



Wrocław University
of Science and Technology

Supervisory control synthesis for mobile agent system Intermediate Project - final report

PAULINA PORCZYŃSKA
4 FEBRUARY 2019

TUTOR: PHD WITOLD PALUSZYŃSKI

FACULTY OF ELECTRONICS
EMBEDDED ROBOTICS



1 Introduction

1.1 Problem description

The main objective of this project is to develop a controller able to administrate group of autonomous robots in such a way that no collisions would occur and movement would be smooth, without unnecessary delays. This controller (called later Supervisory or Main Controller) should be above all robots, should know each robot path and position and be able to detect potential collisions and prevent them. We can say that Main Controller does not have the direct power over any of the robots but rather gives permission for movement when robots ask for it.

Each robot should be an individual, autonomous entity and have it's own task - it's own route to follow. This paths can overlap and intersect. Robots inform the Controller about their next move and wait for appropriate responses. As the result, every robot should be able to fulfil it's individual task as smoothly as possible and without any collisions. Ideally, robot movement could be random, however here, for the sake of the simplicity, each robot has a predefined path and moves like on rails. Also because this reason, the robots are represented by discs with known radius.

Developing a simulation of movement of several robots, each following it's own path and avoiding collisions, was the desired result of this project. In addition, there should be room for future development and upgrade of this system, so the whole project could be easily expanded. All code that was develop for this project is available here (6).

1.2 Assumptions

This project is based on a Engineering Thesis(5) in which similar problem was implemented in Matlab. Here, the whole final system was written in ROS environment, as independent package, written in Python. Main idea that stays behind developing such system is to base the architecture on state machines. The controlling parts of both robots and supervisory controller could be implemented as event-based system (DES). As explained in next parts, this assumption was only partially satisfied but for good reasons.

1.3 Programming Environment and other tools

Project was implemented as ROS Kinetic package(2) with use of Smach automaton library(3). In addition, for better result presentation the python Matplotlib library examples was used (4).

2 Implementation

2.1 System architecture

Using the benefits that *ROS* architecture provides, the whole system consists of several nodes:

- Time generator *try_clock.py* providing simulation time for continuous part of the robot.
- Position estimator *get_coord_server.py* that computes robot position on XY plane. One node of this type should be provided for each robot.
- Logger *plot_server.py* that saves one robot results in .txt file. One node of this type should be provided for each robot.
- Supervisory Controller *main_controller_server3.py* (for 3 robots) that provides collision-free movement.
- Robot node *DummyRobotPath.py* which connects all of the above parts into one system.

Time generator, position estimator and logger are self-explanatory nodes without any tricky parts. While setting up the time generator, the Δt value can be chosen (default value is 0.05). It determines the execution time and accuracy. Position estimator takes the distance traveled by robot and returns its position on XY plane according to provided path.

The Main Controller node is more interesting as it basically is the main functionality of this whole system. While solving the analogical problem in Matlab (shown in (5)) the architecture of Supervisory Controller was based on DES and was controlled by an automaton. The initial idea here was to do it under the same principles. However, after rethinking the architecture and getting to know the ROS better it was clear that Main Controller can work even better as Ros Service. That means, giving orders to robots only when robots want it (we assume that robots want the best for the whole system so they are politely asking for permission when they should). This

solution can be treated as event-based because Main Controller is driven by the event "obtain question from robot".

Main Controller knows the routes of each robot as well as its size. On this basis it computes the collision matrix for these paths. Collision matrix is kind of look-up table that tells in which sectors the robots could be without having a collision or deadlock. If sector n on path $P1$ and sector m on path $P2$ are closer to each other than the sum of radii of the robots going along this paths, then of course, these two robots cannot be in this sectors at once. Other problem is the deadlock avoidance. The formation of deadlock is shown on picture 1. The simplest solution of this problem is by analysing the collision matrix and expand the forbidden states by possible pre-deadlock arrangements.

The last and most important part of this project is the Robot part. The Robot node consists of two parts - discrete and continuous. Discrete path is the less complex one. It is based on a simple automaton that has three states: Moving, Idle and Action. The Action state is not used here and is a seed for future development. The Moving state checks if robot can keep moving. If not (when robot needs obtain permission), the Idle state occurs. In the Idle state, the robot asks Main Controller for permission until it is given and Moving state can occur again. Both states can transit to 'quit' when simulation time is up. This simple automaton was developed using Stage(?) and is shown below on 2.

Finally, the continuous part of the robot has all of its variables defined (size, velocity, path...), is connected to Time Generator, Position Estimator and Logger and update its own position with every clock tick. At this point of development of this system, the robots don't have an acceleration but an easy way of implementing it was provided here.

As stated before, source code for this project is available (6).

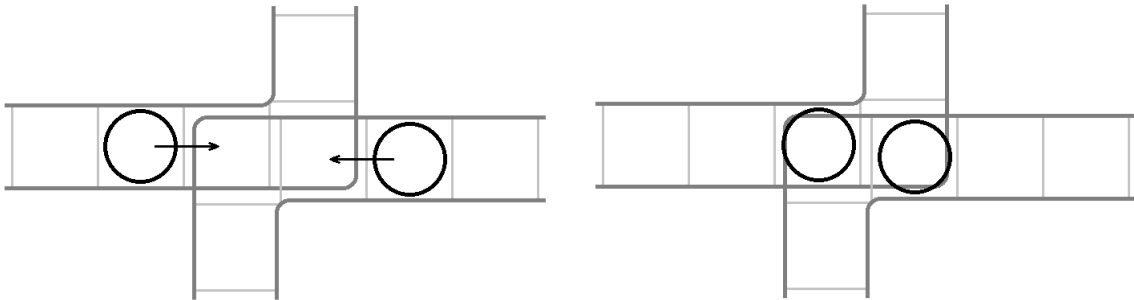


Figure 1: Deadlock

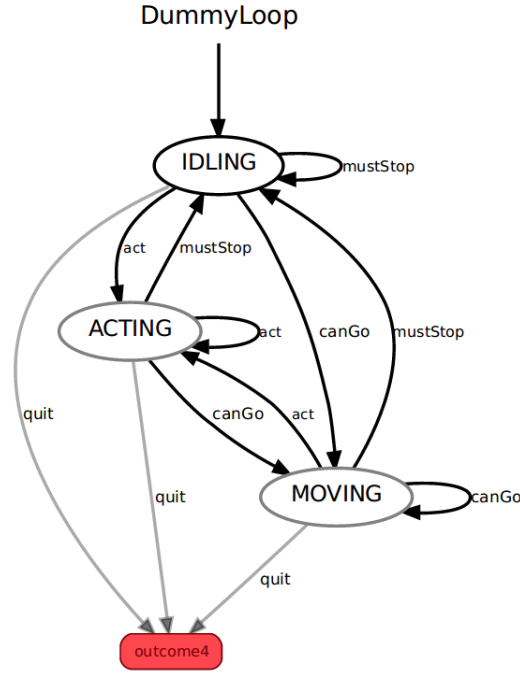


Figure 2: Robot basic automaton

2.2 Start-up

The whole simulation can be start with two launch files: one for starting the services and publishers, one for starting the robots. All important variables describing the simulation can be set in the launch files. It is extremely important to make sure that values set in this files are compatible (e.g. robots sizes and velocities...). After starting *roscore* node, *launch_services.launch* and *launch_three_robots* can be run, possibly with the `--screen --wait` option just to know when simulation can be ended. It is important to close simulation manually as it is the only way of saving the log files.

2.3 Possible paths

For purpose of this project several possible paths was made. They were designed to have multiple intersections and bottlenecks that could lead to collisions or deadlocks. Figure 3 presents paths used in experiments. Each path is read from text file containing consecutive points of robots route. Between every two points the robot

moves in straight line and space between this points is treated as one sector. Because of this, defined points shouldn't be too far apart and too close to each other. The general rule of thumb would be that the sectors should be longer than robot diameter and shorter than three diameters. The content of path file containing blue path from picture 3 is shown below:

```
0 -3
1 -3
2 -2
3 -1
3 0
3 1
2 2
1 3
0 3
-1 3
-2 2
-3 1
-3 0
-3 -1
-2 -2
-1 -3
```

2.4 Animation

Simulation results are saved by *plot_server.py* node under name specified in launch file. This log contains five columns of data: simulation time, x position, y position, sector number, velocity and distance since start of new lap. Then, this can be plot with provided *plot_position2.py* or *plot_position3.py* scripts working for two and three robots consecutively. Results are presented in form of graphs of sector number in time, velocity in time, distance in time and also as simple animation of robot movement on XY plane.

WARNING: Animation runs in a loop, starts over after it ends. Unfortunately during restart is can slightly shift some data caused it to go out of sync. Results can be presented correctly for first few loops, but later the cumulative error will cause the false view on system and can lead to false observation that robots do have collisions. It is recommended to restart animation after 2-5 loops.

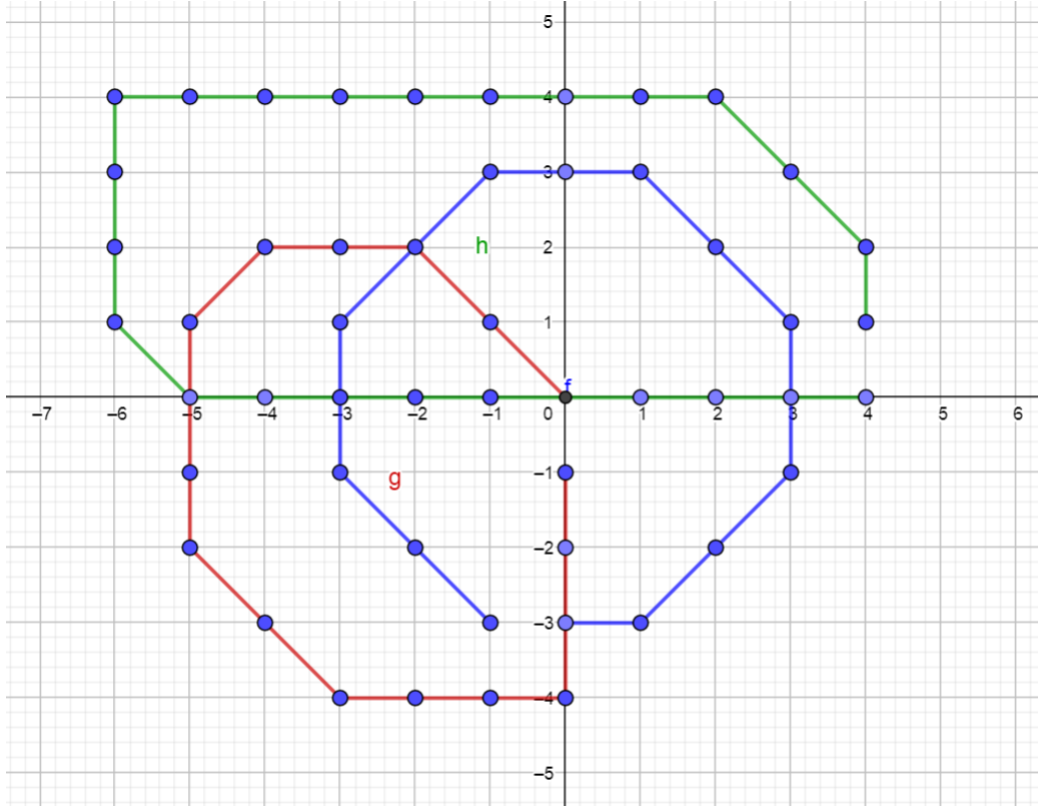


Figure 3: Designed paths

3 Results

Result presented here was obtained for simulating movement of three robots following paths visible on 3. Simulation parameters:

	Robot 1	Robot 2	Robot 3
path	blue	red	green
velocity	0.2	0.2	0.2
size	0.5	0.5	0.5

sim time	800
Δt	0.005

Since it is impossible to include animation in this report, it is recommended to download the results and run it locally. All files together with example results can be download from [??](#). Meanwhile the static plots are shown below on figure 4. First column shows sectors, second distance and the last one - velocity. From the first two

sets of plots it can be seen that each robot indeed have a moment of waiting for clear path. The last set of plots (velocity) shows that overall movement was quite continuous, without longer breaks.

This results combined with animated movement of robots shows that the system designed during this project works well and is good base for future expansions.

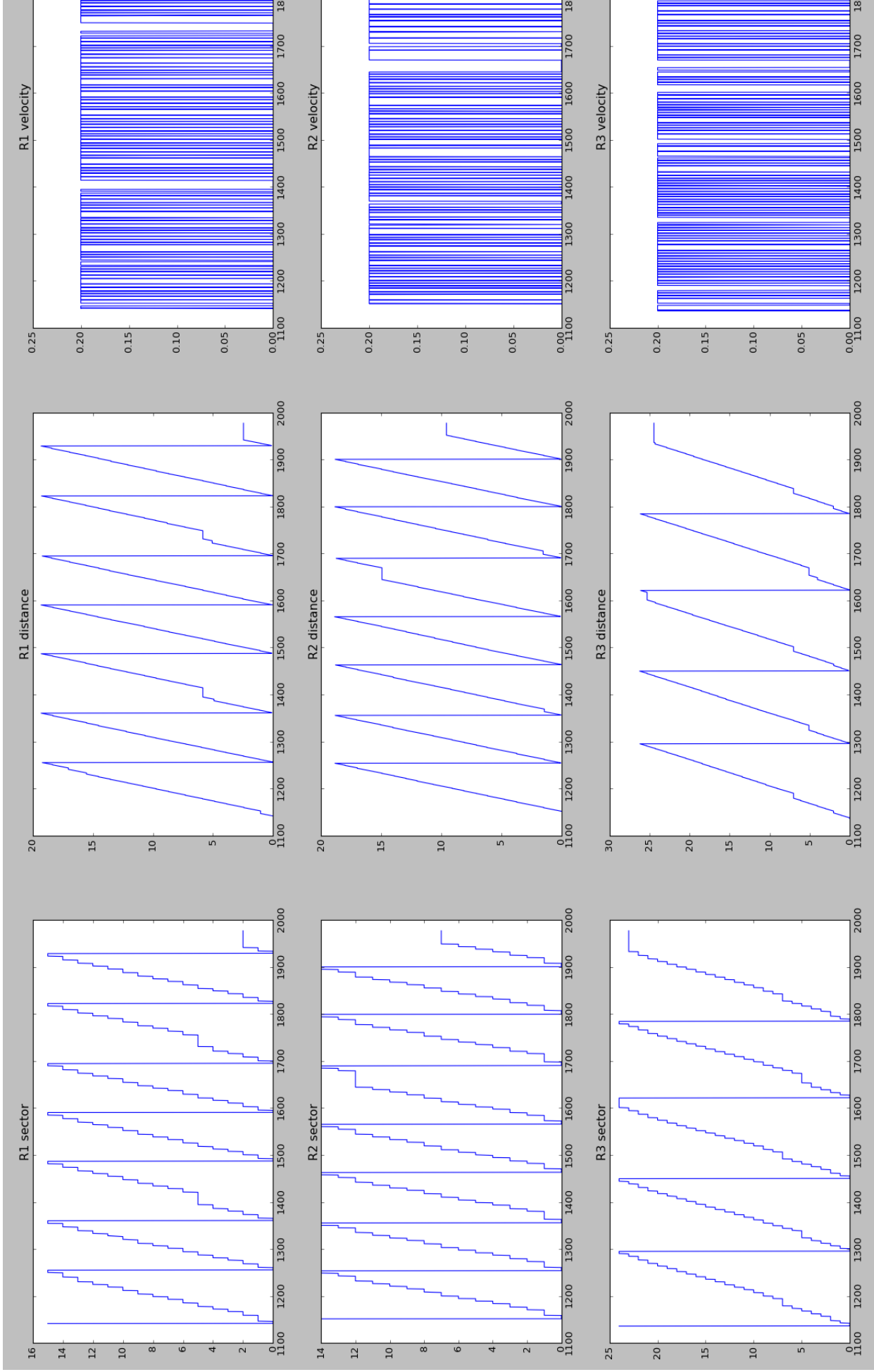


Figure 4: Result of example simulation

References

- [1] <http://wiki.ros.org/Robots/TurtleBot>
- [2] <http://www.ros.org/>
- [3] <http://wiki.ros.org/smach>
- [4] https://matplotlib.org/2.1.2/gallery/animation/basic_example.html
- [5] Paulina Porczyńska "Synteza sterowania systemu agentów mobilnych", 2017, engineering thesis
- [6] Source code of this project available here:
https://drive.google.com/open?id=1FqytDwp8MZY462hxFcFQBOFKz_1YQHK7