* GetParameters() const = 0;
```

C++

```
class IVehicleControllerParameters {
 public:
  virtual void SetDesiredVelocity(const units::velocity::meters_per_second_t& desired_velocity) = 0;
  virtual units::velocity::meters_per_second_t GetDesiredVelocity() const = 0;
  virtual void SetMaxComfLongAcceleration(
      const units::acceleration::meters_per_second_squared_t& max_comf_long_acceleration) = 0;
  virtual units::acceleration::meters_per_second_squared_t GetMaxComfLongAcceleration() const = 0;
  virtual void SetMaxComfLatAcceleration(
      const units::acceleration::meters_per_second_squared_t& max_comf_long_acceleration) = 0;
  virtual units::acceleration::meters_per_second_squared_t GetMaxComfLatAcceleration() const = 0;
  virtual void SetMaxComfLongDeceleration(
      const units::acceleration::meters_per_second_squared_t& max_comf_long_acceleration) = 0;
  virtual units::acceleration::meters_per_second_squared_t GetMaxComfLongDeceleration() const = 0;
  virtual void SetDesiredHeadwayTime(const units::time::second_t& desired_headway_time) = 0;
  virtual units::time::second_t GetDesiredHeadwayTime() const = 0;
  virtual void SetMinGapFront(const units::length::meter_t& min_gap_front) = 0;
  virtual units::length::meter_t GetMinGapFront() const = 0;
  virtual void SetPoliteness(const units::concentration::percent_t& politness) = 0;
  virtual units::concentration::percent_t GetPoliteness() const = 0;
  virtual void SetSafeDeceleration(const units::acceleration::meters_per_second_squared_t& safe_deceleration) = 0;
  virtual units::acceleration::meters_per_second_squared_t GetSafeDeceleration() const = 0;
  virtual void SetLaneChangeRightBias(
      const units::acceleration::meters_per_second_squared_t& lane_change_right_bias) = 0;
  virtual units::acceleration::meters_per_second_squared_t GetLaneChangeRightBias() const = 0;
  virtual void SetLaneChangeThreshold(
      const units::acceleration::meters_per_second_squared_t& lane_change_threshold) = 0;
  virtual units::acceleration::meters_per_second_squared_t GetLaneChangeThreshold() const = 0;
};
```

Parameters influencing the behavior of an *IVehicleController*. Please note that the illustrated approach again is assuming too much about the model implementation, e.g. “LaneChangeRightBias” is a model parameter of lane change model “MOBIL” (Kesting, Treiber).

Maybe it is better to describe *IVehicleController* parameters in a more generic way and let the model implementation take care of how to achieve the desired behavior:

- “urge to overtake” [0..1]
- “respect speed limits” [0..1]
- …

But of course also:

- “desired speed” [m/s]
- “headway time” [s]
- “minimal gap front” [m]
- …

VehicleController: Parameter Factory

C++

```
enum class VehicleControllerType {
  kCalm = 1,
  kActive = 2,
  kSporty = 3,
  kAffective = 4,
  kUnsecure = 5,
  kAggressive = 6,
  kTruck = 7
};

class IVehicleControllerParameterFactory {
 public:
  virtual VehicleControllerParameters CreateParameterSet() = 0;
  virtual VehicleControllerParameters CreateParameterSet(VehicleControllerType type) = 0;
};
```

Factory for providing *IVehicleControllerParameters* that are used to introduce a variance of the same model implementation.

Example:
Assume a *IVehicleController* model implementation that is capable of driving on German roads. Still, a wide variety of driver behavior can be observed. In order to achieve different behavior, the parameters have to be different. A category of driver behavior is provided by *VehicleControllerType.* The implementation of *IVehicleControllerParameters* should take care of the statistically correct distribution of the various driver types based on local observations.

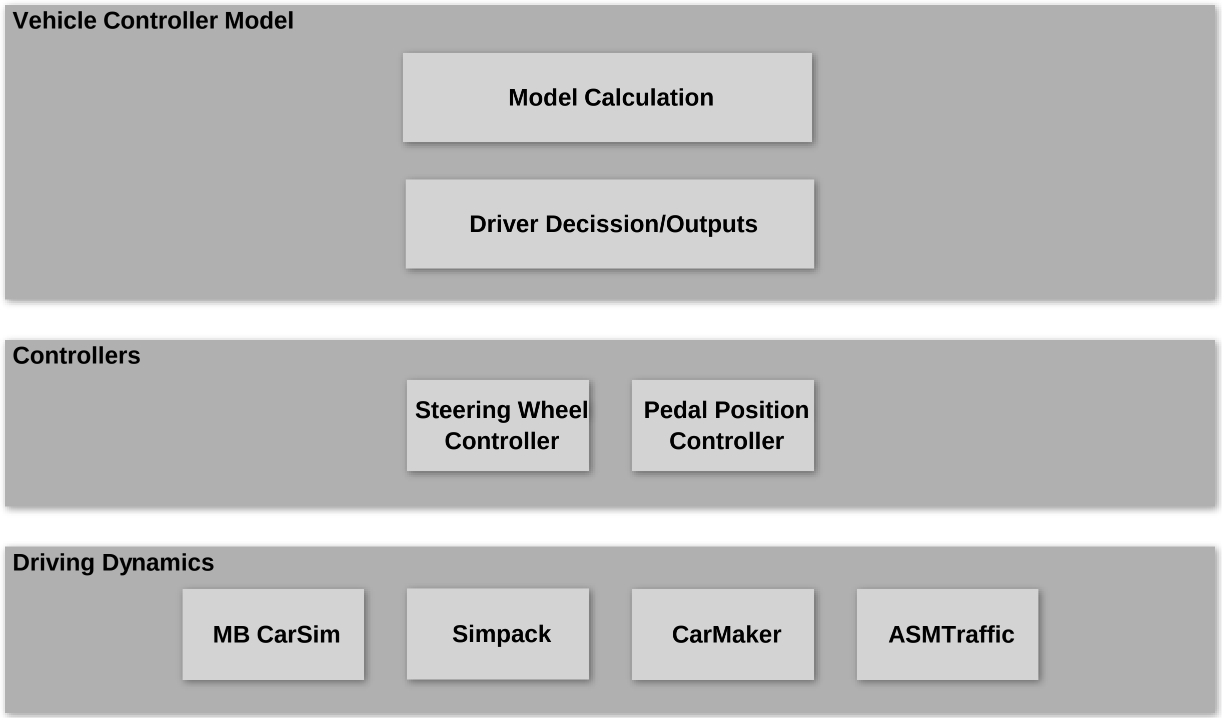
VehicleController: IVehicleController Factory

C++

```
class IVehicleControllerFactory {
 public:
  virtual IVehicleController* CreateVehicleController(IVehicle* vehicle,
                                                      units::velocity::meters_per_second_t desired_velocity) = 0;
  virtual IVehicleController* CreateVehicleController(IVehicle* vehicle,
                                                      const VehicleControllerParameters& parameters) = 0;
};
```

Factory for providing *a* specific implementation of the *IVehicleController* interface.

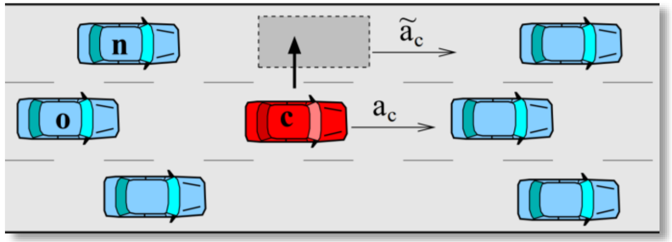
VehicleController: Calculation Results



Use Case: Vehicle Controller Model Using Abstract API

C++

```cpp
auto* entity_repo{environment_simulator->GetEntityRepository()};
auto* veh_self{vehicle_controller->GetVehicle()};
const auto& se{entity_repo->GetSurroundingEntitiesOf(veh_self)};

units::acceleration::meters_per_second_squared_t result{0_mps_sq};

if (lane_change_direction == LaneChangeDirection::kRight) {
  const auto& se_br{se.at(SurroundingVehicleLocation::kBackRight)};
  auto* veh_br{dynamic_cast<scenario::abstract::IVehicle*>(se_br.entity_)};
  scenario::abstract::ICarFollowingModel* cfm_br{nullptr};
  if (veh_br != nullptr && veh_br->GetVehicleController() != nullptr) {
    cfm_br = veh_br->GetVehicleController()->GetCarFollowingModel();
  }
  auto acc_br_old{veh_br != nullptr ? veh_br->GetAcceleration() : 0_mps_sq};
  auto acc_br_new{cfm_br != nullptr ? cfm_br->GetAcceleration(veh_br, veh_self, se_br.distance_) : 0_mps_sq};
  auto dacc_br{acc_br_new - acc_br_old};

  if (acc_br_new > vehicle_controller->GetParameters()->GetSafeDeceleration()) {
    const auto& se_bs{se.at(SurroundingVehicleLocation::kBackSame)};
    auto* veh_bs{dynamic_cast<scenario::abstract::IVehicle*>(se_bs.entity_)};
    const auto& se_fs{se.at(SurroundingVehicleLocation::kFrontSame)};
    auto* veh_fs{dynamic_cast<scenario::abstract::IVehicle*>(se_fs.entity_)};
    scenario::abstract::ICarFollowingModel* cfm_bs{nullptr};
    if (veh_bs != nullptr && veh_bs->GetVehicleController() != nullptr) {
      cfm_bs = veh_bs->GetVehicleController()->GetCarFollowingModel();
    }
    auto acc_bs_old{veh_bs != nullptr ? veh_bs->GetAcceleration() : 0_mps_sq};
    auto acc_bs_new{cfm_bs != nullptr ? cfm_bs->GetAcceleration(veh_bs, veh_fs, se_fs.distance_) : 0_mps_sq};
    auto dacc_bs{acc_bs_new - acc_bs_old};

    const auto& se_fr{se.at(SurroundingVehicleLocation::kFrontRight)};
    auto* veh_fr{dynamic_cast<scenario::abstract::IVehicle*>(se_fr.entity_)};
    scenario::abstract::ICarFollowingModel* cfm_self{vehicle_controller->GetCarFollowingModel()};
    auto acc_self_old{cfm_self->GetAcceleration(veh_self, veh_fs, se_fs.distance_)};
    auto acc_self_new{cfm_self->GetAcceleration(veh_self, veh_fr, se_fr.distance_)};
    auto dacc_self{acc_self_new - acc_self_old};

    result = dacc_self + vehicle_controller->GetParameters()->GetPoliteness() * (dacc_bs + dacc_br) -
             vehicle_controller->GetParameters()->GetLaneChangeThreshold() +
             vehicle_controller->GetParameters()->GetLaneChangeRightBias();
  }
}
```

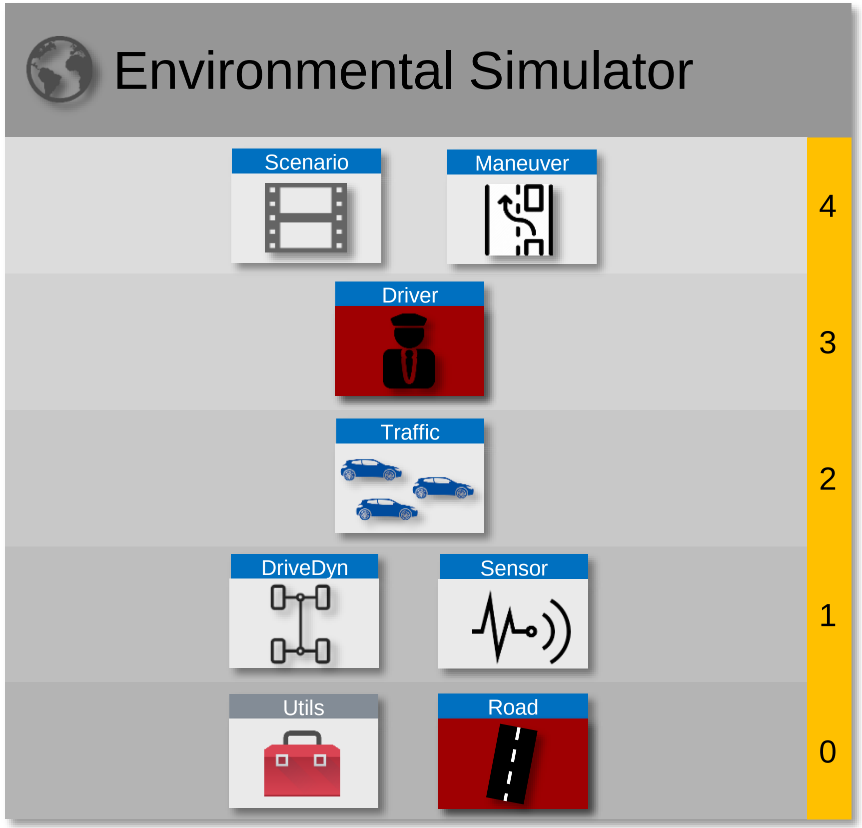


[1]

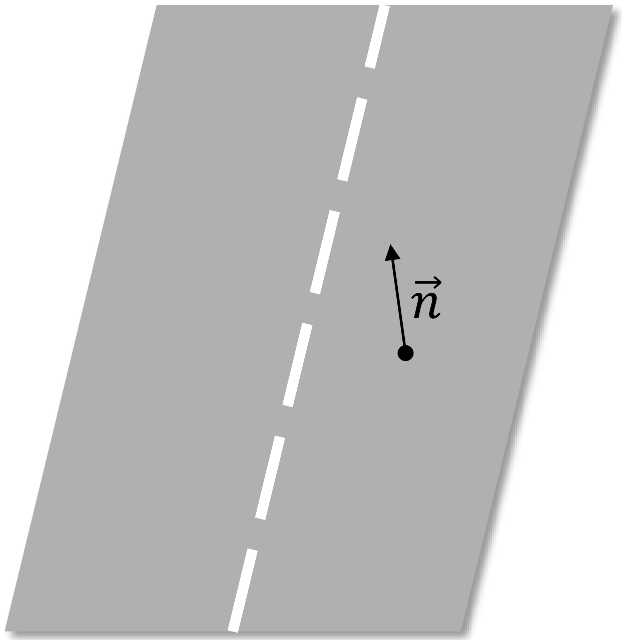
$$\underbrace{\tilde{a}_c - a_c}_{\text{driver}} + p\Big(\underbrace{\tilde{a}_n - a_n}_{\text{new follower}} + \underbrace{\tilde{a}_o - a_o}_{\text{old follower}}\Big) > \Delta a_{th}$$
[1]

[1] https://mtreiber.de/publications/MOBIL_TRB.pdf

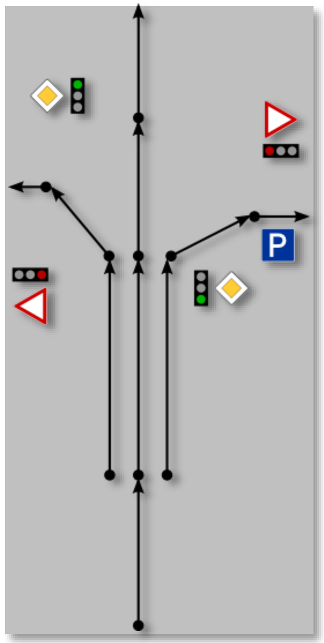
Requirements Of Components Map And VehicleController



Micro- and Macroscopic Queries

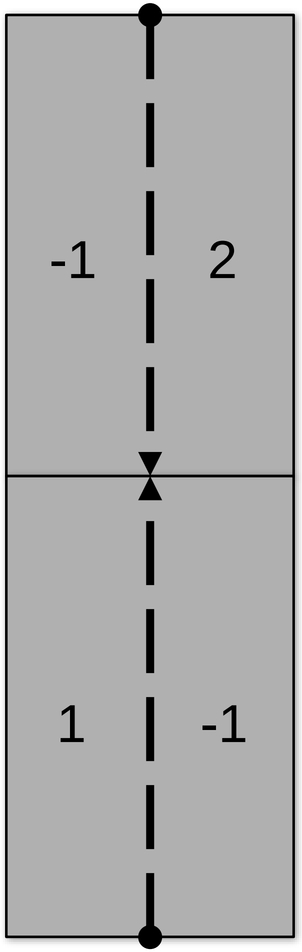


Microscopic



Macroscopic

Coordinate Systems



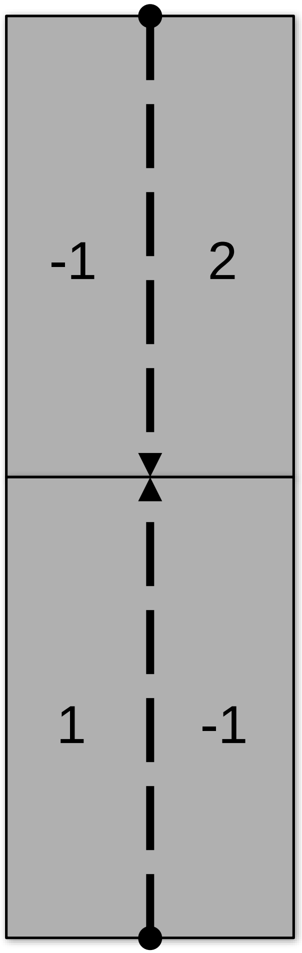
Basic:
- Inertial: (x, y, z)
- Track: (Id_{Road} , S, T)
- Lane: (Track, Id_{Lane} , $Offset_{Lane}$)
- Geo: (Lon, Lat, Alt)

Logical:
- Lane: (Track, Id_{Lane} , $Offset_{Lane}$)
- Offset always relative to driving direction.
  Note: Right-hand vs. left-hand traffic.
- Methods:
  - Lane GetLanePosAt(double dist);
  - double GetDistBetween(Lane other);
  - …?

Route:
- Provision of one continuous track.
- Basic coordinates could be used but with regard to underlying route.

Lane identification:
- LaneRightmost
- LaneMiddle
- LaneLeftmost
- …

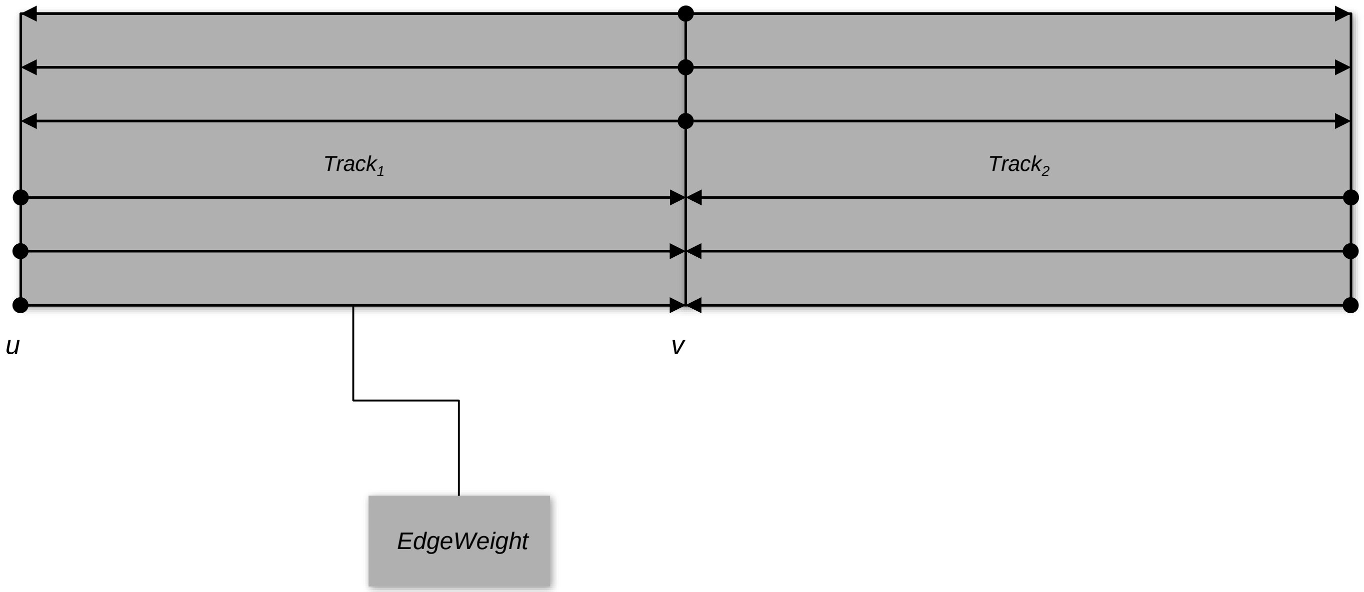
Microscopic Query



Provision of a *IQueryService* interface:

- Transformation between the different coordinate systems.
- Normal vector at position.
- Friction, curvature, angles (+derivatives) etc. at position.
- Road/Lane marks (maybe part of macroscopic query.)
- …

Macroscopic Query



$lane_{xy}$:= Edge(u, v)
$weight_{xy}$:= EdgeWeight(u, v)

Provision of a *IMap* interface:

- General layout of road network graph, road vs. lane based.
- Traversal of graph: iterators, functions, adjacent vs. consecutive lanes.
- Concept of a lane weight? Re-usage of OSI types?
  - Length
  - Signs
  - Lane marks
  - Etc.
- Routing algorithms. Create *IRoute* using waypoints.
- Traversal of junctions, turn directions etc.
- Consideration of a Horizont concept and dynamic reloading of parts of the map.

Vehicle Relation Graph