BARK Documentation

Release 0.2

fortiss GmbH

Oct 27, 2020

Contents

Semantic simulation for interaction-aware multi agent planning.

CHAPTER 1

About BARK

The core focus of BARK is to develop and benchmark behavior models for autonomous agents – thus, **B**ehavior Benchm**ARK**. BARK offers a behavior model-centric simulation framework that enables fast-prototyping and the development of behavior models. Behavior models can easily be integrated — either using Python or C++. Various behavior models are available ranging from machine learning to conventional approaches.

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 several 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 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.

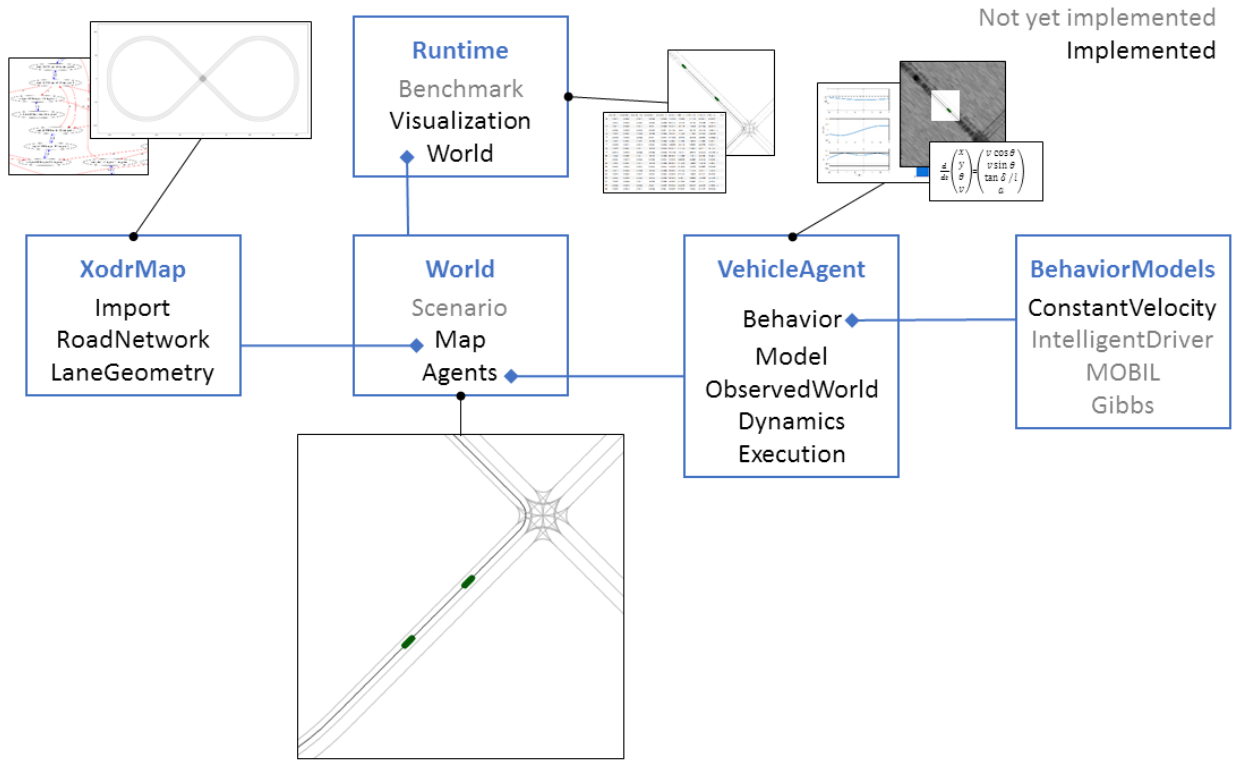
For more information have a look at our latest paper: BARK: Open Behavior Benchmarking in Multi-Agent Environments .

1.2 Approach

BARK is a multi-agent simulation tailored towards the use case of autonomous systems, with a special focus on autonomous driving. Each agent is controlled by a behavior specification in the form of a behavior 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 behavior 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

BARK has a modular and exchangeable architecture that is composed of modules. Core modules of BARK:



CHAPTER 2

Changelog

2.1 v0.1 (March 1, 2019)

- Initial release

2.2 v0.2 (May 26, 2020)

- Road- and LaneCorridors
- Rule-based behavior models and Mobil
- Intersections and merging scenarios
- BenchmarkRunner and Evaluator
- Dataset replay agents
- Novel and upgraded scenario generation
- Many improvements and bug fixes

CHAPTER 3

How to Install BARK

This section describes the prerequisites and installation-steps of BARK.

3.1 Prerequisites

- Bazel > 3(requires Java)
- Python3.7 (`sudo apt-get install python3.7 python3.7-dev python3.7-tk`)
- Virtual Env (`pip3 install virtualenv==16.7.8`) (note that the newest version does not seem to link the Python.h)
- gcc7 (needs to be set as the default compiler)
- Visual Studio Code

3.1.1 Using ad-rss-lib

- sqlite 3 (`sudo apt-get install libsqlite3-dev sqlite3`)

3.2 Install using pip

- pip3 install bark-simulator

3.3 Setup on Linux

Optional: We recommend to use Anaconda. This way, you can create a clean python environment. After installation, create a conda env `conda create --name bark_python_env python=3.7` and follow with activating it: `conda activate bark_python_env`. You can now proceed with the following:

1. Use `git clone https://github.com/bark-simulator/bark.git` or download the repository from this page.
2. Run `bash install.sh`: creates a virtual environment (located in python/venv) and installs all python packages
3. Run `source dev_into.sh`: activates the virtual environment (make sure to run this before bazel)
4. Use `bazel test //...` to validate that BARK is working.
5. Finally, try one of the examples provided in BARK by running `bazel run //examples:merging`.

3.4 Setup on MacOS

1. Install pyenv: `brew install pyenv`.
2. Install a newer version of tcl-tk: `brew upgrade tcl-tk`.
3. Run `pyenv install python3.7-dev`. If you run into trouble with TKInter have a look here.
4. Set this as your global Python version: `pyenv global 3.7-dev`.
5. Also add this Python version to your `~/.zshrc` by adding `eval "$(pyenv init -)"`.
6. Install an older version of the virtualenv package by running: `pip install virtualenv==16.7.8`
7. In order to set TKAgg as backend have a look here.
8. Modify the file `install.sh` by using `virtualenv -p python ./python/venv` instead as python is now the pyenv version.
9. Now you can follow the same steps as when using Linux.

3.5 Build Pip package

- Install twine using python3 -m pip install –user –upgrade twine
- Run script bash package.sh to build code, package and upload to pypi

3.6 Frequently Asked Questions (FAQs)

3.6.1 Python.h not found

Make sure that there is a 'Python.h' file in the `python/venv` folder.

3.6.2 GCC: Interal Compiler Error: Killed (program cc1plus)

You might be running out of memory during the bazel build. Try limiting the memory available to bark via `bazel build //... --local_ram_resources=HOST_RAM*.4` (or any other build or test call).

3.6.3 Feel free to add your questions here or asks us directly by submitting an issue!

CHAPTER 4

Examples

To get started with BARK, we provide several examples that show the basic functionality. All examples are found in the `/examples`-directory of BARK.

4.1 Merging Example

In this example, we show the basic functionality of BARK using a merging scenario. It can be ran using: `bazel run //examples:merging`.

BARK uses a `ParameterServer()` that stores all parameters of the simulation. We can set parameters globally:

```
param_server["BehaviorIDMClassic"]["DesiredVelocity"] = 10.
```

We define the scenario using a scenario generation module of BARK. In the following example, both lanes of the merging scenario are defined with a controlled agent on the right lane.

```
# configure both lanes
left_lane = CustomLaneCorridorConfig(params=param_server,
                                     lane_corridor_id=0,
                                     road_ids=[0, 1],
                                     behavior_model=BehaviorMobilRuleBased(param_
↪server),
                                     s_min=0.,
                                     s_max=50.)
right_lane = CustomLaneCorridorConfig(params=param_server,
                                      lane_corridor_id=1,
                                      road_ids=[0, 1],
                                      controlled_ids=True,
                                      behavior_model=BehaviorMobilRuleBased(param_
↪server),
                                      s_min=0.,
```

(continues on next page)

(continued from previous page)

```
                                        s_max=20.)

scenarios = \
  ConfigWithEase(num_scenarios=3,
                 map_file_name="bark/runtime/tests/data/DR_DEU_Merging_MT_v01_shifted.
↪xodr",
                 random_seed=0,
                 params=param_server,
                 lane_corridor_configs=[left_lane, right_lane])
```

We then define the viewer and runtime in order to run and visualize the scenarios:

```
viewer = MPViewer(params=param_server,
                  x_range=[-35, 35],
                  y_range=[-35, 35],
                  follow_agent_id=True)
env = Runtime(step_time=0.2,
              viewer=viewer,
              scenario_generator=scenarios,
              render=True)
```

Running scenarios can now be easily done as follows:

```
# run 3 scenarios
for _ in range(0, 3):
  env.reset()
  for step in range(0, 90):
    env.step()
    time.sleep(sim_step_time/sim_real_time_factor)
```

However, BARK also provides a `BenchmarkRunner` that runs scenarios automatically and benchmarks the performance of behavior models.

4.2 Other Examples

The other examples can be run in a similar fashion using:

- `bazel run //examples:highway`: Two-lane highway example.
- `bazel run //examples:intersection`: Three way intersection.
- `bazel run //examples:interaction_dataset`: Dataset replay.
- `bazel run //examples:benchmark_database`: Benchmarks behaviors using a scenario database.

CHAPTER 5

Models

Every agent in BARK has three models:

1. **Behavior Model** A model that generates the behavior (e.g. trajectory) of the agent.
2. **Execution Model** Validates the trajectory generated by the behavior-model (e.g. if it is dynamically feasible).
3. **Dynamic Model** Can be used by all models in order to plan and to validate the motion of an agent.

5.1 Behavior Models

In BARK all behavior models are derived from the `BehaviorModel` base-class. This base class defines the interface that all behavior models need to implement.

Outline of the `BehaviorModel` base-class:

```
class BehaviorModel : public bark::commons::BaseType {
 public:
  explicit BehaviorModel(const commons::ParamsPtr& params,
                         BehaviorStatus status)
      : commons::BaseType(params),
        last_trajectory_(),
        last_action_(),
        behavior_status_(status) {}
  ...
  virtual Trajectory Plan(float min_planning_time,
                          const ObservedWorld& observed_world) = 0;
 private:
  dynamic::Trajectory last_trajectory_;
  Action last_action_;
  BehaviorStatus behavior_status_;
}
```

The `Plan` function returns a behavior for an agent for a given time-horizon (current world time + `delta_time`). Each derived class implements its own `Plan` function. The behavior models plan the motion using the `ObservedWorld` as described *here*.

Self-contained *behavior models* in BARK:

- BehaviorConstantAcceleration: Interpolates on a line with const. velocity.
- BehaviorIDMClassic: Interpolates on a line and uses the basic IDM equations.
- BehaviorIDMLaneTracking: Follows a line using a steering function for the single-track model and the basic IDM equations.
- BehaviorMobil: Full Mobil implementation.
- BehaviorLaneChangeRuleBased: Rule-based agent that changes to the lane with the most free-space.
- BehaviorMobilRuleBased: Simple rule-based Mobil implementation.
- BehaviorIntersectionRuleBased: Simple rule-based intersection behavior.
- BehaviorStaticTrajectory: Data-driven replay of agents in BARK.

Behavior models that need the action to be set externally:

- BehaviorDynamicModel: Action has to be set externally. Then uses a dynamic model in the `step`-function.
- BehaviorMPMacroActions: Macro actions, such as follow the lane or change the lane to the left.
- BehaviorMPContinuousActions: Motion primitives with continuous action specification.

5.2 Execution Models

The execution model can be viewed as the controller of the simulation. For example, it can make the motion dynamically feasible or check its validity to enable more realistic simulations.

The class basic outline is given by:

```
class ExecutionModel : public commons::BaseType {
 public:
  explicit ExecutionModel(const bark::commons::ParamsPtr params) :
    BaseType(params),
    last_trajectory_() {}

  virtual ~ExecutionModel() {}

  Trajectory GetLastTrajectory() { return last_trajectory_; }

  void SetLastTrajectory(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_;
};
```

However, it is also possible to skip the execution model by using the `ExecutionModelInterpolate`. This model assumes that the motion of the behavior model can always be followed.

5.3 Dynamic Models

Dynamic models in BARK can be used for planning and validating dynamic motions of agents, e.g. state-space trajectories. All dynamic models are derived from the `DynamicModel` base-class.

The `DynamicModel` is given by:

```
class DynamicModel : public commons::BaseType {
 public:
  explicit DynamicModel(bark::commons::ParamsPtr params) :
    BaseType(params), input_size_(0) {}

  virtual ~DynamicModel() {}

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

  virtual std::shared_ptr<DynamicModel> Clone() const = 0;

  int input_size_;
};
```

Each dynamic model, such as the single-track model implements its own `StateSpaceModel` function. This allows for a flexible implementation of a vareity of linear and non-linear dynamic state-space models.

5.3.1 Single Track Model

By far the most used dynamic model in BARK, it the single-track model. It represents a simplified bicycle vehicle-model.

The `SingleTrackModel` class overloads the `StateSpaceModel` function and is given by:

```
class SingleTrackModel : public DynamicModel {
 public:
  explicit SingleTrackModel(const bark::commons::ParamsPtr& params) :
    DynamicModel(params),
      wheel_base_(params->GetReal("DynamicModel::wheel_base",
                                  "Wheel base of vehicle [m]", 2.7)),
      steering_angle_max_(params->GetReal(
          "DynamicModel::delta_max", "Maximum Steering Angle [rad]", 0.2)),
      lat_acceleration_max_(
          params->GetReal("DynamicModel::lat_acc_max",
                          "Maximum lateral acceleration [m/s^2]", 4.0)),
      lon_acceleration_max_(
          params->GetReal("DynamicModel::lon_acceleration_max",
                          "Maximum longitudinal acceleration", 4.0)),
      lon_acceleration_min_(
          params->GetReal("DynamicModel::lon_acceleration_min",
                          "Minimum longitudinal acceleration", -8.0)) {}
  virtual ~SingleTrackModel() {}

  State StateSpaceModel(const State& x, const Input& u) const {
    State tmp(static_cast<int>(StateDefinition::MIN_STATE_SIZE));
```

(continues on next page)

(continued from previous page)

```
    tmp << 1,
        x(StateDefinition::VEL_POSITION) *
            cos(x(StateDefinition::THETA_POSITION)),
        x(StateDefinition::VEL_POSITION) *
            sin(x(StateDefinition::THETA_POSITION)),
        x(StateDefinition::VEL_POSITION) * tan(u(1)) / wheel_base_, u(0);
    return tmp;
  }
  ...

 private:
  double wheel_base_;
  double steering_angle_max_;
  double lat_acceleration_max_;
  float lon_acceleration_max_;
  float lon_acceleration_min_;
};
```

The equations of the single-track model can be written as:

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 6

Behavior Models

Overview of all behavior models available in BARK.

6.1 Constant Velocity Model

The most basic model available in BARK, is the constant velocity behavior model. The `BehaviorConstantAcceleration` class interpolates an agent along a set line with a constant velocity. There are no collision checks and, thus, vehicles with different speeds can collide with each other.

6.2 Intelligent Driver Model

A more advanced behavior model in BARK is the intelligent driver model (IDM). The `BaseIDM` class is the base class of all IDM models. It provides all functions required for the calculation of the acceleration.

6.2.1 Classic

Similar to the constant velocity model, the `BehaviorIDMClassic` interpolates itself along a line. Additionally, it changes the acceleration based on the free-road- and interaction-term to model realistic lane-following behavior. Thus, if the initial conditions are feasible, the IDM normally does not cause any collisions.

6.2.2 Lane Tracking

The `BehaviorIDMLaneTracking` is similar to the classic IDM model but uses a dynamic model and steering function to follow a line.

6.3 Mobil Model

The Mobil model extends the IDM model even further. It additionally checks, whether a lane-change would be beneficial for the ego vehicle as well as for the surrounding vehicles. If the politeness parameter of the Mobil model is set to zero, it only checks how to proceed the fastest on a road (mult. lanes).

6.4 Rule-based Models

The rule-based models in BARK have been developed to allow more sophisticated behaviors, such as braking, changing lanes, and handling intersections. One of the core concepts of these models is a filter function that allows formulating lambda-functions to sort out lane-corridors. This filtering is versatile and e.g. can use the free-space on the other lane or more sophisticated methods.

6.4.1 Lane Change Model

There are two rule-based lane-changing models currently implemented in BARK: `BehaviorLaneChangeRuleBased` and `BehaviorMobilRuleBased`. The `BehaviorLaneChangeRuleBased` class checks the free-space on all available lanes and changes to the one having the most free space. The `BehaviorMobilRuleBased` class acts as the normal Mobil model, but only can change to filtered lanes (e.g. that have sufficient free-space).

6.4.2 Intersection Model

The intersection model can handle intersections of arbitrary shape. It is prediction- and rule-based. If an agent intersects the ego agent's `LaneCorridor` first, the ego agent has to brake. However, to avoid deadlocks (if both agents intersect at the same time), there is an additional right before left rule. This model is not based on any literature but has empirically shown to work well.

6.5 Behavior Dynamic Model

This model is an externally controlled model, e.g. by a neural-network and requires the action to be set. Once the action is set, the `Step` function of the BARK world can be called and the `BehaviorDynamicModel` will produce a trajectory using the set action. This model is e.g. used in BARK-ML.

6.6 Behavior Motion Primitives

In BARK there are macro and continuous motion primitives. The macro motion primitives have actions, such as follow the lane or change the lane to the left. The continous motion primitives use continuous action inputs and push these to a vector.

CHAPTER 7

World

The world in BARK contains all objects of the simulation. It is modeled as a simultaneous-move game where all agents act at the same time. Each agent moves according to its defined behavior, execution, and dynamic model. After all agents have been moved, the overall validity is then checked.

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 *MapInterface* contains the map, functionalities for routing, and simplified structures for the agents to plan in. The `AgentMap` contains all agents of the simulation and the `ObjectMap` all static objects. Finally, the `World` class also contains the simulation world time `world_time_`.

7.1 Observed World

In each simulation step, an agent in BARK gets passed an `ObservedWorld` that is derived from the current `World`. The agent then plans in this derived world and returns a trajectory. The `ObservedWorld` provides additional interfaces and allows to model further features, such as e.g. occlusions. Besides providing additional functionalities, it also defines and saves the ego agent id `ego_agent_id_`.

```
class ObservedWorld : public World {
 public:
  ObservedWorld(const WorldPtr& world, const AgentId& ego_agent_id) :
    World(world),
```

(continues on next page)

(continued from previous page)

```
    ego_agent_id_(ego_agent_id) {}
  ...
 private:
  AgentId ego_agent_id_;
};
```

7.2 Objects and Agents

In BARK objects are static and can be extended to dynamic agents. Objects in BARK have a position, a shape, and an ObjectId. The agent extends this by adding a behavior, execution, and dynamic model as described *here*. Additionally, the agent also has a `GoalDefinitionPtr` and `RoadCorridorPtr`.

The agent class is defined as follows:

```
class Agent : public Object {
 public:
  friend class World;

  Agent(const State& initial_state,
        const BehaviorModelPtr& behavior_model_ptr,
        const DynamicModelPtr& dynamic_model_ptr,
        const ExecutionModelPtr& execution_model,
        const geometry::Polygon& shape,
        const commons::ParamsPtr& params,
        const GoalDefinitionPtr& goal_definition = GoalDefinitionPtr(),
        const MapInterfacePtr& map_interface = MapInterfacePtr(),
        const geometry::Model3D& model_3d = geometry::Model3D());
  ...
 private:
  BehaviorModelPtr behavior_model_;
  DynamicModelPtr dynamic_model_;
  ExecutionModelPtr execution_model_;
  RoadCorridorPtr road_corridor_;
  StateActionHistory history_;
  uint32_t max_history_length_;
  GoalDefinitionPtr goal_definition_;
};
```

CHAPTER 8

MapInterface

The `MapInterface` class implements all map-related features for BARK. It stores the raw OpenDrive map, has a `RoadGraph` for routing, and provides convenient and easy-to-use classes for the agents - the `RoadCorridor` and the `LaneCorridor`.

The `MapInterface` class is implemented as follows:

```
class MapInterface {
 public:
    ...
 private:
  OpenDriveMapPtr open_drive_map_;
  RoadgraphPtr roadgraph_;
  rtree_lane rtree_lane_;
  std::pair<Point2d, Point2d> bounding_box_;
  std::map<std::size_t, RoadCorridorPtr> road_corridors_;
}
```

Additionally, the `MapInterface` also has an lane r-tree for more performant lane searching.

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. However, for better usability we encapsulate this specification using a `RoadGraph`, `RoadCorridors`, and `LaneCorridors`.

The basic structure of the `OpenDriveMap` map class:

```
class OpenDriveMap {
 public:
  OpenDriveMap() : roads_(), lanes_(), junctions_() {}
  ~OpenDriveMap() {}
  ...
 private:
  XodrRoads roads_;
  XodrLanes lanes_;
  Junctions junctions_;
}
```

8.1 RoadGraph

The `RoadGraph` contains all roads and lanes and their physical location in a graph structure. This enables easy routing functionality for an agent in BARK. The physical locations of the start and goal are sufficient in order to obtain a sequence of lanes or roads for the agent.

However, in order to store this information more efficiently and to increase usability, we additionally use `RoadCorridors` and `LaneCorridors`.

8.2 RoadCorridor

A `RoadCorridor` is composed out of a sequential sequence of roads that an agent in BARK can follow. It contains all the OpenDrive information as well as further information in the form of the `LaneCorridor`. The `RoadCorridor` provides an easy-to-use interface by providing functions that tell an agent the current lane and what it left or right lanes are.

The basic structure of the `RoadCorridor` map class:

```
struct RoadCorridor {
  ...
  Roads roads_;
  Polygon road_polygon_;
  std::vector<LaneCorridorPtr> unique_lane_corridors_;
  std::vector<XodrRoadId> road_ids_;
  std::map<LaneId, LaneCorridorPtr> lane_corridors_;
}
```

8.3 LaneCorridor

A `LaneCorridor` is continuously, sequentially concatenated lanes. It provides many utility functions, such as the distance to the end of the `LaneCorridor`, the merged lane polygon, the boundaries, and more.

```
struct LaneCorridor {
  ...
  std::map<float, LanePtr> lanes_;  // s_end, LanePtr
  Line center_line_;
  Polygon merged_polygon_;
  Line left_boundary_;
  Line right_boundary_;
}
```

CHAPTER 9

Runtime

The runtime module implements the actual simulation in Python. It provides a similar interface as the OpenAI Gym environments.

```
class Runtime(PyRuntime):
  def __init__(self,
               step_time,
               viewer,
               scenario_generator=None,
               render=False):
    self._step_time = step_time
    self._viewer = viewer
    self._scenario_generator = scenario_generator
    self._scenario_idx = None
    self._scenario = None
    self._render = render
    ...

  def reset(self, scenario=None):
    ...

  def step(self):
    ...

  def render(self):
    ...
```

The runtime has a scenario generator that fills in the `self._scenario` and ID of the current scenario `self._scenario_idx`.

9.1 Scenario

The scenario fully defines the initial state of the simulation, such as the agent's positions and models.

The outline of the `Scenario` class is given by:

```
class Scenario:
  def __init__(self,
               agent_list=None,
               eval_agent_ids=None,
               map_file_name=None,
               json_params=None,
               map_interface=None):
    self._agent_list = agent_list or []
    self._eval_agent_ids = eval_agent_ids or []
    self._map_file_name = map_file_name
    self._json_params = json_params
    self._map_interface = map_interface
  ...
```

It also specifies which agents should be evaluated using `self._eval_agent_ids`.

9.2 Scenario Generation

A scenario generation in BARK returns a list of scenarios of the type `Scenario`. These can be run by the BARK runtime or by the `BenchmarkRunner`.

Currently available scenario generators:

- `ConfigurableScenarioGeneration`: Sophisticated scenario generation providing conflict resolution.
- `UniformVehicleDistribution`: Samples the agents uniformly and their parameters.
- `ConfigWithEase`: Configure any scenario fast and with ease.
- `DeterministicScenarioGeneration`: Deterministic, reproducible scenario generation.

9.3 Benchmarking

BARK provides a `BenchmarkRunner` and `BenchmarkAnalyzer` to automatically run and verify the performance of novel behavior models.

9.4 Viewer

A common viewer interface allows easy extension of visualization capabilities of Bark.

Several viewer modules are currently available:

- `MPViewer`: Matplotlib viewer for scientific documentation.
- `Panda3dViewer`: 3D-Visualization.
- `PygameViewer`: Gym-like visualization of the BARK environment.

CHAPTER 10

Common

Commonly used classes and functions in BARK.

10.1 Geometry

BARK provides an easy-to-use geometry library that supports points, lines, and polygons for performant geometric calculations. By wrapping the `boost::geometry` state-of-the-art algorithms as well as high usability is provided. It implements all geometric functions, such as collision checks and distance calculations.

10.2 BaseObject

All objects in BARK share a common base class, the `BaseType`. It provides functionalities and members that are shared and used in all classes. For example, it contains the global `ParameterServer` instance that holds all parameters.

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

  ParamPtr GetParams() const { return params_;}
  ...
 private:
  ParamPtr params_;
};
```

10.3 ParameterServer

The `ParameterServer` is shared with all objects in BARK. 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 GetBool(const std::string &param_name,
                       const std::string &description,
                       const bool &default_value) = 0;

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

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

  // not used atm
  virtual void SetBool(const std::string &param_name, const bool &value) = 0;
  virtual void SetReal(const std::string &param_name, const float &value) = 0;
  virtual void SetInt(const std::string &param_name, const int &value) = 0;

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

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

CHAPTER 11

Debugging with VSCode

11.1 Debugging C++ Code

First, you need to build BARK in the debug mode using

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

Additionally, you need to modify the `.vscode/launch.json` in Visual Studio Code and then launch the debugger (F5 in the 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
    }
  ]
}
```

Now, you can set breakpoints and debug the c++ code.

11.2 Debugging Python Code

To debug python code, you need to add the following code in the `.vscode/launch.json`:

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

Make sure to be in the main executable file when launching the debugger (F5).

11.3 Debugging C++ and Python

Here, both debuggers need to be run in parallel. First, we need to build the "bark.so" in the debug mode by running `bazel build //... --compilation_mode=dbg`. You can also check if the "bark.so" contains debug symbols by running `readelf --debug-dump=decodedline bark.so`. Then, add the following launch configuration and adapt the path in the `.vscode/launch.json`:

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

Debugging process:

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.

11.4 Memory Checking

Use Valgrind to profile the code in order to find memory leaks.

1. Build the target with debug symbols, i.e. `bazel test //bark/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/{path-to-executable-file}`.

CHAPTER 12

Profiling using Easy Profiler

12.1 Step 1: Install Easy Profiler

Install Qt5 (Paket qt5-default), see https://wiki.ubuntuusers.de/Qt/. Clone https://github.com/yse/easy_profiler and build easy profiler. Then install easy profiler using `make install` and add to ld_path using `ldconfig`

12.2 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. Defining the functions `void profiler_startup()` and void `profiler_finish()`, for example in some utility function

```
void profiler_startup() {
  EASY_PROFILER_ENABLE;
//  profiler::startListen();
}

void profiler_finish() {
  auto blocks_written = profiler::dumpBlocksToFile("/tmp/<some example>.prof");
  LOG(INFO) << "Easy profiler blocks written: " << blocks_written;
}
```

and wrap them to Python. In the python runtime, you then have to call them before and after the code you want to profile.

12.3 Step 3: Run BARK

The run: `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`.

12.4 Step 4: Open Dump with Easy Profiler

Go to the build directory of easy_profiler and run `bin/profiler_gui`. You then can use the GUI to open the dump file.

CHAPTER 13

Coding Guidelines

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

13.1 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
```

13.2 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 14

Indices and Tables

- genindex
- modindex
- search