# BARK Documentation

***Release 1.0***

**fortiss GmbH**

**Jan 13, 2020**

# Contents

Bark is a semantic-simulation for multiple agents that are interaction aware.

CHAPTER 1

# About Bark

The name BArk is derived from **B**ehavior Benchm**Ark**. BArk focuses on the development and evaluation of behavior generation components for autonomous driving.

## 1.1 Need

Autonomous agents, such as traffic participants, need to make decisions in uncertain environments having many agents, which might be cooperative or possibly adversarial. Research in decision making brought up many approaches from the fields of machine learning, game, and control theory. However, transferring approaches to real-world applications, such as self-driving cars introduces a number of challenges, that prevent such systems to safely enter the market. One of the remaining challenges is a quantification of the expected performance of behavior generation approaches under true environmental conditions, e.g. unknown true behavior of other participants or uncertainty regarding the observations of the environment. This challenge is currently approached by driving endless amounts of kilometers in simulation-based and in recorded scenarios. However, such an approach impedes getting insights into the causes of performance differences of the evaluated approaches. Implemented improvements of an approach require frequent re-evaluation over the whole set of scenarios. To make behavior generation approaches ready for the real-world, an analysis framework should be established which can accurately model the divergence between real behavior of other participants and the behaviors generated by models (algorithms). Such a framework would allow for a more thorough investigation of the expected performance of behavior generation approaches under true environmental conditions. To make the claims of simulation-based performance results transferable to reality, benchmark scenario specifications must be selected according to certain coverage criteria and agreed on by the whole community of researchers and industry. BArk tries to tackle and solve the above mentioned challenges and problems.

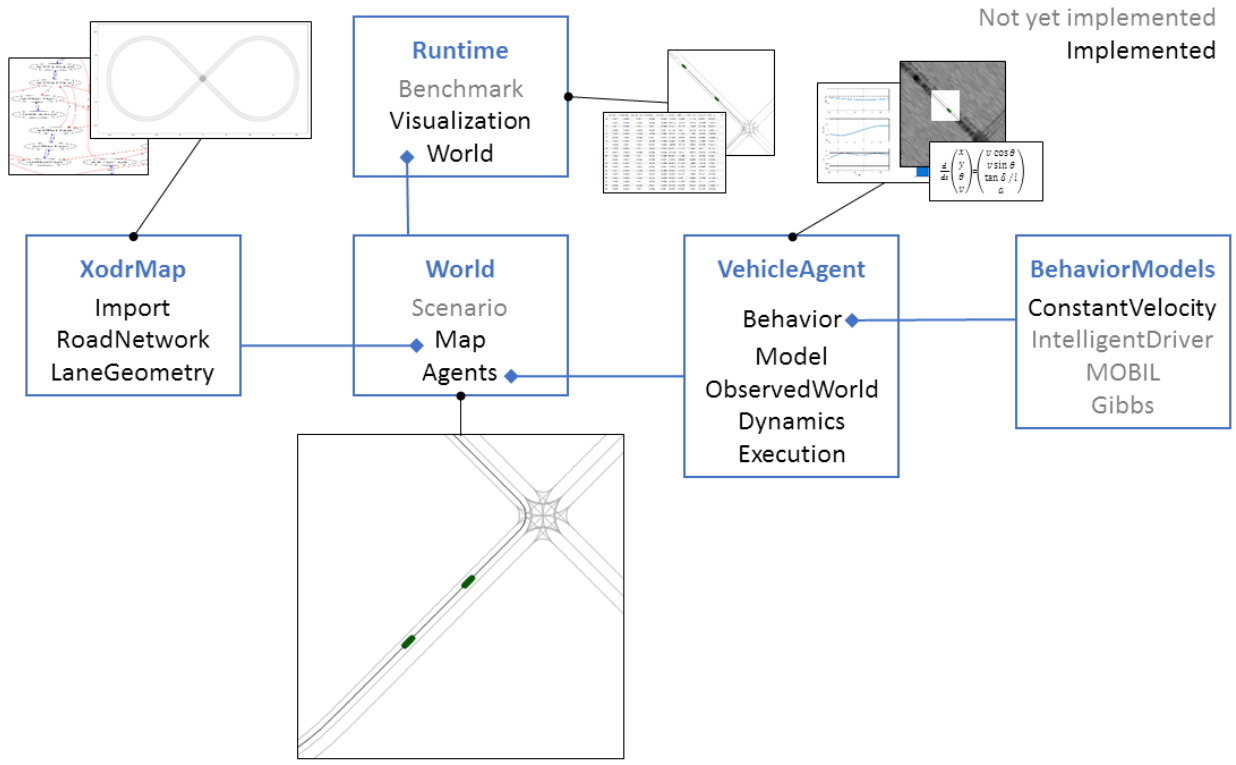
## 1.2 BArk Concept

BArk is a multi-agent simulation environment tailored towards the use case of autonomous systems, with a special focus on autonomous driving. Each agent is controlled by a behavior specification in form of a behavioral model. Behavioral models can be easily exchanged and used both for simulation of other participants and/or as behavior prediction on the behavior generation side. For this, Bark defines abstract interfaces for the development of own behavioral models but also delivers several state-of-the-art behavioral models based on machine-learning and classical

approaches. By additionally having a set of metrics and functions that evaluate individual components, BArk acts as a comprehensive framework for the development and verification of behavior generation approaches.

## 1.3 BArk Architecture

BArks modular architecture is as follows:




## 1.4 Existing Simulation Environments

There are already simulation solutions around, which also provide validation and benchmarking tools. Without a claim of completeness, we shortly compare them to BArk's goals.

- **Vehicle-centered component-based simulations** like veDyna (Tesis), Carmaker (IPG), ASM (dSpace), rFpro: Build around a complex an accurate model of the ego vehicle's dynamics, driving performance, or consumption can accurately be simulated. Primarily use case: (ADAS) control units HiL tests and homologation, also environment and traffic can be simulated. The focus is not on diverse behavior models of the traffic participants as the ego vehicle is considered as a test unit.
- **Multibody physics-based simulations** like Gazebo, SimScape/SimMechanics (Mathworks), Modelica libraries (VDL, Modelon): Correct simulation of the underlying physics is of importance and e.g. accurate sensor models can be achieved using ray tracing. The focus is not simulating the whole traffic scene with agent behaviors due to over-complexity.
- **Game engine simulators** like Carla (Intel), or AirSim (Microsoft): Based on a modern game engine, vision-based sensors and visualizations can be evaluated. Here models for the environment and the ego vehicle are present but a variety of behavior models for traffic is missing.

- **Traffic simulation** like Vissim (PTV), or Sumo (DLR): Microscopic traffic simulation allows individual simulation of each agent but without benchmarking and autonomous driving capacities. The "BEHAVE" project aims to add more sophisticated models for agents (human-driven and autonomously) to CityMoS.
- **Vehicle environment simulations** like Vires Virtual Test Drive (MSC) focus on the visualization and simulation of the environment and sensor setup and the testing process and not on the intelligence of the traffic agents.
- **Atari game simulations** are primarily focused on fundamental research on decision making.
- **Robotics benchmarking libraries** like Drake (MIT) or OMPL do not cover agent models for behavior benchmarks.

CHAPTER 2

# Changelog

## 2.1 v1.0 (March 1, 2019)

- initial release

CHAPTER 3

# Installation

This section describes the prerequisites and installation-steps of BARK. In order to be system independent, we decided to use a fully sandboxed build system (Bazel).

All features are currently supported under Linux/MacOS. For Windows, Bark only works for the C++ modules due to missing support of combining Bazel and Pybind11.

## 3.1 Prerequisites

- Bazel (requires Java)
- Python3
- virtualenv (Virtual environment)

### 3.1.1 Java (necessity for Bazel) on Mac:

We recomment to install JDK it manually from the OpenJDK download page

- Download OpenJDK for Mac OSX from http://jdk.java.net/
- Unarchive the OpenJDK tar, and place the resulting folder (i.e. jdk-12.jdk) into your /Library/Java/JavaVirtualMachines/ folder since this is the standard and expected location of JDK installs. You can also install anywhere you want in reality.

### 3.1.2 Bazel on Mac:

See https://docs.bazel.build/versions/master/install-os-x.html, we recommend installing using the binary installer

## 3.2 Linux/MacOS

### 3.2.1 Setup

1. Clone the repository, change to base repository directory
2. `bash install.sh`: this will create a virtual python environment (located in python/venv) and install all necessary python dependencies.
3. `source dev_into.sh`: this will activate the virtual environment (keep this in mind for the future: each time you use Bazel, even beyond this installation, be sure to have run this command beforehand)
4. `bazel build //...`: this will build the whole library and test cases (specific test cases or specify individual modules can be built with e.g `bazel build //modules/world/tests:world_test`, see the documentation of Bazel for more insights).
5. `bazel test //...`: this will run all specified tests (individual tests can be executed using, e.g. for the C++ tests `//modules/world/tests:world_test` and for the python test cases `bazel test //python:name_of_module`, e.g `bazel test //python:importer_test`).
6. If you experience problems with Pybind, you might not have installed the header files and static libraries for python. You can install them via `sudo apt-get install python3-dev`
7. If the viewer stays empty, you probably have to install tkinter using `sudo apt-get install python3-tk`.

### 3.2.2 IDE

We recommend Visual Studio Code as IDE under Linux/MacOS and also provide launch and task files to work with Bark in VSCode.

## 3.3 Windows

**CURRENTLY NOT FULLY SUPPORTED! Please use Linux or MacOS!**

### 3.3.1 Setup

1. Install Bazel and Prerequisites for windows: https://docs.bazel.build/versions/master/install-windows.html
2. Add JAVA_HOME!! Even if we do not want to build Java code
3. Run Bazel: https://docs.bazel.build/versions/master/windows.html (BAZEL_VS and BAZEL_VC) set BAZEL_VS=C:\Program Files (x86)\Microsoft Visual Studio 14.0 set BAZEL_VC=C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC
4. cd into bark directory and use bazel
5. Check if setup worked by //tests/geometry:geometry_tests, bazel run //tests/geometry:geometry_tests

### 3.3.2 IDE

As pybind11 depends on the vscompiler, Visual Studio is the recommended IDE under Windows. An experimental setup for the C++ part of Bark is achieved with

1. Install lavender to generate Visual Studio Projects for Bazel.
2. Then run eg. python ..\lavender\generate.py //modules/geometry //tests/geometry:geometry_tests to generate a Visual Studio project.

The python interfacing under Windows is currently untested.

## 3.4 Generate Documentation

A local version of the Bark documentation can be generated using

```
cd docs
make html
```

This will create the documentation which can be accessed at `/docs/build/html/index.html`.

# CHAPTER 4

# Examples

In order to get started with BARK, we provide several examples that show the basic functionality. All examples are found within the `/examples`-directory.

## 4.1 Constant Velocity OpenDrive 8

This example shows how to place a single agent in the OpenDrive example map "standard crossing" and how to simulate it with a constant-velocity. To run this example use the command `bazel run //examples:od8_const_vel_one_agent` (make sure to be in the virtual environment!) which yields:

We go step-wise through the python code. First, a parameter-server for the simulation has to be defined. If we do not pass a parameter-file, default arguments are being used.

```
param_server = ParameterServer()
```

Next, the world object using the parameter-server is created.

```
world = World(param_server)
```

In order to generate a behavior, validate it and for it to be dynamically feasible we define the models as *described*.

```
behavior_model = BehaviorConstantVelocity(param_server)
execution_model = ExecutionModelInterpolate(param_server)
dynamic_model = SingleTrackModel()
```

The map is specified using:

```
xodr_parser = XodrParser("modules/runtime/tests/data/Crossing8Course.xodr")
```

Furthermore, the `XodrParser` generates the `OpenDriveMap` and `Roadgraph`. These generated entities are then passed to the `MapInterface`.

```
map_interface = MapInterface()
map_interface.set_open_drive_map(xodr_parser.map)
map_interface.set_roadgraph(xodr_parser.roadgraph)
world.set_map(map_interface)
```

Next, our environment needs agents! Therefore, we initiate our agent with a vehicle-shape, a new parameter and an initial-state `x = [t, x, y, theta, v]`. Furthermore, we set the previously initiated models (behavior, execution and dynamic) to the agent.

```
agent_2d_shape = CarLimousine()
init_state = np.array([0, -11, -8, 3.14*3.0/4.0, 50/3.6])
agent_params = param_server.addChild("agent1")
agent = Agent(init_state, behavior_model, dynamic_model, execution_model, agent_2d_
↪shape, agent_params, 2, map_interface)
world.add_agent(agent)
```

In order to have insights on how our agent act we can specify a viewer. Here we choose a simple, real-time 2D-viewer (PygameViewer). If required, the MatplotlibViewer can be used in order to generate publication-ready figures.

```
viewer = PygameViewer(params=param_server, x_range=[-20, 20], y_range=[-200, 20],␣
↪follow_agent_id=agent.id)
```

Finally, to run the simulation use the following lines:

```
sim_step_time = param_server["simulation"]["step_time", "Gives the amount of time in␣
↪which one behavior planning call has to produce the result", 1]
sim_real_time_factor = param_server["simulation"]["real_time_factor", "How much␣
↪faster than real-time, simulation shall be played", 1]

for _ in range(0, 30):
    world.step(sim_step_time)
    viewer.drawWorld(world)
    viewer.show(block=False)
    time.sleep(sim_step_time/sim_real_time_factor)
```

In order to make the experiment reproducible, use:

```
param_server.save(os.path.join(os.path.dirname(os.path.abspath(__file__)), "params",
↪"od8_const_vel_one_agent_written.json"))
```

CHAPTER 5

# Debugging

## 5.1 Debugging with VSCode

### 5.1.1 Debugging C++ Code

Use build bazel in debug mode

```
bazel build //... --compilation_mode=dbg
```

Add the debug configuration in `.vscode/launch.json` in Visual Studio Code and then launch the debugger (F5 in current file)

```
{
    "name": "(gdb) Launch",
    "type": "cppdbg",

    "request": "launch",
    "program": "${workspaceFolder}/bazel-bin/{path-to-executable-file}",
    "args": [],
    "stopAtEntry": true,
    "cwd": "${workspaceFolder}",
    "environment": [],
    "externalConsole": true,
    "MIMode": "gdb",
    "setupCommands": [
        {
            "description": "Enable pretty-printing for gdb",
            "text": "-enable-pretty-printing",
            "ignoreFailures": true
        }
    ]
}
```

Set breakpoints and debug

### 5.1.2 Debug Python Code

```
{
    "name": "Python: Current File",
    "type": "python",
    "request": "launch",
    "program": "${file}",
    "console": "integratedTerminal",
    "env": {
        "PYTHONPATH": "${workspaceFolder}/bazel-bin/examples/od8_const_vel_one_agent.
↪runfiles/__main__/python:${workspaceFolder}/bazel-bin/examples/od8_const_vel_one_
↪agent.runfiles/__main__"
  }
```

The path `examples/od8_const_vel_one_agent.runfiles` needs to be changed if another python file should be debugged. Furthermore, make sure to be in the main executable file when launching the debugger (F5).

### 5.1.3 Debug from Python into C++ Extension

We run two Debuggers in parallel. First, check if the shared object "bark.so" contains debug symbols with e.g. `readelf --debug-dump=decodedline bark.so` which should print out line numbers and source file name for each instruction. If not use `bazel build //path_to_python_binary --compilation_mode=dbg` to build all dependencies with debug symbols. Then, add the following launch configuration and adapt the path in "additionalSOLibSearchPath":

```
{
            "name": "(gdb) Attach",
            "type": "cppdbg",
            "request": "attach",
            "program": "${workspaceFolder}/python/venv/bin/python3",
            "cwd" : "${workspaceFolder}",
            "additionalSOLibSearchPath":"${workspaceFolder}/bazel-bin/examples/od8_
↪const_vel_two_agent.runfiles/__main__/python",
            "processId": "${command:pickProcess}",
            "MIMode": "gdb",
            "sourceFileMap" : {"/proc/self/cwd/": "${workspaceFolder}"},
            }
```

For debugging;

1. Add a breakpoint in the python file you want to debug, somewhere before an interesting code section, run the launch configuration "Python: Current File" (see before) and wait until the breakpoint is reached.
2. Run the "(gdb) Attach" launch configuration, select the python interpreter whose path contains "/.vscode/". You will be promted to enter your user password.
3. Set breakpoints in the C++ Files
4. The python debugger is currently stopped at a break point. Switch back from the debugger "(gdb) Attach" to the other debugger "Python: Current File" and press F5 (Continue). Now, vscode automatically jumps between the two debuggers between python and c++ code.

### 5.1.4 Memory Checking

Use Valgrind to profile the code in order to find memory leaks. Valgrind can be installed using apt-get.

1. Build the target with debug symbols, i.e. `bazel test //modules/world/tests:py_map_interface_tests --compilation_mode=dbg`
2. Profile via `valgrind --track-origins=yes --keep-stacktraces=alloc-and-free --leak-check=full ./bazel-bin/modules/world/tests/map_interface_test`. There are a lot of options, check out Valgrind's documentation!

CHAPTER 6

# Profiling

## 6.1 Profiling using Easy Profiler

### 6.1.1 Step 1: Install Easy Profiler

install Qt5 (Paket qt5-default), see https://wiki.ubuntuusers.de/Qt/

clone https://github.com/yse/easy_profiler

build easy profiler using cmake (following the instructions in the repo)

install easy profiler using `make install`

Add to ld_path using `ldconfig`

### 6.1.2 Step 2: Prepare BARK Project

make sure to have `build:easy_profiler --linkopt='-L/usr/local/lib/ -leasy_profiler' --copt='-DBUILD_WITH_EASY_PROFILER'` in your bazel rc file

include easy_profiler using `\#include <easy/profiler.h>`

in every function you want to profile, write `EASY_FUNCTION();` at the beginning

### 6.1.3 Step 3: Run BARK

`bazel run --config=easy_profiler <some example target>`

make sure to only run a short example, otherwise the profiling dump will get too big

Profiling Dump should now be in `/tmp/<some example>.prof`

### 6.1.4 Step 4: Open Dump with Easy Profiler

go to the build directory of easy_profiler and run `bin/profiler_gui`

within the gui, open the dump

CHAPTER 7

# Coding Guidelines

We use the Google Style Guides for Python and C++ as a reference.

## 7.1 Coding Guidelines C++ Code

For C++ code, we use cpplint. It is installed automatically within your virtual environment. However, to use it in VS Code, we use the VS Code Extension cppline (Developer: mine, version 0.1.3). You can install it in the market place. When installed, rightclick on the extension and select "Configure Extension Settings". You now need to define the path to cpplint as local user.

```
Cpplint: Cpplint Path
The path to the cpplint executable. If not set, the default location will be used.
/(YOUR LOCAL PATH FOR BARK)/bark/python/venv/bin/cpplint
```

## 7.2 Coding Guidelines Python

Pylint and autopep8 are installed automatically to your virtual environment. When sourced in your VS Code terminal, you should be able to only click "Format Document". In case it asks for formatting guide, select pep8. However, this should come automatically with .vscode/settings.json.

CHAPTER 8

# Models

Each agent contains individual instances of the following model-components:

1. **Behavior-Model** Creates a sequence of desired states the agent should follow.
2. **Execution-Model** Validates the state-sequence generated by the behavior-model. For example, in the case of a vehicle, it might check the dynamical feasibility.
3. **Dynamic-Model** Can be used by all models in order to plan and to validate in a dynamically feasible way.

## 8.1 Behavior-Model

The `BehaviorModel` is a base-class from which different behavior-models are derived. The `Plan`-function plans a behavior for an agent over a given time-horizon (current time + `delta_time`).

```
class BehaviorModel : public modules::commons::BaseType {
    virtual Trajectory Plan(world::objects::AgentId agent_id,
                            float delta_time,
                            const world::ObservedWorld& observed_world) = 0;
}
```

The behavior-model has only access to the observed world as described *here*.

### 8.1.1 BehaviorConstantVelocity

The constant-velocity model interpolates an agent along a set route with constant velocity.

```
class BehaviorConstantVelocity : public BehaviorModel {
 public:
  explicit BehaviorConstantVelocity(commons::Params *params) :
    BehaviorModel(params) {}

  virtual ~BehaviorConstantVelocity() {}
```

(continues on next page)

(continued from previous page)

```
  Trajectory Plan(AgentId agent_id,
                  float delta_time,
                  const ObservedWorld& observed_world);

  virtual BehaviorModel *Clone() const;
};
```

## 8.2 Execution Model

The execution-model validates or makes the generated behavior dynamically feasible. However, it is also possible to skip this layer using the `ExecutionModelInterpolate` model.

```
class ExecutionModel : public commons::BaseType {
 public:
  explicit ExecutionModel(modules::commons::Params *params) :
    BaseType(params),
    last_trajectory_() {}

  virtual ~ExecutionModel() {}

  Trajectory get_last_trajectory() { return last_trajectory_; }

  void set_last_trajectory(const Trajectory& trajectory) {
    last_trajectory_ = trajectory;
  }

  virtual Trajectory Execute(const float& new_world_time,
                             const Trajectory& trajectory,
                             const DynamicModelPtr dynamic_model,
                             const State current_state) = 0;

 private:
  Trajectory last_trajectory_;
};
```

### 8.2.1 ExecutionModelInterpolate

This model does not modify the behavior generated by the `BehaviorModel`. This can be useful for applications, such as reinforcement learning.

### 8.2.2 ExecutionModelMpc

The Model Predictive Control (MPC) optimizes the original behavior producing a dynamically feasible motion. The MPC is implemented utilizing the Ceres-solver library.

```
class ExecutionModelMpc : public ExecutionModel {
 public:
  explicit ExecutionModelMpc(commons::Params *params);

  ~ExecutionModelMpc() {}
```

(continues on next page)

(continued from previous page)

```
  Matrix<double, Dynamic, Dynamic> get_last_weights() {
    return last_weights_;
  }

  Trajectory get_last_desired_states() { return last_desired_states_; }

  void set_last_weights(const Matrix<double, Dynamic, Dynamic> &weights) {
    last_weights_ = weights;
  }

  void set_last_desired_states(const Trajectory &desired_states) {
    last_desired_states_ = desired_states;
  }

  virtual Trajectory Execute(const float &new_world_time,
                             const Trajectory &trajectory,
                             const DynamicModelPtr dynamic_model,
                             const State current_state);

  Trajectory Optimize(std::vector<double*> parameter_block,
                      const Trajectory &discrete_behavior,
                      const Matrix<double, Dynamic, Dynamic> weights_desired_states);

 private:
  execution::OptimizationSettings optimization_settings_;
  Matrix<double, Dynamic, Dynamic> last_weights_;
  Trajectory last_desired_states_;
};
```

## 8.3 Dynamic Model

The `DynamicModel`-class implements the base-class from which dynamic models are derived. The `StateSpaceModel`-function allows for a flexible implementation of a variety of linear and non-linear dynamic state-space models.

```
class DynamicModel {
 public:
  DynamicModel() {}

  virtual State StateSpaceModel(const State &x, const Input &u) const = 0;

  virtual DynamicModel *Clone() const = 0;
};
```

### 8.3.1 SingleTrackModel

The `SingleTrackModel`-class represents a simplified vehicle-model.

```
  public:
    SingleTrackModel() {}
    virtual ~SingleTrackModel() {}
```

(continues on next page)

(continued from previous page)

```
  State StateSpaceModel(const State &x, const Input &u) const;
  DynamicModel *Clone() const {
    return new SingleTrackModel(*this);
  }
```

The single-track model uses the following equations:

$L_f$ : wheel-base

The following equations present the model

$$\text{input vector: } \mathbf{u} = \begin{pmatrix} u_0 \\ u_1 \end{pmatrix} \begin{matrix} \text{: acceleration} \\ \text{: steering angle} \end{matrix}$$

$$\text{state vector: } \mathbf{x} = \begin{pmatrix} t \\ x \\ y \\ \theta \\ v \end{pmatrix} \begin{matrix} \text{: time} \\ \text{: x-position} \\ \text{: y-position} \\ \text{: vehicle-angle} \\ \text{: velocity} \end{matrix}$$

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}) = \begin{pmatrix} 1 \\ v \cdot cos(\theta) \\ v \cdot sin(\theta) \\ v \cdot \frac{tan(u_1)}{L_f} \\ u_0 \end{pmatrix}$$

# CHAPTER 9

# World

The BARK-world models a simultaneous-move game where all agents generate a behavior given a cloned world state at a specific time. All agents are moved according to their specified behavior models for a fixed time period `delta_time` and the validity of the resulting world state is checked. In principle, the `World` class is defined as

```
class World {
    public:
        void Step(float delta_time);
    private:
        MapInterface map_interface_;
        AgentMap agents_;
        ObjectMap objects_;
        double world_time_;

}
```

The most important functions in the `World` class are listed below.

> void **`Step`**(float *delta_time*)
>
> The *Step* function loops through all agents, creates a cloned world and then calls the *Move* function of each agent. The Move function in combination with the Step function integrates the main functionality of BARK.
>
> **`inline`** World *`World::Clone`**`()`** **`const`**
>
> Returns a cloned world of the current world.

## 9.1 ObservedWorld

The `ObservedWorld` is derived from the (protected) `World`. Furthermore, it offers additional functionality in order to increase its usability in the planning modules. In the future, perturbations and other effects could be modeled. For example, the ego agent only having a partial world-view.

```
class ObservedWorld : protected World {
 public:
    ObservedWorld(const World& world, const AgentId& ego_agent_id) :
      World(world),
      ego_agent_id_(ego_agent_id) {}
    ~ObservedWorld() {}
    double get_world_time() const { return World::get_world_time(); }
    const LocalMap& get_local_map() const {
      return *World::get_agent(ego_agent_id_)->get_local_map();
    }
    AgentPtr get_ego_agent() const {
      return World::get_agent(ego_agent_id_);
    }
    MapInterfacePtr get_map() const { return World::get_map(); }
    State current_ego_state() const {
      return World::get_agents()[ego_agent_id_]->get_current_state();
    }
    Point2d get_ego_point() const {
      State ego_state = current_ego_state();
      return Point2d(ego_state(X_POSITION), ego_state(Y_POSITION));
    }

 private:
    AgentId ego_agent_id_;
};
```

## 9.2 Agents & Objects

All objects are static with a fixed state and 2D geometry. An agent is a dynamic object functioning as a container for the behavioral-, execution- and dynamic-model as described *here*. The agent class provides a `Move` function taking care of the necessary calls to generate a behavior and to check its feasibility. The agent tracks its movements in the `state_action_history_`. The goal of an agent is initially specified during instantiation using the `goal_lane_id`. Moreover, the `World` class manages all objects and agents based on IDs.

```
        class Agent {
        public:
                 void Move(const float &dt, const ObservedWorld &observed_world);
        private:
                BehaviorModel behavior_model_;
                ExecutionModel execution_model_;
                DynamicModel        dynamic_model_;

                StateActionHistory state_action_history_;
        }
```

void **Move**(**const** float *&dt*, **const** ObservedWorld *&observed_world*)

The function calls the above-described models sequentially and updates the *state_action_history_*.

## 9.3 OpenDriveMap

The `OpenDriveMap` class implements the specifications provided by the OpenDRIVE 1.4 Format. This allows an easy parsing and integration of maps available in the OpenDrive format. Furthermore, it is additionally used as a

data-container to store the map in memory.

## 9.4 MapInterface

Provides an interface to the `OpenDriveMap` in order to speed up the search and it additionally also implements a routing functionality. Therefore, it utilizes an `R-Tree` in order to find the nearest map objects, such as lanes. Furthermore, all lanes are being saved and connected in a graph.

```
class MapInterface {
        public:
                bool interface_from_opendrive(const OpenDriveMapPtr& open_drive_map);
                bool FindNearestLanes(const modules::geometry::Point2d& point,
                                      const unsigned& num_lanes,
                                      std::vector<opendrive::LanePtr>& lanes);
                std::pair< std::vector<LanePtr>, std::vector<LanePtr> >
                        ComputeLaneBoundariesHorizon(const LaneId& startid, const␣
↪LaneId& goalid);

        private:
               OpenDriveMapPtr open_drive_map_;
                  RoadgraphPtr roadgraph_;
                  rtree_lane rtree_lane_;
}
```

std::pair<std::vector<LanePtr>, std::vector<LanePtr>> **ComputeLaneBoundariesHorizon**(**const** LaneId *&startid*, **const** LaneId *&goalid*)

Generates a route using boost-graph (the LaneGraph) and returns the left-, right-boundary as well as the centerline.

bool **FindNearestLanes**(**const** modules::geometry::Point2d *&point*, **const** unsigned *&num_lanes*, std::vector<opendrive::LanePtr> *&lanes*)

A function that returns the nearest lanes for a given point.

Based on the Roadgraph-interface, the LocalMap searches a valid route for an agent given its `goal_lane_id`. It then and concatenates the lane geometry to 2D lines which can easily be used in the behavior generation models.

```
class LocalMap {
        public:
                bool generate(Point2d point,
                              LaneId goal_lane_id,
                              double horizon = numeric_double_limits::max());

                Line get_inner_line() const { return inner_line_; }
                Line get_outer_line() const { return outer_line_; }
                Line get_center_line() const { return center_line_; }
        private:
                MapInterfacePtr map_interface_;

}
```

CHAPTER 10

# Runtime

The runtime module implements the actual simulation in Python. Currently, it uses a simple step-based loop:

```
class Runtime:
    def __init__(self, world, step_time, viewer):
        self.world = world
        self.step_time = step_time
        self.viewer = viewer

    def run(self, steps):
        for step_count in trange(steps, leave=True):
            self.world.step(self.step_time)
            self.viewer.drawWorld(self.world)
            sleep(...)
```

In the future, a fully featured simulation with a benchmarking suite is planned.

## 10.1 Viewer

A common viewer interface allows easy extension of visualization capabilities of Bark. Currently, we provide a real-time capable, 2D visualization based on PyGame (PygameViewer) and for single-scene shots one based on Matplotlib (MatplotlibViewer).

# CHAPTER 11

# Common

Commonly used functions and classes in BARK.

## 11.1 Geometry

BARK comes with a geometry library allowing manipulations and calculations with 2D-Point-, Line- and Polygon-objects. It wraps the `boost::geometry` implementations in order to provide higher level geometric manipulations, such as searching for the nearest point on a linestring and returning the corresponding `s`-value.

## 11.2 BaseObject

A common base class for all BARK classes provides common functionality. Currently, it contains the global `ParameterServer` instance. In the future, this class can be extended with loggers and more modules.

```
class BaseType {
 public:
  explicit BaseType(Params* params) : params_(params) {}
  ~BaseType() {}

  Params* get_params() const { return params_;}

 private:
  Params* params_;  // do not own
};
```

### 11.2.1 ParameterServer

The ParameterServer is shared within the whole runtime. Its abstract implementation is reimplemented in Python. Child nodes can be added by using the `AddChild`-function.

```
class Params {
 public:
  Params() {}

  virtual ~Params() {}

  // get and set parameters as in python
  virtual bool get_bool(const std::string &param_name,
                        const std::string &description,
                        const bool &default_value) = 0;

  virtual float get_real(const std::string &param_name,
                         const std::string &description,
                         const float &default_value) = 0;

  virtual int get_int(const std::string &param_name,
                      const std::string &description,
                      const int &default_value) = 0;

  // not used atm
  virtual void set_bool(const std::string &param_name, const bool &value) = 0;
  virtual void set_real(const std::string &param_name, const float &value) = 0;
  virtual void set_int(const std::string &param_name, const int &value) = 0;

  virtual int operator[](const std::string &param_name) = 0;

  virtual Params* AddChild(const std::string &name) = 0;
};
```

CHAPTER 12

# Indices and tables

- genindex
- modindex
- search

# Index