Autonomous Robots Report Group 7

Mårten Sanderöd Abishek Swaminathan Yinsong Wang Emiel Robben

May 24, 2024

Introduction

This project is developed in Linux using different distributions such as OpenDLV Desktop and Ubuntu. All development is done on local developer machines and is then deployed with Docker to a Raspberry PI v4 which drives the machine hereby referred to as the "kiwicar". The kiwicar is a small autonomous vehicle consisting of driving capabilities, a camera, two IR sensors, and two ultrasonic sensors. The goal of the project was to complete three different tasks that test knowledge and problem-solving in perception and driving logic as well as the deployment pipeline using CI/CD.

Microservices

The microservices developed for the milestones and subsequently used for the project include two different perception routines, one for simple cone detection and one for kiwicar detection amongst others as well as driving/collision avoidance routines for each task. In general, the microservice "thinking" was applied throughout the project. All microservices were isolated only to do one thing, either drive or perceive. This made it easy to replace one service if updates were needed without any impact on the other programs given that the message types were not changed. To enable communication between different microservices the project utilized the Libcluon library as well as the OpenDLV message standard v1. For image sharing, the Libcluon library is not suitable for camera applications and instead, a shared memory was utilized and the perception microservices was connected to it by volume mapping.

Perception

During the course of the project, we had two different implementations of the camera perception. The perception was first introduced in milestone 2 where a simple cone detection algorithm was created. Afterward, a more sophisticated model was deemed to be necessary to handle more diverse detection such as kiwicars, blue A4 papers, Post-it notes, and cones.

OpenCV cone detection using color filtering

The implementation of the perception logic utilized the built-in functions in OpenCV to filter the image based on color thresholds. Since the color thresholds are very sensitive to lighting it was important that these values could easily be changed in real time. This was done by the use of OpenCV trackbars. By setting a default upper and lower color threshold and then using the trackbars to update the thresholds, optimal values could then be achieved.

The program is built to be modular and works for any color and shape detection. This is done by having one microservice per color and controlling certain variables such as **minEdges**, **maxEdges**, **erosions**, and **dilations**. These variables control the image processing. Before any image processing is done, the image is divided and only the relevant area of interest is analyzed. This is controlled by variables set by our trackbar. By default, only the bottom of the image is used for the masking together with a mask for the kiwicar. This increases the efficiency of the program since the computationally heavy algorithms are running on a smaller mask.

Before any processing is done, the image is converted to the HSV color space which is preferred by OpenCV. The pixels that survived the mask are then dilated and eroded X number of times based on the trackbar values. The image is then passed to the **findContours** function which is a built-in function in OpenCV. This finds different contours in the image and returns a vector containing these. We then loop over every contour found and apply the OpenCV functions **ApproxPolyDP** and **convexHull** which approximates a polygonal curve in the contour and finds the smallest convex hull of it. This is being done so we can judge the contour based on shape. For cone detections, we filter out any shape that has either 3 or 4 points which ensures that false positives are filtered out.

The detections are now ready to be sent with OpenDLV and Libcluon. Before any detection messages are sent, the **startFrameMessage** is sent to indicate that a new detection is ready and will be sent. Then for each detection, the following messages are sent after normalizing all coordinates between 0 and 1:

- opendlv::logic::perception::DetectionBoundingBox
- opendlv::logic::perception::DetectionType
- opendlv::logic::perception::DetectionProperty

and after all messages are sent the **endFrameMessage** is sent to indicate that it is done transmitting.

OpenCV + Darknet

A comprehensive dataset was collated by creating images and labels using several data replays that were gathered in the lab of different scenarios on which we wanted to train the model. This dataset included kiwicars, cones, A4 papers, and post-it notes.

Initially, we tried to use YOLOv8n in Python to train the model. This worked great until we tried to integrate the saved model into the C++ logic by OpenCV command and the model wouldn't load. After many hours we instead decided to go with Darknet which only has support for YOLOv4 and YOLOv7 but the integration with the C++ program worked better. We picked YOLOv4-tiny and set the input image size to 416 by 416. Lowering the image size would make the model quicker but would also sacrifice accuracy. The Darknet model was trained through Google Colab, leveraging

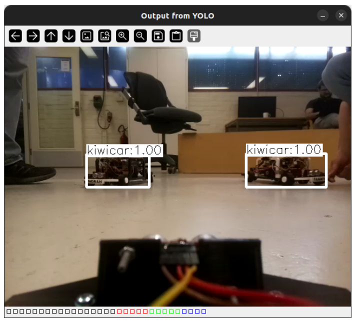

Figure 1: Kiwicar detection using YOLOv4-tiny

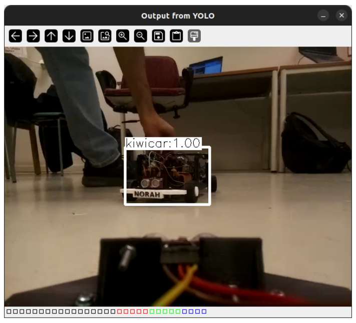

Figure 2: Kiwicar detection using YOLOv4-tiny

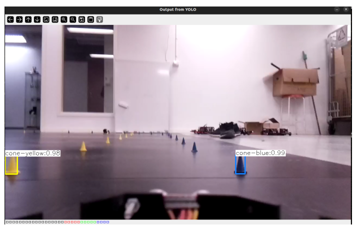

Figure 3: Cone detection using YOLOv4-tiny

its free GPU computing power to speed up the training process. In general, the perception works great for kiwicars, blue A4, and Post-it notes. It works partly for cone detections and would probably need more annotated data to perform better. When deployed on a Raspberry Pi, we noticed that the performance was really bad, as low as 2 FPS. This made our plan to use it for all tasks to falter and we had to revert to our other perception algorithm.

The code itself connects to a shared memory and then runs the image through the model. Afterwards, the output is post-processed by performing non-maximum suppression to eliminate overlapping boxes and boxes falling under our chosen confidence threshold. The resulting boxes are then sent as a Libcluon message in three parts using the standard OpenDLV messages provided. The messages sent are:

- opendlv::logic::perception::DetectionBoundingBox
- opendlv::logic::perception::DetectionType
- opendlv::logic::perception::DetectionProperty

In the DetectionBoundingBox, we set the detectionId based on what type of detection we made. This made it easy to only use the DetectionBoundingBox message to convey all important information. Similar to our other perception implementation, this microservice also sends the **StartFrameMessage** and **EndFrameMessage** to indicate which detections correspond to which frame. By sending the same type of messages as our other perception logic, we can easily swap out the logic depending on the task.

Driving actuation

We had two different driving microservices but with very similar setup. To make the behavior as simple as possible, we aimed to create a behavior that focuses on the most explicit characteristics of the perceptual input. For Milestone 2 we already created code for cone detection, so we decided to leverage that and create a microservice that listens to the detection messages sent and takes driving decisions based on the inputs. The code is heavily inspired from assignment 2 where the "Behavior" class was introduced which utilizes mutexes to store and retrieve data. The code also has a time trigger based on a frequency passed to the program. Each time the time trigger activates, the **Behavior::step** function runs which executes the driving logic. The **Behavior** class also exposes functions to update the current bounding boxes captured by the message trigger setup. This is where it is important to use mutexes so we are not able to read and write to the same memory at the same time. Each time a message with the **opendlv::logic::perception::DetectionBoundingBox** ID is received, the content is pushed to a vector so all bounding boxes in the frame are captured. This is controlled by the **StartFrameMessage** and **EndFrameMessage**. On a **StartFrameMessage** the vector is cleared of all **DetectionBoundingBoxes** and the **receivingData** flag is set to true. This flag is set to true until a **EndFrameMessage** is sent. This flag is later utilized in the **Behavior::step** function.

The **Behavior::step** function includes the logic for setting the two messages **opendlv::proxy::PedalPositionRequest** and **opendlv::proxy::GroundSteeringRequest**. These two messages are sent to the kiwicar which contains several internal microservices running to convert these messages to signals to be sent to the motor and steering. The angle sent to the **opendlv::proxy::GroundSteeringRequest** is decided by a finite state machine (FSM) while the speed is controlled either constant or controlled by a trackbar.

CI/CD Pipeline

The CI/CD pipeline visualized in Fig. 4 has played a crucial part in our project and has been the main way of deploying our code onto the kiwicar. Each project is containerized inside a docker container and can be downloaded to the kiwicar. This is possible since we are cross-compiling our images and allowing each image to be executed on different platforms. This removes the hassle of using tools like **scp** to copy over code to the kiwicar and ensures that we are always using the latest version by pointing the image to the **dev** image tag.

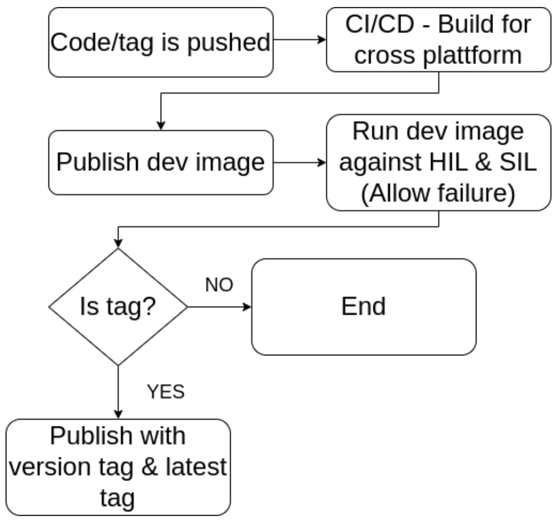
Code/tag is pushed
CI/CD - Build for cross plattform
Publish dev image
Run dev image against HIL & SIL (Allow failure)
Is tag?
NO
End
YES
Publish with version tag & latest tag

Figure 4: Data pipeline for most repositories used in the project.

On every push to the main branch (and in task 3's repository we enable every branch) we trigger the CI/CD pipeline to build a dev image with **docker buildx**. After the build command is done, the image is published with a dev tag, and the integration testing is triggered. Since the integration testing sometimes failed, the pipeline allowed failures and kept on. If a git tag was present, it also published a **latest** tag together with a tag corresponding to the commit tag. This configuration ensured that we could backtrack and revert changes if need be. However, since the docker-compose file was often updated for variables and fine-tuning as well this created trouble for us.

DODO integration testing

We had major issues with the DODO integration testing. We were one of the groups piloting the software and never got it to run a successful test. This was due to many reasons but the main ones were

- Images were too big due to using an Ubuntu docker image
- Coordinates were not normalized to 0-1 range
- SIL never ran since HIL would always time out
- HIL still timed out on **docker pull** even if the image was $\sim$ 100 MB

We believe that the last point was due to the Gitlab runner caching our big image and not correctly pulling it. It timed out after 300 seconds and we could not figure out why. By the time we got it to work, we were so far into the project that lab testing was prioritized over DODO testing.

Our algorithms would probably be better if we got the SIL to work since we could've tested the driving logic without having to worry so much about the perception logic.

Task 1

The goal of task 1 is to create a robot behavior that drives around a track as quickly as possible without hitting any cones. The track is 12 meters long and has blue cones on the right, and yellow cones on the left, separated by a distance of 40 cm. No other machines will be on the track.

Workflow

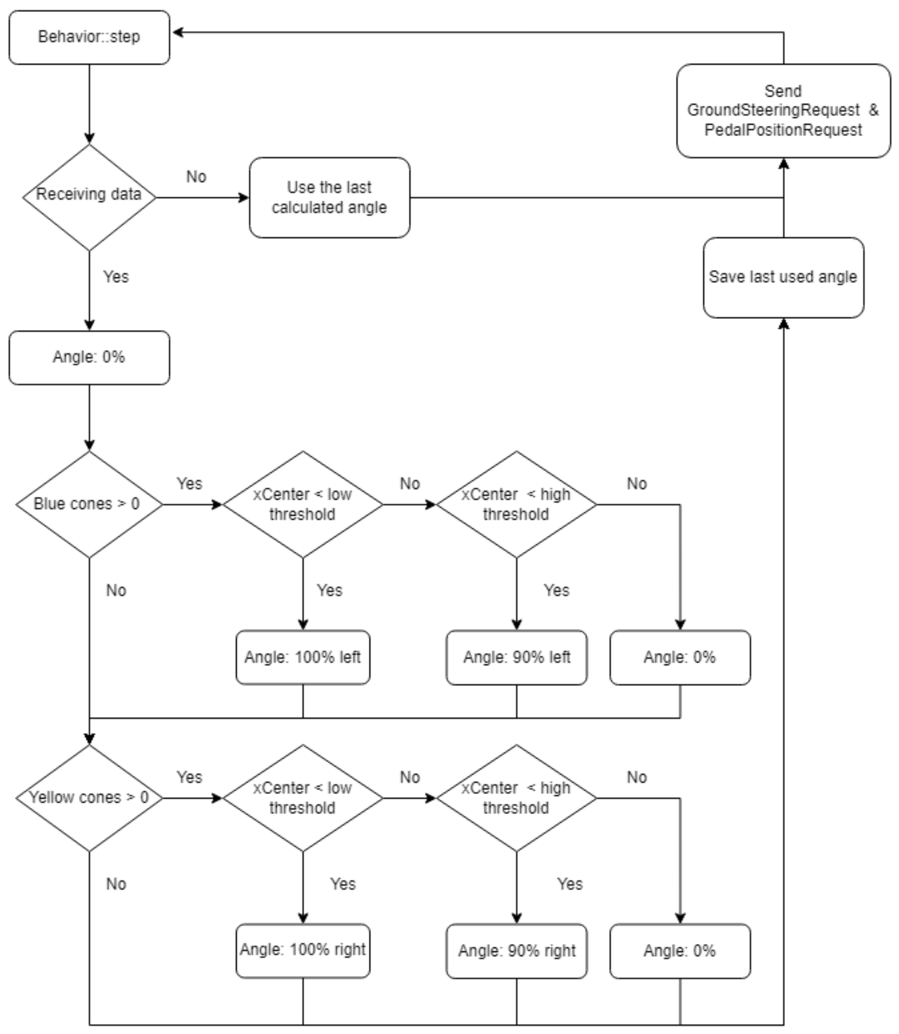

Figure 5: A Finite State Machine diagram for the behavior implemented for task 1.

To fine-tune and test the behavior of the kiwicar, the trackbars from OpenCV come in handy. These were especially useful for controlling the perception parameters as well as angle calculations. The main influence of this was a better detection of the cones in dynamic conditions such as a change in lighting, the presence of objects of similar color, etc. One other use of this was in deciding optimal values for the X-axis and Y-axis threshold values to decide what cone/s we wanted to look at and prioritize our driving logic on the cone/s as well as controlling the speed of the kiwicar.

The driving actuation implements the previously described microservice with the **Behavior** class but with a custom **Behavior::step** function. The step function implements a finite state machine (FSM) for calculating the **GroundSteeringRequest** message. The FSM logic works by analyzing the center of each bounding box. The bounding boxes have normalized x and y values between 0 and 1 where x corresponds to the horizontal plane and y the vertical plane. (0,0) is located in the top left of the image. First, it checks if we are still receiving data from the **receivingData** flag, if so, use the last calculated angle and return. If there exists any blue bounding boxes, find the bounding box that lies most to the left in the image and use it as a reference point. If the horizontal center of the bounding box lies more to the left than **xThresholdLow**, fully turn to the left. The threshold is controlled by the **xThresholdLow** variable and can be adjusted with the trackbar. If the bounding box lies above **xThresholdLow** but below **xThresholdHigh** still turn to the left but not as much.

The exact same logic is implemented for each yellow cone detected but with different thresholds and instead of turning left, we are turning right. Since the yellow check is being done after the blue check, higher priority is being given to the yellow bounding boxes which makes sense since the yellow cones are on the left and it's a lefthand track. The angle calculated is then mapped to radians and sent with the **opendlv::proxy::GroundSteeringRequest**. This algorithm is shown in a FSM diagram in Fig. 5.

Programs running during the simulation and while on the track

When simulating, 3 microservices are running namely the kiwi-recordings, opendlv-camera-perception, and the project-task-1. When the kiwicar is actually on the track, it uses its live camera feed instead of the data replay container.

Task 2

The goal for task 2 is as described in the assignment description: "In a fenced-in open space, automatically park on top of a blue A4 paper. Then drive around looking for five green post-it notes. When a post-it is found, let the robot drive over it and wiggle the front wheels to indicate that it found the target. The robot needs to be repositioned on the blue paper every three minutes, otherwise, the robot loses."
We chose to prioritize task 1 and 3, and consequently, did not harvest the working code for task 2. What follows here is an outline for a plan to follow when solving task 2.

Plan

For this task, we want to use the code we wrote for milestone 3. There, we succeeded in letting the kiwicar drive towards a blue paper, and park on top of it. Additional requirements beyond milestone 3:

- Find green post-its.
- Driving towards the green post-it (here we can use the driving logic code from milestone 3).
- When on the green post-it, wiggle its front wheels
- Continuing to the next post-it.
- Go back to blue paper before the end of the third minute.
- Continue finding new green post-its (if not visited).

An open question is how to make sure to visit all green post-its. A strategy is to employ a 2-dimensional random walk until it finds its next post-it, until it finds all post-its, or until it needs to return to the blue paper in 3 minutes.
Another open question is how to make sure that the kiwicar returns to the paper every 3 minutes. One can start a counter once the car leaves the blue paper, and at a to-be-determined time stamp, the kiwicar should abandon what it is doing and go on the exploratory search for the blue paper instead. An alternative approach is to use the kinematic equations from the lecture and assignment 2, one could calculate the coordinates that the robot is at for every time step. When the timer is halfway, it

could ask the cart to drive back with the exact angles (inverted) to return to the blue paper. Another alternative approach could be, to make use of a map and calculate the shortest path back to the blue paper (labeled as the origin), and then go back to the origin.

Task 3

The goal of task 3 was to find and follow other kiwicars, without being seen. For this task, the code for milestone 4 comes in handy.

Workflow

We decided to use the perception which utilizes the YOLOv4-tiny network to find kiwicars very accurately as shown in Fig. 1, 2. However, the performance of the model requires us to drive slowly so that the control logic can get up-to-date information from the perception microservice.

We use the same software architecture as in task 1 together with the same message-receiving pipeline with mutexes and lock guards. The **Behavior::step** function executes actions to drive the kiwicar which is triggered by a time trigger defined by a set frequency. If we do not detect a kiwicar, we will switch into a "searching" mode where we run in a circle unless we have some obstacles detected in front of the car which will then trigger the collision-avoidance mode. The collision-avoidance mode checks if we are closer than **closeDistance** parameter and if so the kiwicar will reverse until it hits the **farDistance** distance. When it hits the **farDistance** threshold, it checks which side is close to a wall and then drives the other way. If no walls are senced, it drives to the left. When it is further than the **farDistance** it will go back to "searching" mode and continue looking for kiwicars.

If we detect a kiwicar, we will switch to the "following" mode. We calculate the angles based on the X-axis from the detection bounding box sent by our perception algorithm. The angle is then converted to radians and sent with **opendlv::proxy::GroundSteeringRequest**. If we are closer than the set close distance, the kiwicar will stop, and if we are between the close distance and the far distance, the kiwicar will keep base speed; if we are far away from the far distance, the kiwicar will speed up. This can be visualized in the FSM diagram presented in Fig. 6

Programs running during the simulation and while on the track

When simulating, 3 terminals are opened running inside the kiwi-recordings, opendlv-camera-perception, and the opendlv-kiwicar-drive folders. The first functions as a virtual perception, the second as a pattern-recognition program, and the third makes decisions based on the patterns it gets, according to the goal of task 3. When the kiwicar is actually on the track, it uses its live camera feed as an input to perception instead of a data replay.

GitLab Repository

Following is the parent directory consisting of our codes hosted on Gitlab for all microservices deployed for the project.
`https://git.chalmers.se/courses/tme290/2024/group7`

- Kiwicar-recordings consist of the files required for the data replay.
- opendlv-camera-perception consists of the Perception microservice for cone detection
- opendlv-kiwicar-perception consists of the Perception microservice for detection using YOLOv4-tiny as well as code for labeling the data and the model weights and configuration

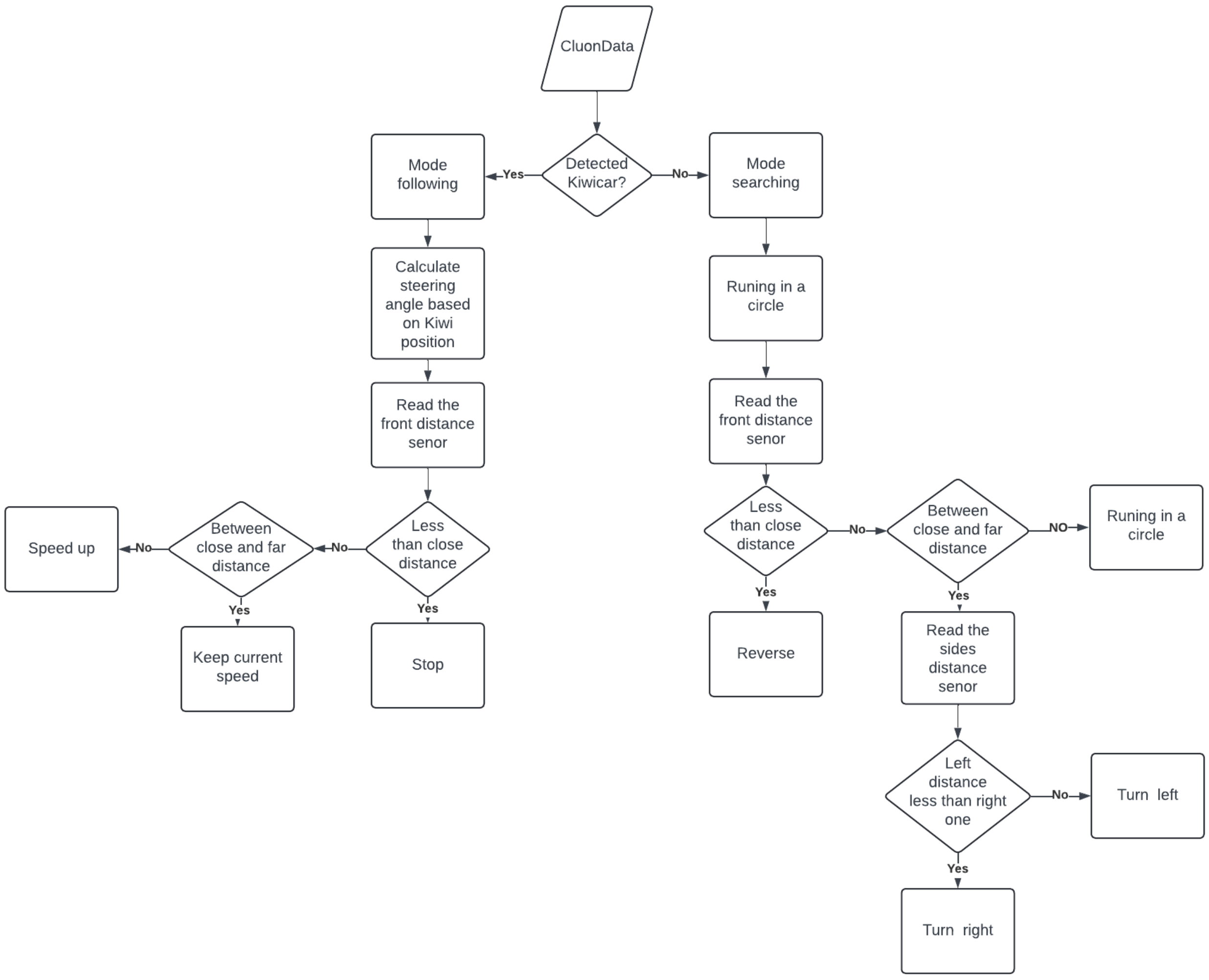
CluonData
Detected Kiwicar?
Yes
No
Mode following
Mode searching
Calculate steering angle based on Kiwi position
Runing in a circle
Read the front distance senor
Read the front distance senor
Less than close distance
No
Between close and far distance
No
Speed up
Yes
Keep current speed
Yes
Stop
Less than close distance
No
Between close and far distance
NO
Runing in a circle
Yes
Reverse
Yes
Read the sides distance senor
Left distance less than right one
No
Turn left
Yes
Turn right

Figure 6: Drive logic for tracking another kiwicar

- Project-task-1 consists of the Drive microservice with logic specific to task 1
- Milestone 2 and 3 are for various milestones
- opendlv-kiwicar-drive also consists of the Drive microservice with driving logic specific to task 3