Autonomous Parking Scenario

Linux and L4, SS 2017, Group 03

Alexander Reisner reisner@tum.de
Alexander Weidinger alexander.weidinger@tum.de
David Werner david.werner@tum.de

2017-08-06

Contents

1 Introduction (Alexander Weidinger) 4

1.1 Project Overview 4

1.2 Sub Tasks 5

2 Speed Dreams 2 (Alexander Weidinger) 6

2.1 Proximity Sensor 6

2.1.1 SCRC Implementation 6

2.1.2 Our Implementation 7

2.2 Starting Grid 12

2.2.1 Idea 12

2.2.2 Implementation 12

2.3 Driving bots 12

2.3.1 human 13

2.3.2 usr 14

2.4 Simulation Coupler 15

3 S/A VM (Alexander Reisner) 16

3.1 Data exchange with Speed Dreams 2 16

3.1.1 Protobuf 16

3.1.2 TCP/IP 16

3.2 Our protocol 17

3.2.1 Deserialization 17

3.2.2 Serialization 17

3.2.3 Nagle's algorithm 17

3.2.4 Do not wait for answer of ECU 18

3.3 Data exchange with ECU 18

3.3.1 Subscribe 18

3.3.2 Publish 18

3.3.3 Callbacks 19

3.4 Starting the algorithm 19

3.4.1 Car information 19

3.4.2 Compute next step 19

3.4.3 Publish next step 20

4 ECU (David Werner) 21

4.1 Autonomous Parking 21

4.2 Concept of the Algorithm . . . 21
4.2.1 Motion of a vehicle . . . 22
4.2.2 Searching phase . . . 23
4.2.3 Calculation phase . . . 23
4.2.4 Controlling phase . . . 24
4.3 Implementation . . . 24
4.3.1 CarInformation . . . 24
4.3.2 Map . . . 25
4.3.3 Parking . . . 25
4.4 Problems . . . 28
4.4.1 Safety distances . . . 28
4.4.2 Dependance on unrealistic time estimations . . . 29
4.5 Improvements . . . 29
4.5.1 Rework of calculation phase . . . 29
4.5.2 Expanding the parked condition . . . 29
4.5.3 Implementing the original algorithm . . . 30
4.6 Conclusion . . . 30

1 Introduction (Alexander Weidinger)

This document is the final report, written by the students Alexander Reisner, Alexander Weidinger and David Werner, for the "Linux and L4" SS 2017 practical course at the Technical University of Munich.

The overall given task for the practical course was to implement an autonomous parking algorithm.

This parking algorithm is implemented as an user space application for a Pandaboard, running the Genode OS Framework in version 16.08 with the Fiasco.OC version r56 as kernel. We will refer to this compound as *ECU* from now on.

Our ECU is integrated in the hybrid simulator framework preexisting by the KIA4SM project. This hybrid simulator consists of a car racing simulator, Speed Dreams 2 (SD2), a simulation coupler (SimCoupler), a so called *S/A VM* which basically represents a multiplexing entity for each car in the simulation and a model car, representing the main car in the simulation.

Operating the model car is not part of this group's task but rather a task of the second team in this practical course. Due to the structure of the data exchange that was agreed on at the Level of the ECU and S/A VM, calculated actuator data is also received by the other team's software and can be used to control the model car.

1.1 Project Overview

Figure 1.1 shows the overview of the project and the information flow between all involved components. SD2 produces sensor information and sends them to the S/A VM. The S/A VM multiplexes the data and publishes all information to the ECUs. They are now able to compute control commands and publish them back to the S/A VM. The S/A VM now itself receives all control commands, bundles them in a single messages and sends them back to SD2. This behavior runs endlessly in a loop and gets executed each simulation step.

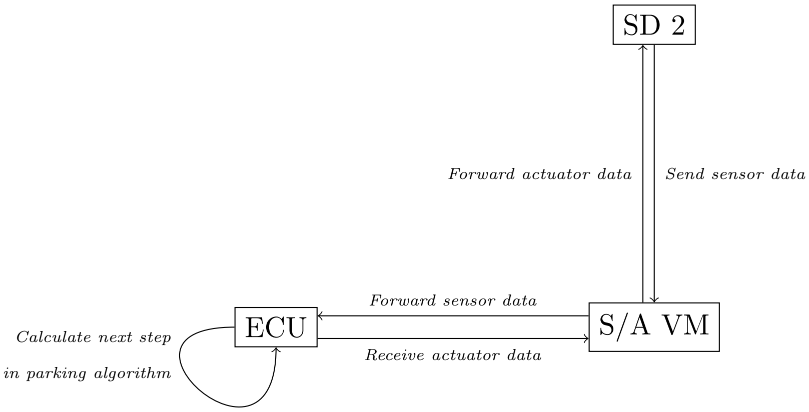

Figure 1.1: Overview of components

1.2 Sub Tasks

The project was split into sub tasks, assigned to each member of the group and also represents the parts written by each one in this report:

- **Alexander Reisner**
  - Design and Implementation of the S/A VM
  - Data exchange between S/A VM and SD2
  - Data exchange between S/A VM and ECU
  - Implementation of the connection interface (Mosquitto client) in the ECU
  - Final port of the Mosquitto library to the Genode OS Framework
- **Alexander Weidinger**
  - Extension of SD2 by a proximity sensor
  - Data exchange between SD2 and S/A VM
  - Extension of the human driver bot to enable autonomous driving
  - Extend SD2 by a parked car driver bot
  - Change starting grid of cars to allow reproducible testing results
- **David Werner**
  - Implement the autonomous parking algorithm
  - Optimize the algorithm and adapt it for usage in our scenario

2 Speed Dreams 2 (Alexander Weidinger)

Speed Dreams 2 is a fork of The Open Racing Car Simulator (TORCS) and was used as our main car simulation. Since the simulator was merely intended as a racing simulator, it only provides limited functionality in case of (virtual) sensors. Fortunately it can be easily expanded since the project itself is open source and more or less sufficiently documented for such tasks. Additionally there is a mailing list[1] and an online forum[2], where other people are willing to help in case of questions.

For our autonomous parking use case we needed to make some changes to SD2, which affect the proximity sensor itself, the starting position of the cars and finally changes to the driving bots.

2.1 Proximity Sensor

Since SD2 itself doesn't provide virtual proximity sensors, we either need to implement one ourselves or we make use of related projects. Luckily there exists the "Simulated Car Racing Championship 2015" (SCRC) [1], which extends the TORCS simulator by two new sensors. One being an "opponent" sensor, which measures the distance between the driving car and an opponent, furthermore there is a "focus" sensor, which measures the distance between the car and the track edge.

2.1.1 SCRC Implementation

The problem with the given implementation is, that the opponent sensor just measures the distance between two cars given the middle point of both cars. Given the example in figure 2.1, the distance between both cars should be relatively small, since they are more or less next to each other. But due to the implementation, the distance d_1 is over 2 meters and if they were directly next to each other, the distance between both cars would still be exactly 2 meters (given that each car has a width of 2 meters). Therefore the distance given by this sensor can be used to get a basic idea on where other cars are and how far they are away from oneself but are completely useless for our parking scenario. The red dashed lines in the figure additionally shows the way how distances to objects are sorted into multiple proximity "sensors", by calculating the angle between both cars and mapping it into a sensor. The implementation gives a 360° view by splitting it in intervals of 30°[3].

[1] `https://sourceforge.net/p/speed-dreams/mailman/`

[2] `https://community.speed-dreams.org/`

[3] Different values for the interval is also possible.

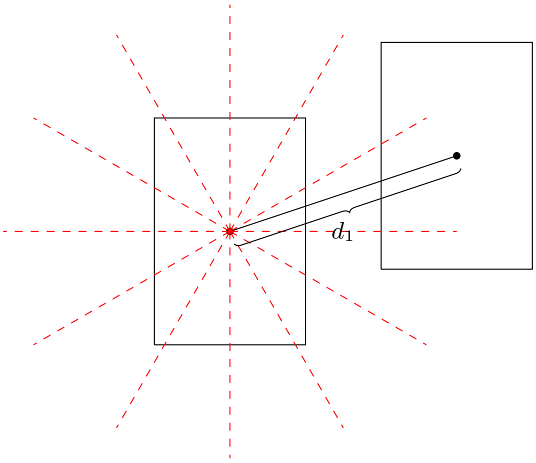

Figure 2.1: Proximity sensor implemented by the Simulated Car Racing Championship 2015

2.1.2 Our Implementation

Idea

We simplify the virtual proximity sensor to a mere straight line, with a given angle and position relative to the center of the car.

The opponent car can be cut down to four straight lines, acting as the four sides of a car.

In order to now calculate the shortest distance to an opponent car, the algorithm takes the following steps:

1. Iterate over all opponent cars:
    a) Calculate intersections between the straight line from the laser and the four straight lines from the sides of the opponent car.
    b) Iterate over all intersections:
        i. Test if point of intersection is within the domain of the respective side of the car.
        ii. Check if the intersection is ”in front” of the sensor.
    c) Save the potentially shortest distance to an opponent, if not already a shorter distance was found.
2. Accept the shortest distance as the current value of the proximity sensor.

Step 1(b)ii is necessary, since we are working with straight lines and not vectors. Otherwise it could happen, that objects on the other side of the car have to shortest distance to the sensor and therefore are accepted incorrectly as the current value of the sensor. A mockup of our implementation idea can be seen in figure 2.2. The distance d_2 would be the correct value for the proximity sensor, indicated by the dashed blue line.

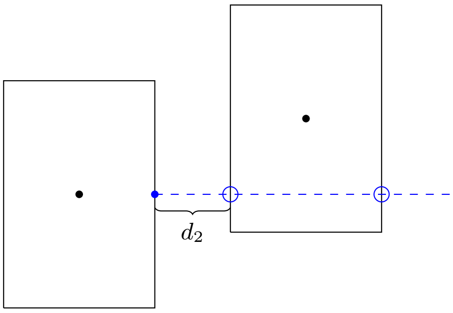

Figure 2.2: Mockup of our (laser) proximity sensor implementation

Implementation

Since a sensor position is described as a relative point to the center $(0, 0)$ of a car, we need to move the starting point of the sensor to the correct absolute position. The distance from the center point to the sensor is calculated and the x and y coordinates are added to the absolute position of the car, in regard to the yaw of the car (see listing 1).

```
/* position sensor at desired location */
double dis = distance(point{0, 0}, point{(*it).getMove_x(), (*it).getMove_y()}); // tota
double phi = atan2((*it).getMove_y(), (*it).getMove_x()); // angle in which direction we
sensorPosition = { myc->_pos_X + cos(myc->_yaw + phi) * dis,
        myc->_pos_Y + sin(myc->_yaw + phi) * dis
}; // calculate distance for x and y coordinates and add it to middle point
```

Listing 1: `src/libs/sensors/obstacleSensors.cpp`

To calculate the straigt line equation $m*x+t=y$ of the sensor, m must be calculated by adding the custom angle to the current yaw of the car using *tan*. t can be then obtained by simple inserting the x and y coordinates of the sensor. The associated code can be seen in listing 2.

```
m = tan(myc->_yaw + (*it).getAngle() * PI / 180); // add custom angle of sensor (in degr
...
double t = sensorPosition.y - m * sensorPosition.x;
```

Listing 2: `src/libs/sensors/obstacleSensors.cpp`

After calculating the straight line equation for the sensor, we need to create four additional ones, representing the four sides of the car. We calculate the slope m by using the yaw of the car, to get the left and right sides. For the front and back sides we do the same but use the yaw +90° for the rotation. These four straight lines are moved to the correct position, again relative to the center of the obstacle car (compare listing 3).

```
/* calculate slope of obstacle car + 90 degree turned */
double m_obst = tan(obstacleCar->_yaw);
double m_obst_90 = tan(obstacleCar->_yaw + PI/2);

/*
* corners:
*
* 1   front   0
*   +--------+
* l |        | r
* e |        | i
* f |        | g
* t |        | h
*   |        | t
*   +--------+
* 3   back    2
*/

/* build 4 straight lines (the 4 sides of the car) for obstacleCar */
double t_left = obstacleCar->_corner_y(1) - m_obst * obstacleCar->_corner_x(1);
double t_front = obstacleCar->_corner_y(1) - m_obst_90 * obstacleCar->_corner_x(1);
double t_right = obstacleCar->_corner_y(2) - m_obst * obstacleCar->_corner_x(2);
double t_back = obstacleCar->_corner_y(2) - m_obst_90 * obstacleCar->_corner_x(2);
```

Listing 3: `src/libs/sensors/obstacleSensors.cpp`

Now that all five straight lines are calculated, the next step is the computation of the intersections between the straight line of the sensor and the four straight lines, representing the sides of the obstacle car (see listing 4).

After obtaining the intersections, the domain is checked by verifying, that the intersection is between the two points restricting the side of the car. Additional to that, we need to make sure the obstacle isn't behind the sensor. For that we reuse our `is_between` function (which checks if a point is between two points) and adding a reference point, which basically is just an additional point ”behind” the sensor in a fixed distance (see listing 5).

Now that we can find the intersections between a sensor and an obstacle car, we do this in a loop for all sensors and all cars. That way we can determine the shortest

```
/* calculate intersections
*
* m_1 * x + t_1 = m_2 * x + t_2
* => m_1 * x - m_2 * x = t_2 - t_1
* => x * (m_1 - m_2) = t_2 - t_1
* => x = (t_2 - t_1) / (m_1 - m2)
*/
point i_left = { (t_left - t) / (m - m_obst), m * i_left.x + t };
point i_front = { (t_front - t) / (m - m_obst_90), m * i_front.x + t };
point i_right = { (t_right - t) / (m - m_obst), m * i_right.x + t };
point i_back = { (t_back - t) / (m - m_obst_90), m * i_back.x + t };
```

Listing 4: `src/libs/sensors/obstacleSensors.cpp`

```
point reference = { sensorPosition.x - 1 * cos(myc->_yaw - (*it).getAngle() * PI / 180),
        sensorPosition.y - 1 * sin(myc->_yaw - (*it).getAngle() * PI / 180) };
```

Listing 5: `src/libs/sensors/obstacleSensors.cpp`

distance overall for each sensor and set it as the correct distance value for the respective sensor.

Usage

As an example on how to use the newly created obstacleSensors library, one can take a look at the human driver in listing 6.

```
/* add laser obstacle sensors */
sens = new ObstacleSensors(curTrack, car);

/* (-car->_cimension_x/2, car->dimension_y/2)
 *      +-----L-----+
 *      B     0     F
 *      +-----R-----+ (car->_cimension_x/2, -car->dimension_y/2)
 */
/* car, angle, move_x, move_y, range */
sens->addSensor(car, 0, car->_dimension_x/2, 0, 20); // front
sens->addSensor(car, 90, car->priv.wheel[2].relPos.x, -car->_dimension_y/2, 20); // righ
sens->addSensor(car, 180, -car->_dimension_x/2, 0, 20); // back
```

Listing 6: `src/drivers/human/human.cpp`

The code adds three sensors, one in the front with an 0° angle and one in the back with an angle of 180°. Both sensors are moved relative to the center point of the car, according to the schematics in the comment. The last sensor (second sensor in the example code) is directed in an 90° angle and moved to the right side of the rear axle. As a side note, all three sensors have a maximum range of 20 meters, which can be changed with the last parameter.

Evaluation

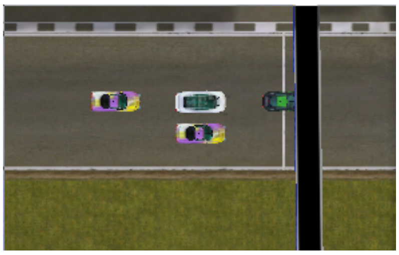

Figure 2.3: Example parking situation

Given the situation in figure 2.3 (we are the white car) and having one sensor in the front, one in the back and one to the right, we receive the following values for the sensors:

- **Front:** 3.306382 m
- **Back:** 3.400575 m
- **Right:** 1.010126 m

While with the implementation by the SCRC project, we receive different values:

- **Front:** 8.086382 m
- **Back:** 8.180575 m
- **Right:** 3.010126 m

This means, that our sensor implementation is much more precise and usable for the parking scenario.

Open points

Our main focus was getting the overall setup running, therefore a few open points are still left open and can be fixed in future versions.

The whole implementation is based on straight lines and therefore needs some workarounds to deal with directions. To overcome this issue it would make sense to change the implementation to vectors. This could also fix some potential glitches with wrong distance values, which can occur on collisions due to precision errors. Lastly the implementation only represents a laser proximity sensor in contrast to the implementation by the SCRC, which can detect or more precisely assigns opponents in angle ranges. One could work around this limitation by randomizing the direction of the laser sensor in a specific range of angles and afterwards taking the average of these values.

2.2 Starting Grid

In order to test the successful execution of our parking algorithm, we need to be able to get reproducible results in every run.

2.2.1 Idea

As we are not able to create some kind of run configuration files for SD2 and an implementation of such a system would take a lot of work, we made use of the starting grid in the initialization phase of a race.

2.2.2 Implementation

Normally the **.xml* file of a SD2 track specifies the number of rows, the cars are placed in at the start line. The distance between the cars is then automatically adjusted and calculated, taking in mind the overall width of the track, the course of the road, etc.

For the purpose of the parking scenario we hard coded the row value to one, as can be seen in listing 7.

```
#ifdef PARKING
rows = 1; // only use one row
#endif
```

Listing 7: `src/modules/racing/standardgame/raceinit.cpp`

We also made sure to place each car 8 meters further away from the start line by choosing a fix value in the code. Additionally to get the second and last car to either be 3 meters to the left or right, we adapted the position change to the right accordingly in the code (listing 8).

In order to ensure, that the changes don't interfere with future development, the changes only get used, if the code is compiled with the `PARKING` option enabled.

2.3 Driving bots

Further changes were needed for two of the bots in SD2, since we still want to be able to drive for ourselves, although we need to make sure that the car can move autonomously. Additionally we need parked cars for our proximity sensor to detect. We chose to make use of the *human* driver for the first use case and extend it by the autonomous driving. For the second use case we chose the *usr* driver to act as the parked car.

```
#ifdef PARKING
startpos = ReInfo->track->length - i * 8.0; // change distance between cars to 8m
#else
startpos = ReInfo->track->length - (d1 + (i / rows) * d2 + (i % rows) * d3);
#endif
tr = a + b * ((i % rows) + 1) / (rows + 1);
#ifdef PARKING
if (i == 1)
    tr -= 3; // move car 3 meters to the right
if (i == 3)
    tr += 3; // move car 3 meters to the left
#endif
```

Listing 8: `src/modules/racing/standardgame/raceinit.cpp`

2.3.1 human

The changes to the *human* bot consists of two parts. First add the data exchange between SD2 and the S/A VM and second include an autonomous driving logic.

The code for the data exchange is relatively trivial, since its a simple socket based connection, where SD2 acts as the server and globally listens for connections on port 9002[4]. The only really relevant part is, that one needs to disable Nagle's algorithm, by setting the `TCP_NODELAY` socket option, because we are creating a real time application and are in need of a low latency.[5] The message flow between the S/A VM and SD2 is always the same and based on an exchange between both applications. First SD2 collects all sensor information (proximity sensors, spin velocity of the wheels, etc.) and packs this data into a Protobuf *State* message.[6]

The Protobuf message is then serialized to a string, the length of this message gets extracted and both information are transmitted over the socket connection, as can be seen in listing 9.

For the back channel, SD2 receives a Protobuf *Control* message from the S/A VM, following the same protocol (message length, message content). This message is parsed and information about the desired speed, the intended steering angle and if the car will drive autonomously in the next simulation step are interpreted.

The autonomous driving logic is simplified to accelerate with maximum force, if the desired speed is higher than the current speed and to brake with maximum force if the desired speed is lower than the current speed. This is necessary, since the autonomous parking algorithm only is capable of providing a desired velocity and no acceleration or deceleration values. The autonomous driving logic gets activated, as soon as the *Control* message sends an `autonomous=true` value. We integrated a safety feature to always

[4] The listen address and port can be changed at the top of the `human.cpp` file

[5] A short explanation of Nagle's algorithm can be found in section 3.2.3.

[6] For an explanation of the Google Protobuf protocol see section 3.1.1.

```
/* prepare protobuf message (serialize, calculate length, ...) */
uint32_t message_length = 0;
std::string output;
current_state.SerializeToString(&output);
message_length = htonl(output.size());

/* send protobuf message to S/A VM
 * 1. message length as uint32_t
 * 2. message (State) itself
 */
write(newsockfd, &message_length, 4);
write(newsockfd, output.c_str(), output.length());
```

Listing 9: `src/drivers/human/human.cpp`

brake with full force, if the car was driving autonomously but isn't anymore and the driver didn't take back control until now, see listing 10.

```
if (car->_accelCmd != 0.0 || car->_brakeCmd != 0.0)
  autonomous = false;

if (autonomous) // if user didn't overtake control after parking...
  car->_brakeCmd = 1.0; //... full brakes (obv. for safety reasons)
```

Listing 10: `src/drivers/human/human.cpp`

This of course only makes sense for the parking scenario and a different approach should be taken for autonomous driving, e.g. on the highway. A failure of an ECU while driving on the highway obviously shouldn't lead to a full brake for obvious reasons.

2.3.2 usr

The *usr* bot should be extended to just stay at the current position and don't move in any way to represent a parked car. We can achieve this behavior by forcing the `accelCmd` to a value of zero, the `brakeCmd` to a value of one and the `steerCmd` to a value of zero, as seen in listing 11.

This solution does work quite well in our setup, but the code should be moved to a completely new driving bot with an expressive name for the future, to still be able to use the *usr* bot in its normal programming.

```
// LL4
Car->_accelCmd = 0.0;
Car->_brakeCmd = 1.0;
Car->_steerCmd = 0.0;
// ---
```

Listing 11: `src/drivers/usr/src/usr.cpp`

2.4 Simulation Coupler

In the beginning of the practical course, it was planned to interpose the SimCoupler between SD2 and the S/A VM. To ease up the development process, reduce latency and prevent additional sources of errors we decided to exclude the SimCoupler for now. Unfortunately we had to cope with latency problems, optimization issues of the parking algorithm and other unforeseeable obstacles until the end of the practical course. In the end we didn't have enough time to reintroduce the SimCoupler but for the future it should be easily expendable, since for this use case the SimCoupler more or less acts as a proxy and doesn't introduce any additional logic. The only problem with such an integration could be a foible of the Protobuf library. Multiple instances of a Protobuf library are tricky to integrate in a single project, as one can also see in a discussion on StackOverflow [3] with possible solutions.

3 S/A VM (Alexander Reisner)

3.1 Data exchange with Speed Dreams 2

3.1.1 Protobuf

Google Protobuf is a protocol for data exchange. The newest version can be found here[1]. To exchange data, protobuf files have to be created beforehand. One of the protobuf files designed during this pratical course looks like this:

```
syntax = "proto3";
package protobuf;

import "sensor.proto";
import "wheel.proto";
import "specification.proto";

message State {
  repeated Sensor sensor = 1;
  repeated Wheel wheel = 2;
  Specification specification = 3;
  float steer = 4;
  float brakeCmd = 5;
  float accelCmd = 6;
}
```

- The version of the syntax is defined.
- All packages and imports are declared.
- The message is constructed. Attributes can be other Protobuf messages, simple datatypes, like int or float, or a list ”repeated” of any attributes named.

Figure 3.1: state.proto

3.1.2 TCP/IP

Before any network interaction can take place, the network has to be initialized. Therefore the library lwip is usually needed. But since the mosquitto library already initializes lwip, another initialization within the components code caused lwip and moquitto to freeze.
Furthermore dhcp and manual ip configuration have to be separated in the initialization, since dhcp ip assignment takes up to 10 seconds, however manual ip assignment takes place instantly. The configuration takes place in the run file of the component.

[1] `https://github.com/google/protobuf`

```
<network dhcp="yes" ip-address="192.168.178.3" subnet-mask="255.255.255.0" default-gateway="192.168.178.1" />
<speedDreams ip-address="10.200.32.15" port="9002" />
<mosquitto ip-address="10.200.32.15" port="1883" />
```

Figure 3.2: S/A VM run file

If dhcp is set to ”yes” any following configuration is ignored.
If dhcp is set to ”no” there is our address followed by the subnetmask and the default gateway of the VDE adapter or the connected ethernet network.
SD2’s address and port as well as mosquitto server’s address and port can be configured.

3.2 Our protocol

To be able to exchange data, some preparation needs to be done beforehand. Since it is not useful to send data without knowledge over its size, some kind of a protocol around the protobuf data exchange had to be designed. Therefore it was decided to send a 4 byte long value which contains the size of the following protobuf state, is sent to start the actual exchange. Any other messages are dropped.

3.2.1 Deserialization

Once a message of given size was received it needs to be deserialized. Fortunately the method ”ParseFromArray” does the job. Afterwards all received information is stored within a protobuf state. This object has automatically generated getter and setter methods, which makes it easy to access the actual data. Finally the multiplexing of the S/A VM takes place. Any data within the state object is published via mosquitto within the topic ”state”.

3.2.2 Serialization

Right after the deserialization of the state message, a new message called control is created. Any information the S/A VM got from the ECU is now put into this protobuf object. Afterwards the object is serialized using SerializeToString. As mentioned in 3.2 before the actual protobuf file can be sent over ethernet, its size in a separate 4 byte message, initializes the data exchange. But finally the serialized protobuf message is sent back to SD2 which can now continue with the next simulation step.

3.2.3 Nagle’s algorithm

The goal of the nagle’s algorithm[2] is to minimize the number of packets sent by the network stack. Therefore it piles up small messages. Unfortunately this approach brakes completely with the real time requirement of the S/A VM. Therefore this algorithm had to be disabled by activating the ”TCP_NODELAY” option of the lwip library.

[2] https://www.lifewire.com/nagle-algorithm-for-tcp-network-communication-817932

3.2.4 Do not wait for answer of ECU

To keep latency as low as possible the S/A VM answers immediately after it got a state message from SD2. No matter whether the S/A VM already got an update for the next step of the ECU. If the S/A VM waited for the ECU's mosquitto messages, there would be so much latency that SD2 had to run in 1/8 realtime. For the sake of realtime the S/A VM does not care if it misses a message from the ECU. This is not a problem since the ECU does not care about the past but only about the next step. Therefore the ECU is able to correct missing instructions in the next step.

3.3 Data exchange with ECU

MQTT

MQTT[3] alias Message Queue Telemetry Transport is a protocol based on the publish-subscribe pattern.
The publish-subscribe pattern describes a client server communication, where the server acts as a message broker and the clients subscribe and publish on topics. The broker knows the subscriptions of clients and sends them a message if a message is published under a certain topic. Therefore network load can be taken from the clients and put to the server, since the clients only get messages it is interested in.

3.3.1 Subscribe

The subscription is handled via "mosquittopp" which provides a method called "subscribe". This method needs the topic and the quality of message as input. Once a message on the given topic arrives at the server, the subscribed client is notified. This notification is handled by the callbacks 3.3.3 of mosquitto.

3.3.2 Publish

The publishing is also handled via "mosquittopp" which provides another function called "publish". This method takes the topic, the message to be published and its size. The message finds its way to the server via the configuration done in the run file of the ECU. This configuration works the same way as described in TCP/IP 3.1.2 for the S/A VM.

[3] `http://mqtt.org/`

3.3.3 Callbacks

Callbacks are invoked once a certain event occurs. In case of MQTT those events can be for example:

```
/* connect callback */
void on_connect(int ret);

/* publish callback */
void on_publish(int ret);

/* log callback */
void on_log(int ret);

/* disconnect callback */
void on_disconnect(int ret);

/* error callback */
void on_error();
```

Figure 3.3: publisher.h

3.4 Starting the algorithm

3.4.1 Car information

Once the car length, the car width, the wheel radius and the maximum steering angle arrived at the ECU via pub/sub from the S/A VM, a new car information object is created. This needs to be done only once per ECU, since this static information does not change over time. To be sure that the needed information arrived at the ECU boolean values were created that are set to true if a value arrived.

3.4.2 Compute next step

The linker between pub/sub and the parking algorithm is the ”receiveData” method. It is called once new values for the front, side and back laser, as well as the spin velocity, since the last timestamp, and the actual timestamp arrived. The topicality is handled via boolean values, which are set to true, if a value arrived and set to false if a values was true and sent to the parking algorithm. Only full sets of data are used, for the sake of real time.

3.4.3 Publish next step

After the computation of the parking step, the resulting actuator data is published on topic ”car-control”. The S/A VM and the team responsible for the actuator movement in the model car, subscribed to this topic. Consequently SD2 and the model car execute the commands of the parking algorithm.

4 ECU (David Werner)

4.1 Autonomous Parking

As part of our project we had the task of implementing an autonomous parking algorithm which is capable of controlling a car in the SD2 simulation. The goal of the algorithm is to find a parking lot and performing a parallel parking maneuver afterwards. Thereby it uses three range sensors (front/back/right) the car is equipped with to monitor distances to obstacles around the car. As these sensors are only able to detect other cars, parking bays need to be bounded by three cars as seen in figure 4.1.

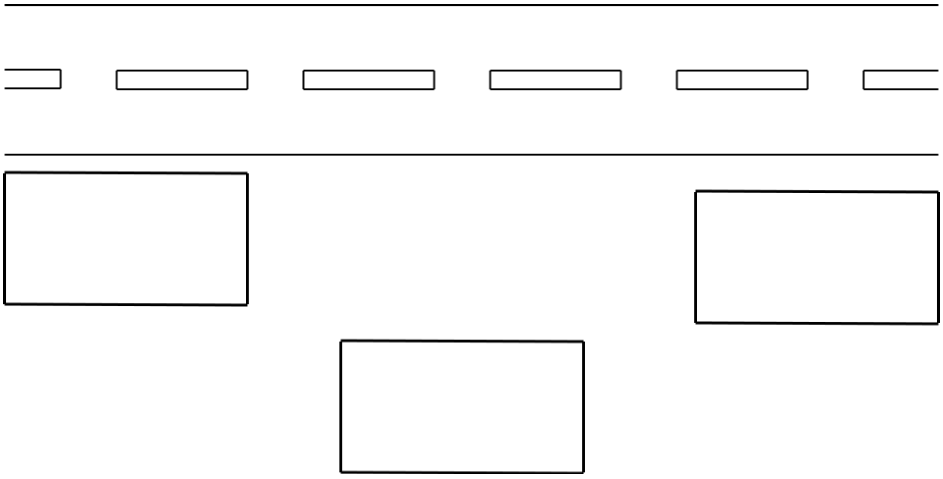

Figure 4.1: Parking lot bounded by three cars

4.2 Concept of the Algorithm

The autonomous parking algorithm presented by us is entirely based on the description of motion generation and control for parking an autonomous vehicle in [2]. The algorithm in the paper consists of three steps:

1. Localization of a parking bay
2. Adjustment of the vehicle position to a starting location
3. Execution of an iterative parking maneuver

Steps one and two are not discussed in detail. The third step is described as iterative task which comprises four steps:

1. Obtaining longitudinal and lateral displacement by processing range data
2. Calculate maneuver duration T and maximum steering angle ϕ_{max}
3. Steer the car into the parking bay by controlling its velocity and steering angle
4. Check if parking position is reached and start with step 1 if not

During all four steps the range data from the sensors is processed. The goal of the algorithm is to control the car only by influencing its velocity and steering angle. It is not desired that the car precisely follows a pre-calculated path.
For our purposes we modified the algorithm in two ways and thereby simplified it to a non-iterative 3-phase algorithm. We combined step one and two into one phase called searching phase in which the algorithm finds a parking lot and assumes the car to be positioned correctly for the following maneuver as soon as the lot is found. The third step of the algorithm is split up into two phases. The calculation of the parking parameters T and ϕ_{max} takes place in the calculation phase. Performing the actual parking maneuver is done in the controlling phase. Phase 2 and 3 are only carried out once. Therefore our algorithm is no longer iterative.

4.2.1 Motion of a vehicle

In order to be able to move a car using an algorithm one needs to have knowledge about how to mathematically describe its motion. Maneuvering a car into a parking lot is a two-dimensional problem. Therefore the car has three degrees of freedom: two coordinates, x and y, and an orientation angle θ. For the algorithm the coordinates refer to the middle of the rear wheel axle. With the help of these the motion can be described by the following equations:

- $\dot{x} = v \cdot cos(\phi) \cdot cos(\theta)$
- $\dot{y} = v \cdot cos(\phi) \cdot sin(\theta)$
- $\dot{\theta} = \frac{v}{L} \cdot sin(\phi)$

The change of the coordinate is dependent on the velocity v and the steering angle [phi]. Both of those can be described by formulas which are dependent on a timestamp t:

- $\phi(t) = \phi_{max} \cdot k_{\phi} \cdot A(t)$ - k_{ϕ} indicates the direction of the steering
- $v(t) = v_{max} \cdot k_v \cdot B(t)$ - k_v indicates algebraic sign of the velocity

They are used by the algorithm to calculate the actuator data during the parking maneuver. Besides the time dependency, the formulas rely on T and ϕ_{max}. These parameters are calculated within the calculation phase. Both formulas also use the following equations to calculate the ratio of velocity/steering at the timestamp t:

- $A(t) = \begin{cases} 1, & 0 \leq t < t' \\ cos(\frac{\pi(t-t')}{T^*}), & t' \leq t \leq T - t' \\ -1, & T - t' < t \leq T \end{cases}$

- $B(t) = 0.5 \cdot (1 - cos(\frac{4\pi t}{T})), \qquad 0 \leq t \leq T$

where $t' = \frac{T-T^*}{2}$. Using all these equations the motion of a car in a two-dimensional environment can be described.

4.2.2 Searching phase

In the beginning of the parking procedure the control over the car is handed to the algorithm. The algorithm starts with its first phase: the searching phase. Its goal is to find a suitable parking lot.
For our algorithm a suitable parking lot has two main characteristics. On the one hand it offers enough space to park and on the other hand it is bounded by two cars, one in front and one in the back. Empty spaces which are not between two cars or are too small will not be considered. The measurements which qualify a parking bay as suitable are dependent on the measurements of the car as well as predefined safety distances. An adequate parking lot therefore has a minimum length l_{min} and a minimum width w_{min}. In order to find such a parking lot the algorithm makes the car move along the traffic lane with constant, slow speed. While the controlled car passes parallel parked cars, the algorithm monitors the range data received from the right sensor looking for a potential parking lot. A potential parking lot starts if the range data implies that there is at least w_{min} empty space. As long as this condition is satisfied the algorithm sums up the distance the car traveled since the start of the parking lot. As soon as the range data returns to a value smaller than w_{min} the algorithm checks whether the summed up distance is as least l_{min} or not. If the parking lot was long enough the algorithm stops the car and transitions into the next phase. If the potential parking lot was too short its length is discarded and the searching phase continues.

4.2.3 Calculation phase

After the algorithm successfully finished the searching phase, i.e. a parking bay was found, it continues with the calculation phase. The tasks of the whole phase are carried out while the car is standing still. The purpose of the calculation step is to determine two important parameters for the parking maneuver: the duration T and the maximum steering angle ϕ_{max}. Both calculations are based on estimations which are then optimized. The duration of the parking maneuver is lower bounded by T^* the time it takes the car to steer from the negative maximum steer angle to the positive maximum steer angle. Therefore we use T^* as an estimation for T. The first estimation for ϕ_{max} is the maximum steering angle the controlled car offers. The optimization of both values relies on the evaluation of the equations for $\dot{x}$, $\dot{y}$ and $\dot{\theta}$ with the help of $\phi(t)$ and $v(t)$. The evaluation returns the coordinates of the final position of the car. On top of that the calculation phase needs a longitudinal condition C_{long} and a lateral condition C_{lat} which indicate whether the car can still move backwards (C_{long}) or to the right (C_{lat}). The optimization for T then works as follows:

1. Evaluation of equations using T as duration to get end position of the car
2. Check C_{long}
3. If C_{long} still holds increase T by a small amount and start at step 1
4. If C_{long} is harmed T is set to the last valid value and the optimization of T is finished

After we found the optimal value for T we search for the optimal value for ϕ_{max}. Therefore we check C_{lat}. If it holds we know that we already have the optimal value. If C_{lat} does not hold we reduce ϕ_{max} by a small amount and evaluate our equations again before checking C_{lat} again with the new end position. If C_{lat} is still not fulfilled we iteratively keep reducing ϕ_{max} until its optimal value is found. The reduction of ϕ_{max} decreases the slope of the car's steering curve which increases the distance of the car to the obstacle to its right after the parking maneuver.

4.2.4 Controlling phase

With all necessary parameters calculated, the algorithm is able to transition into the controlling phase. During this phase the actual parking maneuver is performed. Utilizing the formulas for velocity and steering the algorithm calculates the actuator data for the next simulation step and hands it to the simulation. For these calculations the optimized parameters from the calculation phase are used. While maneuvering the car, the algorithm always monitors all sensor data in order to prevent crashes. As soon as an upcoming collision is detected the algorithm triggers an emergency stop. After the car is steered into the parking bay the algorithm drives the car backwards until a safety distance constraint is violated. At this point the car is in the parking position and gets stopped by the algorithm which finished its execution by then.

4.3 Implementation

The implementation of the algorithm is based on three C++ classes/structs. While the *Parking* class implements the actual algorithm, *CarInformation* and *Map* serve as helper classes storing important information which are needed for the algorithm's calculations.

4.3.1 CarInformation

CarInformation is implemented as struct. Its purpose is being a container for car specific data which characterizes the car. As the parking algorithm is working for cars of any sizes CarInformation structs need to be constructed specifically for each car which should be controlled by the algorithm. This happens in the beginning of a Parking scenario.

The CarInformation is filled with data by subscribing to the S/A VM which provides necessary information. It contains the following data:

- length and width of the car
- wheel radius
- safety distances (as they depend on car measurements)
- max. velocity of the car during the whole parking scenario
- max. steering angle the car offers
- min. measurements of a suitable parking lot

4.3.2 Map

The Map class stores information about the parking lot and the car's starting position for the parking maneuver. When calling the constructor of a Parking object an empty Map is created. During the process of finding a parking lot the parking lot's measurements and the position of the car are calculated and put into it. The information in a Map plays an important role in the calculation of T and ϕ_{max} which is described in 4.2.3 conceptionally and in 4.3.3 from the implementation's point of view.
A Map object contains the following information:

- coordinates (middle of rear axle) and orientation angle of the car at the beginning of the park maneuver
- longitudinal and lateral displacements (length and width of parking lot)

Additionally, Map is equipped with getter/setter-methods in order to access and modify the data.

4.3.3 Parking

The Parking class implements the algorithm. On code-level the algorithm has four different states:

- Searching
- Calculating
- Controlling
- Parked

Each of the first three states represents one conceptional phase of the algorithm. The fourth state is entered if the car is in its final parking postion or if an emergency stop is performed. A Parking object maintains a state variable which indicates the current state of the algorithm. The main method of Parking is *receiveData*. It is the only public method and needs six parameters which are all received from the simulation:

- sensor_front - range data from sensor in the front
- sensor_right - range data from sensor to the right
- sensor_back - range data from sensor in the back
- spin_velocity - spin velocity of the car's wheels
- timestamp - duration of the current simulation step
- publisher - pointer to the publisher object which is called to send actuator data

Each time *receiveData* is called the method starts by checking the range data for an upcoming collision which would trigger an emergency stop. Afterwards actuator data for the velocity and the steering angle is calculated and then published. How the actuator data is determined and which values it has depends on the state:

Searching
The initial state of a Parking object is the Searching state. In this state the value of the actuator data is determined based on the return value of the helper function *_findParkingLot*. The values of *sensor_right* and *spin_velocity* are handed to this method as parameters. It uses them to determine whether a suitable parking lot was found or not. If *_findParkingLot* returns true the car passed a parking lot of reasonable size and all actuator data is set to zero which means the car needs to stop. The algorithm then transitions into the Calculating state. If *_findParkingLot* return false the velocity actuator value is set to the maximum speed value from the CarInformation struct. In this case the car keeps moving with constant speed as the algorithm keeps searching a parking bay. In order to determine its return value *_findParkingLot* uses the data of the right sensor and the spin velocity to find a parking lot according to the procedure described in 4.2.2. The total traveled distance is calculated by summing up the traveled distances in each simulation step. The distance covered in the current simulation step can be determined by multiplying the spin velocity with the current timestamp and the wheel radius from CarInformation. Additionally the Map object gets updated.

Calculating
When *receiveData* is called while the algorithm is in Calculating state the optimized values of T and ϕ_{max} are calculated according to 4.2.3. When the constructor of Parking was called both of them were set to their first estimation value. The final value of T is determined by calling the *_calculate_T()* method. Within this method the motion equations of the car are evaluated using discrete integration. This procedure is depicted in listing 12. In order to have current values of the velocity and the steering angle the algorithm evaluates *_velocity* and *_steering_angle* in each step of the integration.

```
for(double ts = 0; ts <= _T; ts += _sampling_period){
        s_angle = _steering_angle(ts);
    velo = _velocity(ts);

    if(s_angle == 0){
                _phi_old = _phi;
                _phi = _phi;
                _x = _x + (velo * _sampling_period * cos(s_angle));
                _y = _y + (velo * _sampling_period * sin(s_angle));
        } else {
                _phi_old = _phi;
                _phi = _phi + (((velo * _sampling_period) / _info.length_car) * sin(s_angle));
                _x = _x + ((_info.length_car / tan(s_angle)) * (sin(_phi) - sin(_phi_old)));
                _y = _y - ((_info.length_car / tan(s_angle)) * (cos(_phi) - cos(_phi_old)));;
        }
}
```

Listing 12: Discrete integration performed in *_calculate_T()* and *_calculate_local_max_steer()* - Parking.cc (101-116 and 145-160); comments removed

The evaluation returns the coordinates of the end position of the car. These coordinates are given to the *_longitudinal_condition* method which checks the C_{long} condition. If it is fulfilled T is increased by the value of *_sampling_period* (value of current timestamp) and another discrete integration is performed. This procedure continues iteratively until C_{long} is no longer satisfied. At this point the last valid value is assigned to T. After T was determined *_calculate_local_max_steer() is called.* This method checks whether C_{lat} is satisfied. If not ϕ_{max} is reduced and the discrete integration is performed again. This is repeated iteratively until C_{lat} is fulfilled and therefore a valid value for ϕ_{max} was found. After finishing all calculations the algorithm immediately transitions into the Controlling state.

Controlling

While in Controlling state the actuator data is determined by simply evaluating *_sterring_angle* and *_velocity* with an own timestamp value *_maneuver_timestamp*. This value is needed as the parking maneuver needs to start at timestamp 0. After claculating actuator values *_maneuver_timestamp* is increased by the current timestamp.

Parked

The Parked state is entered in one of two cases: either the car needed to perform an emergency stop or it reached the parking position. Regardless of which is the case this state was transitioned to as some range data indicated an collision in the near future. If parked is the current state *receiveData* keeps publishing 0 as actuator data for both, the velocity and the steering angle as the car needs to stand still.

4.4 Problems

Although the algorithm is able to perform all three phases of the algorithm without any disturbances in most of its executions there are still some issues which need to be fixed in order to make it work deterministically for any parking scenario.

4.4.1 Safety distances

During the process of controlling the vehicle's motion the algorithm has some problems in the interplay with SD2. As SD2 is not capable of explicitly setting the velocity of a car, it instead accelerates and brakes in order to reach the velocity which was transmitted as actuator data. Furthermore the controlled SD2-car tends to slide when the algorithm brakes it even at low speed. These issues cause that the car is not stopped instantly if zero is sent as velocity value. As a result of this, crashes occur in the simulation if reasonable safety distances are chosen. Therefore these safety ranges (especially the one in the back of the car) need to be set bigger than necessary. This enlargement has two consequences: the size of suitable parking lots grows considerably and unneeded emergency stops occasionally happen. While larger parking lots are not too much of a problem in a simulation environment, the second problem is more severe. The algorithm stops the car as soon as some sensor's data implies an upcoming collision. With overly large safety distances this behavior leads to unwanted situations where the algorithm unnecessarily stops the car. The worst case of this phenomenon occurs during the parking maneuver while the car is being steered into the parking lot as seen in figure 4.2. At this point of time the parking maneuver fails for no actual reason as there is enough space to perform the rest of the maneuver.

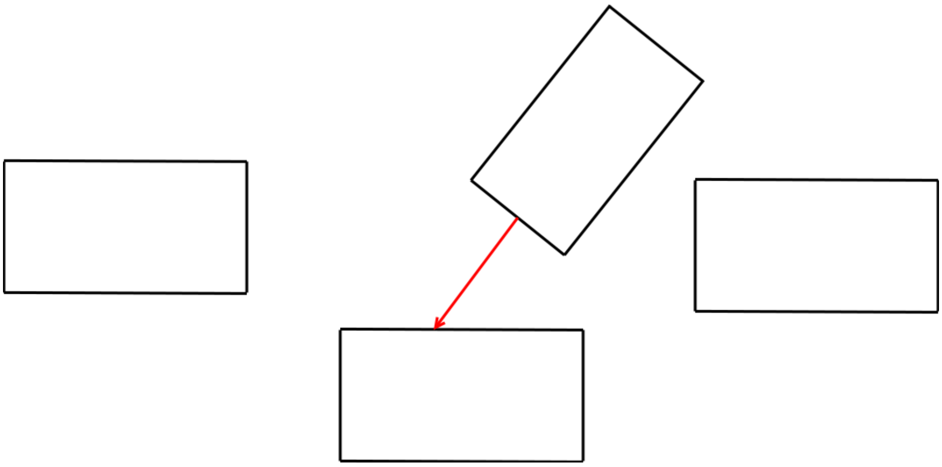

Figure 4.2: Emergency stop due to unnecessary large safety distances

4.4.2 Dependance on unrealistic time estimations

In the current state of the implementation, the algorithm seems to calculate wrong values for the duration of the parking maneuver T. The problem is that in the calculation phase where T is optimized, only one optimization step is done. This is shown by some sample data in table 4.4.2.

T*	T (optimized)
5	5,0022
10	10,0023

Conceptionally the number of optimization steps should only depend on the length of the parking lot (longitudinal displacement) as well as the maximum velocity and the maximum steering angle of the car. This means that for any value chosen for T* the optimization should return the same value for T as long as the initial estimation for T* is below the optimized value. Unfortunately this is not the case. As shown by the sample data T* = 5 gets optimized to 5,0022 although it should at least be optimized to 10,0023.

The consequence is that a successful execution of the parking maneuver is entirely reliable on an unrealistic estimation of T* (in our case: 12). The closer the value for T* is chosen to a reasonable value for T which should be calculated automatically, the better the algorithm performs.

4.5 Improvements

4.5.1 Rework of calculation phase

As the value for T do not seem to be calculated correctly as described in 4.4.2 a possible improvement of the algorithm would be a rework of the calculation phase. In order to do that one would at first need to test the current implementation extensively to find out what the exact problem is. We suspect the calculation of the longitudinal and the lateral conditions to not work as they are intended although it might also be that our implementation of the discrete integration might be buggy.

4.5.2 Expanding the parked condition

The current implementation of the algorithm determines whether the car is parked or not solely on range data received from the sensors. After the algorithm steered the car into the parking bay, the car is forced to drive backwards until a safety distance is violated. This causes the algorithm to stop the car and go into the PARKED state. Because of the problem discussed in chapter 4.4.1 this behavior is justifiable. We need to have overly large parking lots and it therefore makes sense to exceed T in order to drive backwards until a reasonable distance to the obstacle in the back is reached. Nevertheless it could be necessary to modify this approach. Even though the processing of sensor data will always be necessary in order to support emergency stops this mechanism does not have to be the only way to stop the car after the parking maneuver is finished. The algorithm could be altered in order to stop the car after the time bound T for the parking maneuver is reached. Unless the calculation phase is fixed as described in 4.5.1 this modification can possibly lead to scenarios where the car parks fairly far away from the car behind of it.

4.5.3 Implementing the original algorithm

As we decided to simplify the original algorithm by reducing the iterative step to only one step there is the possibility to expand our algorithm to an iterative algorithm. Such an modification would mean that our calculation and control phases are performed within each iterative step and are therefore performed multiple times. This might increase the precision of the algorithm regarding the positioning of the car. With multiple maneuvering iterations the car is more likely to end up at the optimal parking position. Additionally an algorithm supporting iterative steps should be able to park the car in smaller parking lots.

4.6 Conclusion

Based on [2] we implemented an algorithm that is capable of finding a parking lot and perform a parking maneuver autonomously. Due to issues caused by the interplay with SD2 and the calculation of parking parameters the algorithm does not perform deterministically. Although it parks the car correctly most of the time the algorithm still needs some more optimization. In order to motivate the reader of our paper to do such an optimization, we offered multiple suggestions for improvements.

Bibliography

[1] Mohommad Reza Bonyadi, Samadhi Nallaperuma, Daniele Loiacono, and Frank Neumann. Simulated car racing championship 2015. `http://cs.adelaide.edu.au/~optlog/SCR2015/index.html`. Last visited 2017-08-05.

[2] Igor E. Paromchtchik and Christian Laugier. Motion generation and control for parking an autonomous vehicle. `http://www.cs.dartmouth.edu/~spl/Academic/RealTimeSystems/Spring2002/Labs/FuzzyControl/ControlForParking.pdf`. Last visited 2017-08-06.

[3] piaoxu. Several shared object using same proto leading the the error: file already exists in database. `https://stackoverflow.com/questions/37051635/several-shared-object-using-same-proto-leading-the-the-error-file-already-exist/`. Last visited 2017-08-06.