Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science (60 credits) in Computer Science with Specialization in Software Engineering - 15.0 credits - DVA423

SOFTWARE FOR SAFE MOBILE ROBOTS WITH ROS 2 AND REBECA

Kostiantyn Sharovarskyi
ksi19002@student.mdh.se

Examiner: Sasikumar Punnekkat
Mälardalen University, Västerås, Sweden

Supervisor(s): Marjan Sirjani, Ali Sedaghatbaf
Mälardalen University, Västerås, Sweden

May 20, 2020

Abstract

Robotic systems are involved in our daily lives and the amount of traction they have received is non-negligible. In spite of their sizeable popularity, the quality of their software is often dismissed. That may hinder an important property of robotic systems: safety.

The movement of mobile robots introduces an obvious safety concern. The collision of a robot with various things can lead to disastrous results. By amplifying the development process with formal verification techniques, one can decrease the probability of such failures. In order to facilitate close integration of safety assurance and the development process, we propose a method to develop safe software for ROS 2-powered mobile robots. We conduct a case study by going through all the proposed steps and reporting the results.

The case study focuses on a scenario in which mobile robots move from a starting position to the target position. Models of various ROS 2 components utilised in mobile robots are developed. Extensibility is a core property of our model. We show that it allows to verify both single- and multi-robot scenarios. Furthermore, that flexibility allowed us to model two path-finding approaches: one naive approach without collision avoidance and one efficient approach based on the A algorithm.*

The proposed method is tightly coupled with modelling, hence, the abstraction will lead to some mismatches between the model and reality. We report such mismatches by deploying the developed software to a simulation environment (i.e. Gazebo) and examining the behavior of the robot(s).

Table of Contents

List of Figures

Listings

1. Introduction

1.1. Motivation

Robotic applications are used across many different domains: space, manufacturing, aeronautics etc. Although the behavior of robots can be sophisticated, researchers claim that the development of such robots often does not incorporate system development processes and is based on the "ad-hoc development processes" [1]. Robotic software has various challenges in software architecture [2], cloud integration [3], software engineering processes [4] and other areas. In addition to traditional software engineering challenges, robotic software introduces an increased emphasis on the issue of safety.

These machines, robots, are relatively new, compared to cars and airplanes. And that is the reason why the body of knowledge for developing software in this area has less information available. Although the robotic technology is not that old, it can be used in complex sites, like manufacturing areas [5] and open space [6]. Complex sites introduce complex behaviors, which also substantially increases the complexity of the underlying software.

A lot of robotic applications have movement scenarios [7]–[9]. One of the first mobile robots as we know them now, were designed by W. Grey Walter [10] in late 1940s. They were known as 'robot tortoises'. They could move, recharge themselves, avoid obstacles, stroll around the garden. Software systems with such complex behavior have higher risks regarding the probability of functional faults. One of the safety-critical problems for the mobile robotic applications is the collision avoidance problem. If it is not taken seriously, this problem can lead to a large variety of hazards which may cause accidents that endanger human lives. One example of that is collision between a robot and a human being in a factory setting. This hazard could lead to an accident, and the consequences can vary based on the speed of movement and configuration of the robot.

Automotive domain, for example, is heavily regulated and has mechanisms to be proactive in regards to the safety of software systems. One of the regulations is the standard ISO-26262 [11]. The main idea of the standard is to strive for better safety and document all of the efforts. It also provides the process which is required in order to comply with the standard and which forces the developers to achieve safety goals.

The robotics industry has its equivalent of ISO-26262 which is ISO-10218 [12]. The standard for robots, in contrast to the automotive safety standard ISO-26262, does not have a dedicated section for software development. The software parts are reviewed in the same sections as the hardware parts. The standard is under review at the time of writing, so the software may receive more attention in the next release.

Software engineering techniques can be used as instruments for improving a vast number of software attributes: testability, cost of change, performance and more. However, for safety-critical domains, safety has the utmost importance. Safety-oriented architecture [13], fault-tolerance [14], verification [15], [16] and testing [17] can be used to improve the safety of robotic software. In this thesis, We focus on the verification approach to safety assurance of the robotic software.

We decided to narrow our scope from all robotic software to software for mobile robots. The main objectives for the above-mentioned kind of robots are various kinds of movement objectives. Our goal is to investigate how formal modeling and verification can be used to build safe software for mobile robots, therefore to make the movement safe, we must decrease the chance of collision.

Both formal verification and testing can decrease the probability of software failure. The major benefit of formal verification to testing is the guarantee that it provides with respect to the correctness of the program under analysis. Formal verification methods are already being applied to robotic software, but without considering safety implications [18], [19]. Robots interact with each other frequently, so the challenge is to formally verify that the interactions of the components in several robots are safe.

Model checking is one of the formal verification techniques used to find faults in many different applications [20], [21]. The basic idea of model checking involves building a model for the piece of software under study and verifying correctness properties on the defined model. We can build models for robotic systems and utilise formal methods to verify that these models conform to various properties. In the case of mobile robots, this technique may be utilised to verify that the system under study is safe.

Rebeca (Reactive Objects Language) is a modelling language based on the Hewitt actor model [22]. Rebeca is supported by a full-featured IDE called Afra [23], which embeds a powerful model checker called Rebeca model checker (RMC) [24]. RMC facilitates verification of various correctness properties based on the state-space analysis [25]. Rebeca has been successfully used to formally verify concurrent distributed systems [26].

Now, the robotic industry is much more developed than at the time W. Grey Walter built his 'tortoises' and there are various tools and approaches to build robots now. One approach is to build robotic applications on top of a reusable ecosystem, or middleware. Robotic Operating System (ROS) is a heavily utilized implementation of such middleware which strives to remove some of the burden from the engineers who build robotic systems.

ROS 2, the latest version of ROS, gives engineers a set of tools and software packages that allow rapid development of different types of robots. The central idea is decentralization by means of message passing. Instead of synchronous calls between components, ROS 2 provides an asynchronous development model. Noticeable differences from the previous ROS version include: new performance-focused underlying message transportation layer, changed syntactic interfaces, removal of centralized node and more.

In order to assure safety of mobile robots, we can utilise the model checking capabilities of RMC to check that the model of the software for the mobile robot is safe. Specifically, we target robotic software in mobile robots on top of ROS 2 middleware.

1.2. Research questions

This thesis aims to provide engineers with a method which would help to build safe software for mobile robots. In order to do that, we would need to understand how these software are developed, how to model the components of such software. And, finally, how to go from model to actual implementation and deploy that implementation on a robot in a real or an equivalent of a real environment. To satisfy those goals, we came up with the following research questions:

1. What ROS 2 elements are involved in robot movement scenarios and how can we model them in Rebeca?

   The ROS 2 framework has many types and abstractions. Some examples of ROS 2 elements are nodes, services, interfaces, topics, messages etc. The first problem to solve is to find which constructs are essential in mobile robots. The Rebeca model must be compatible with the robotic framework, so it is important to determine how mobile robots are implemented with ROS 2 and which key components these implementations use. Afterwards, we can model these elements in Rebeca. This activity helps us to develop an approach to model movement scenarios in Rebeca such that they can afterwards be implemented in ROS 2.

2. How do changes in ROS 2 affect the existing mapping rules from Rebeca to ROS?

   After modelling and verifying the movement scenarios in Rebeca, we need to have a working application in ROS 2. For that, we need to map Rebeca model to ROS 2 implementation. Some mapping rules for transforming Rebeca to ROS have already been devised in [27]. However, ROS 2 has major changes compared to the first version of ROS. So, many of the rules may not apply to the new framework. The goal is to investigate and devise updated mapping rules.

3. What problems can arise when deploying the ROS 2 code mapped from the Rebeca model?

   Even though the model of the software is verified by a model checker, this does not necessarily mean that the software will not fail. The model is a simplified version of some real-world interaction, so various details may be omitted for the purpose of abstraction. In some cases, such omitted details may cause the implemented model to be inoperable in the real world. In other cases, they can decrease performance, accuracy or degrade some other critical metric of the application. These issues will be visible after deployment. For instance, the model does not account for floor friction. And the deployed environment contains different types of floor surfaces. If two robots are approaching each other from different floor surfaces, the same amount of force applied using the actuators leads to different speed changes. In some cases, this could lead to a collision.

1.3. Contributions

The thesis work was focused on modelling components which are present in a mobile robot. It provides sophisticated reusable models for mobile robots.

We proposed an approach to model robotic applications by dividing modelled components into three groups: Environment, Robot Hardware and Robot Software. The Environment component(s) hold all required information regarding the model of the real world that are important for other components. Additionally, they provide functionality for the Robot Hardware to model interaction with the real world. Robot Hardware uses the Environment in order to model the actual sensors and actuators which are present inside the robot. Both Environment and Robot Hardware groups of components can be reused for many scenarios which are not limited to the one discussed in this thesis. They were designed with reusability in mind. The group of components representing the actual software that the software engineer is responsible for is called Robot Software.

Also, we modelled a representation of Environment for robotic applications. The Environment group was represented by Map reactiveclass. The Map holds the information about all the obstacles and the robots which are present in the model. Also, the Map provides functionality to move a robot from one position to the other. This move operation includes an assertion for no collisions. This ensures that the Rebeca Model Checker verified safety of robot movement. Although the Map is designed for the use-case of a mobile robot, it could be built upon for other use-cases.

The Robot Hardware model part included two robotic components: movement actuator and a laser sensor.

We modelled a detailed reusable model of a Laser sensor hardware component. The laser sensor uses the knowledge provided by the Map reactiveclass to get all of the obstacles. Then, it models each individual laser ray and identifies how these rays would interact with the obstacles. The main idea was to provide the same exact data structure as one would use in the actual ROS 2 code. This data structure is an array of distances to the nearest obstacle that each individual laser ray has encountered. Ultimately, this allows future code generation to create the code as close to ROS 2 as possible. However, the code generator was not implemented in the context of this work.

Additionally, we propose a model of robotic movement actuator. The movement command model included such concepts as rotation and position. The model accepts the movement commands and executes them periodically. Periodicity was introduced in order to model the delay between the command and the actual movement. Regarding the timing specification, we observed that the laser sensor fires multiple times before the robot movement is visible, hence we added that to our model. This actuator can be reused for any model that involves generic robotic movement.

The approach with three groups enabled the flexibility of our model. We were able to provide two different Robot Software models. Firstly, we produced the Naive implementation. That model did not react to laser sensor data in any way. The behavior was to move to the target in any way. We showed that the model verification fails. Then, we modelled the A* pathfinding algorithms. The reaction to the laser sensor data was modelled, meaning the Robot Software inferred the position of the obstacle by looking at the distances to the obstacles and positional information (coordinates and rotation). Then, based on the internal map with the detected obstacles, the model calculated the path by using the A* pathfinding algorithm. The algorithm relies heavily on the list data structure and the tree data structure in order to produce a path. Therefore, we had to overcome the absence of the data structures by modelling them based on operations with arrays.

We were interested in the interaction between robots too. The Environment is shared for all robots. Therefore, to model multiple robots we need to instantiate a separate Robot Hardware and Robot Software group for each robot. We could not model interactions between two robots implementing A* pathfinding algorithm due to the state explosion problem. We decided to model interactions between one robot which implements A* and the second one which follows a predefined path. This shows interactions between the A* implementation and moving obstacles.

Additionally to the models, we devised how to verify both safety and the correctness of the model. This was achieved by verifying that each movement did not result in a collision and that the target was reached.

After modelling, we manually mapped the model to ROS 2 code and proposed the mapping rules for such transition. The major difference between our mapping rules and the ones presented in RoboRebeca approach [27] is the absence of components that interact with a central server. The robots modelled in our work communicate neither with each other nor with a central entity.

Lastly, we verified our approach by deploying the mapped ROS 2 artifacts to a simulation environment. During deployment, we found that we have not modelled the robot dimensions. That led to collisions as the pathfinding algorithm did not account for additional space required by robots dimensions. Additionally, we had to manually add logic for handling real valued coordinates opposed to integer values coordinates used in the model. Physical interaction which we were not able to model in Rebeca was discovered: sharp speed changes led to the robot model rolling over. We had to add stabilization mechanisms and mechanisms to smoothen out the speed changes.

The key property of our work is the details which we were able to model. Opposed to modelling the pathfinding algorithm only (e.g. done by Germanos and Secco [28]), we also included the process of locating the position of obstacles from laser sensor information.

Final Rebeca code is presented in Appendix A.

2. Background

2.1. Rebeca

Rebeca is a modelling language based on the actor concept. The syntax is similar to Java and C#, so engineers who work using these languages will easily understand the code. The main components are rebecs. Rebecs are the instances of ***reactive classes***, and are the Rebeca equivalent for actors. Rebecs can communicate with each other by asynchronous message passing.

Structurally, each Rebeca model consists of three parts: (1) a possibly-empty set of environment variables, (2) reactive class definitions, and (3) a main block, in which the reactive classes are instantiated i.e. rebecs are defined. Each reactive class consists of the following elements:

- knownrebecs

  To be able to send messages to another rebec, a rebec needs to add the target rebec to its list of known rebecs.

- statevars

  The state variables represent the current state of the rebec at each moment of time. They may be manipulated by message servers, constructors or methods. They are central to the property verification performed by the RMC, as together with the message queues they are representing the state of the model.

- constructor

  Similar to object-oriented languages e.g. Java, a constructor in Rebeca is responsible for initializing the state variables of a reactive class.

- message servers

  Message servers model the message-passing behavior of a rebec. A rebec can send messages to its own message servers and the message servers of its known rebecs. The messages will be processed asynchronously. Message servers are the only way of interaction between rebecs in Rebeca.

- methods

  Methods can be used to perform mathematical computations. In contrast to message servers, a method can return a value and is called synchronously by the message servers or methods of the same reactive class.

- message queue

  A queue to hold the incoming messages before being processed by message servers. The size of this queue has an upper bound, which needs to be specified beside the name of the reactive class.

An example of Rebeca model in Listing 1 shows a rebec *send*, as an instance of reactive class *Sender*, sending a number to rebec *rec*, which is an instance of reactive class *Receiver*.

```
reactiveclass Receiver(10){
  msgsrv receive(int i) { }
}

reactiveclass Sender(10){
  knownrebecs {
    Receiver receiver;
  }

  Sender() {
    self.sendThings();
  }

```

```
  msgsrv sendThings() {
   receiver.receive(5);
   self.sendThings() after(1);
  }
}
main{
 Receiver rec():();
 Sender send(rec):();
}
```

Listing 1: Recurrent message passing and behavior in Rebeca

Rebeca model checker (RMC) [24] examines all possible states of the Rebeca model in order to prove or disprove a claim about the correctness properties. Each state consists of the two following entries:

- values of the state variables of each rebec
- content of the message queue of each rebec

Based on these entries, RMC verifies that each valid combination of them is supporting the correctness property which is being verified.

Timed Rebeca (or TRebeca) is a timed extension of Rebeca, which allows modeling of timing constraints. To model the flow of time, it adds the following primitives to Rebeca: (1) *delay*(), which models computation delays, (2) $after()$ for modeling message transmission delays, and (3) *deadline*() for specifying operation deadlines. Complimenting the language constructs, RMC provides support for verifying the timing constraints specified through these primitives.

2.2. ROS 2

One way to develop mobile robotic applications is to use the Robot Operating System (ROS). ROS is a software framework consisting of libraries and tools which focuses on robotic applications [29]. The central concepts of ROS are topics and messages.

Each topic can send and receive messages of a certain type. The receiving functionality is supported by programmatically subscribing to the topic. The robot components subscribe to topics which are populated by other components publishing messages. This model allows introducing decoupled components as each of them does not need to have knowledge about the others.

The latest ROS version is ROS 2 which introduces large amount of changes to the framework. The core difference between versions is the new Data Distribution Service (DDS) transport layer [30]. The transport layer introduces performance improvements, reusability and has a DDSI-RTPS (DDS-Interoperability Real Time Publish Subscribe) protocol which works well with the publish-subscribe nature of ROS. The adoption of DDS helps the developers get rid of the "master node". In ROS 1, the role of master node was to allow discoverability and communication between other components. The change improves the independence of different ROS components interacting in the ecosystem.

Another feature based on the new transport layer is the QoS (Quality of Service) feature [31]. It allows subscribers and publishers to specify such settings as queue depth, message durability, deadline duration, message lifespan duration and others.

Other notable changes include: new build tools (i.e. colcon and ament) introduced to replace the previous catkin tool, migration from XML-based launch configurations to python-based launch configurations for flexibility purposes, removal of central configuration node and a big amount of syntax changes.

A simple ROS 2 subscriber which logs the message received is presented in Listing 2, assuming the package *package_name* publishes a message $Move$ with x and y coordinates on topic */move*.

```
// ROS 2 library
#include <rclcpp/rclcpp.hpp>

// Include the Move message contract
#include "{package_name}/msg/move.hpp"
```

```

class Subscriber : public rclcpp::Node
{
public:
  Subscriber() : Node("subscriber")
  {
    _moveCmdSub = this->create_subscription<Move>("move", rclcpp::QoS(10),
        std::bind(&Subscriber::OnMoveCommand, this, std::placeholders::_1));
  }
private:
  rclcpp::Subscription<Move>::SharedPtr _moveCmdSub;

  void OnMoveCommand(const Move::SharedPtr moveInfo)
  {
    // Log the message
    RCLCPP_INFO(this->get_logger(), "Received Move: %.3f, %.3f", _target->x,
        _target->y);
  }
}

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);
  auto node = std::make_shared<Subscriber>();

  while(rclcpp::ok())
  {
    rclcpp::spin(node);
  }
  rclcpp::shutdown();
  return 0;
}
```

Listing 2: ROS 2 subscriber example

2.3. Gazebo

ROS 2 has a lot of value in its ecosystem. The middleware introduces a huge variety of developed components designed for reuse, robots that support ROS out of the box and simulation environments. Simulation environments are useful for scenarios when it is problematic or resource-intensive to deploy the software to actual robots. They provide an environment which is close to real world in certain aspects.

Gazebo - a simulation environment [32] supported by the organization that created and supports ROS - OpenRobotics. The simulation environment adds ability to model robots, add sensors and actuators and place the robots in the simulated physical world (see Figure 1). The physics engine then handles all interactions and graphically shows what happens in real time.

The integration of the Gazebo simulation environment with ROS 2 is vital. The sensors and actuators that the simulator provides have ROS 2 interfaces which allows robots to interact with them as if they are deployed with actual hardware. For example, simulated movement actuator provides the same topic as industrial robots. It can change the velocity and other properties of a robot dynamically. This allows developers to test the ROS 2 software logic in the simulation environment.

Without such a simulator, each change in robotic software may produce the need to deploy it on a robot in the real world. This will introduce a noticable time lag between developing software and seeing it work in practice.

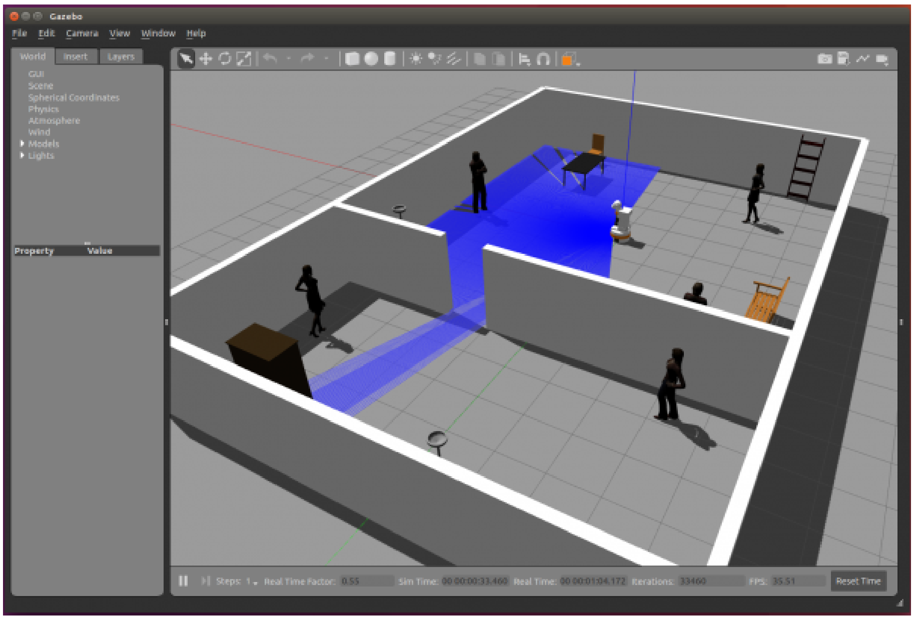


Figure 1: An example of Gazebo simulation from RobMoSys[33]

3. Related Work

3.1. RoboRebeca

This thesis work is focused on the same problem addressed by the "RoboRebeca" framework introduced by Saeid Dehnavi [27]. He released the first research publication showing how to model generic ROS robots in Rebeca and verify their properties.

RoboRebeca considers a generic robotic application with a command center. This entity dispatches commands to robots and tracks their progress. For these types of applications, Dehnavi proposed a way to model them in Rebeca and an automatic mechanism to map them to ROS 1 programs. The core principle of the automatic mapping is to utilise attributes for specifying metadata which is important for ROS, but is ignored in the Rebeca model. For example, the reactive classes that indicate robot software are marked with specific attributes to convey that information to the mapping software.

Current thesis contribution also concerns robotic software, but operates with mobile robots and focuses on modelling more. Additionally, it is assumed that there is no command center and multi-robot behavior is modelled as interactions between two independent robots.

Moreover, extra attention is brought to the modelling phase, which strives to replicate the structure and behavior of a mobile robot. The results in the thesis are less generic, but have more relevant details for mobile robots.

3.2. Verification of geometric properties

Banusić et. al [34] introduces PGCD - a way to program robotic applications with a proprietary ROS-based runtime and a PGCD verifier. The structure of the programming model (P) allows to verify the Geometric (G), Concurrency (C) and Dynamic (D) properties. That means, that the platform allows checking communication issues (e.g. deadlocks) and collisions in concurrent environment by design. Verification of geometric properties is powered by instructing the developer to describe how the robot and the components use local geometric space (authors call it the 'footprint'). Then, the verifier has the ability to check overlaps between such footprints in different components.

Comparing our work to PGCD, we separated the modelling and deployment phase by connecting them with a mapping transition. Banusić et. al, on the other hand, proposed a framework which does not need such mapping and forces the developer to adhere to a model that allows built-in verifications of many properties. These different approaches have their own advantages and disadvantages. The PGCD helps the developer by removing the necessity to switch between the model and the codebase. However, it could be argued that this approach limits the developer to PGCD only, increasing the cost of a possible switch of modelling technologies in the future.

Geometric properties verification is visible on both this contribution and the considered paper. Our attention to modelling ROS 2 components interfaces requires modelling geometric properties and interactions. In spite of the fact that we did not use the geometric properties in the verification statements, they were used throughout the model in different modelled components. This way, the model captures more details about the behavior of the robot, allowing more precise code generation in the future. Additionally, PGCD supports integration with ROS 1, whereas our focus is on the latest ROS 2 release.

3.3. Verification of mobile robots

Experimental verification. In 2015, Kowelczyk et. al [28] worked on a mobile robot with collision avoidance feature. Similarly to this work, they looked at a mobile robot with a laser sensor and used that to build a map. Consequently, that map was used to implement the collision avoidance feature. ROS 1 features were used to implement such behavior. To verify the robot, experimental verification in form of a simulation was performed. Researchers put predefined circular obstacles and observed the behavior of the robot.

In contrast to that, this thesis aims to apply modelling techniques and formal verification to the same set of problems. The approach described in this contribution can be classified as 'model-first', meaning the model of the behavior is designed and developed first. Only when the iteration of the model is completed, the deployment will be attempted.

Formal verification of navigation. Formal verification of robotics navigational algorithms was also performed by a Germanos and Secco [35]. BUGs algorithms (inspired by ant movement) were modelled and formally verified. Two different BUGs implementations were also compared by the number of states and verification time in order to evaluate their performance.

While the paper does not model actual robots, the premise of mobility is pertained, so the idea of verifying movement is the same. Comparing our approach, they assumed the knowledge of the environment is provided and did neither model nor verify the obstacle detection, but the navigational algorithm only. Additionally, the modelling did not contain sensors and actuators which puts it on another level of abstraction. Our work attempts to include the structure of robot, sensors and actuators in the model.

Verifying a firefighting robot. A more recent successful attempt to develop and verify mobile robots was performed for a firefighting robot [36]. There, an autonomous robot was tasked with putting out fires by utilising color sensors, sonar sensors and a magnetometer. A special arena and a specific task sequence was designed in order to drive the formal verification and testing. The authors translated the algorithm into an UPAAL model and verified a large number of properties: safety properties, deadlock prevention properties and liveliness properties. Moreover, they built the arena and tested it experimentally too.

Similarly to the firefighting robot, in this work we developed a scenario, created a model for it, formally verified it and deployed to a real-world like environment. However, our additional focus was on ROS 2 component modelling and integration. The UPAAL model of the reserachers checked the overall behavior of the robot, omitting the details like sensors and actuators. The UPAAL model is therefore very specific for a certain robot. Our work attempts to model sensors and actuators independently of the robot, so that the models of such components can be reused.

4. Method

In order to conduct this research and provide proper answers to the research questions defined for the thesis, we had to acquire knowledge and make appropriate decisions about the following items:

1. the type of research method
2. the robotic middleware
3. the simulation environment
4. the formal modelling and verification tool

4.1. Research method

The research area of mobile robots is complex. In order to capture and document the important nuances, we need to see how the issues we are raising are evident throughout the development process. In order to capture that information, we decided on a qualitative research method.

The goal of the thesis is to propose and evaluate a method for developing safer software for mobile robots, so we decided to base our research methodology on the Action research methodology. Håkansson describes Action research methodology [37] as one aimed to improve how people solve some specific problems. Runeson et. al place action research method as a variation of a case study [38]. They also define case study as a research method "aimed at investigating contemporary phenomena in their context". In our case, that phenomena is the development process of safe mobile robots.

More specifically, the case study research method allows us to accumulate knowledge about mobile robots, ROS 2 and verification software (Rebeca) throughout the whole period of thesis work. We performed several case studies on robot movement scenarios, their formal modeling and safety verification. These case studies are reported in Section 7.

The process for conducting the case study is presented in Figure 2. The following steps were conducted:

1. Research on ROS 2
2. Build simple mobile robot in ROS 2
3. Document the ROS 2 components
4. Research on Rebeca
5. Build the model in Rebeca
6. Document the Rebeca structure
7. Research mapping for ROS 1
8. Map Rebeca model to ROS 2
9. Deploy the ROS 2 model
10. Document mapping differences for ROS 2

Firstly, we research the way mobile robots are built in ROS 2. In order to do that, we try to build a simple mobile robot. Afterwards, we document which ROS 2 components were required in such a robot. Based on these components, we create a Rebeca model. After documenting the structure of this model, we check the existing mapping rules for ROS 1 and devise a way to map our model to ROS 2. Afterwards, this mapped model is deployed to a simulation environment, we document the mapping rules. There are multiple places where we go back in the process if something goes wrong, as indicated in Figure 2.

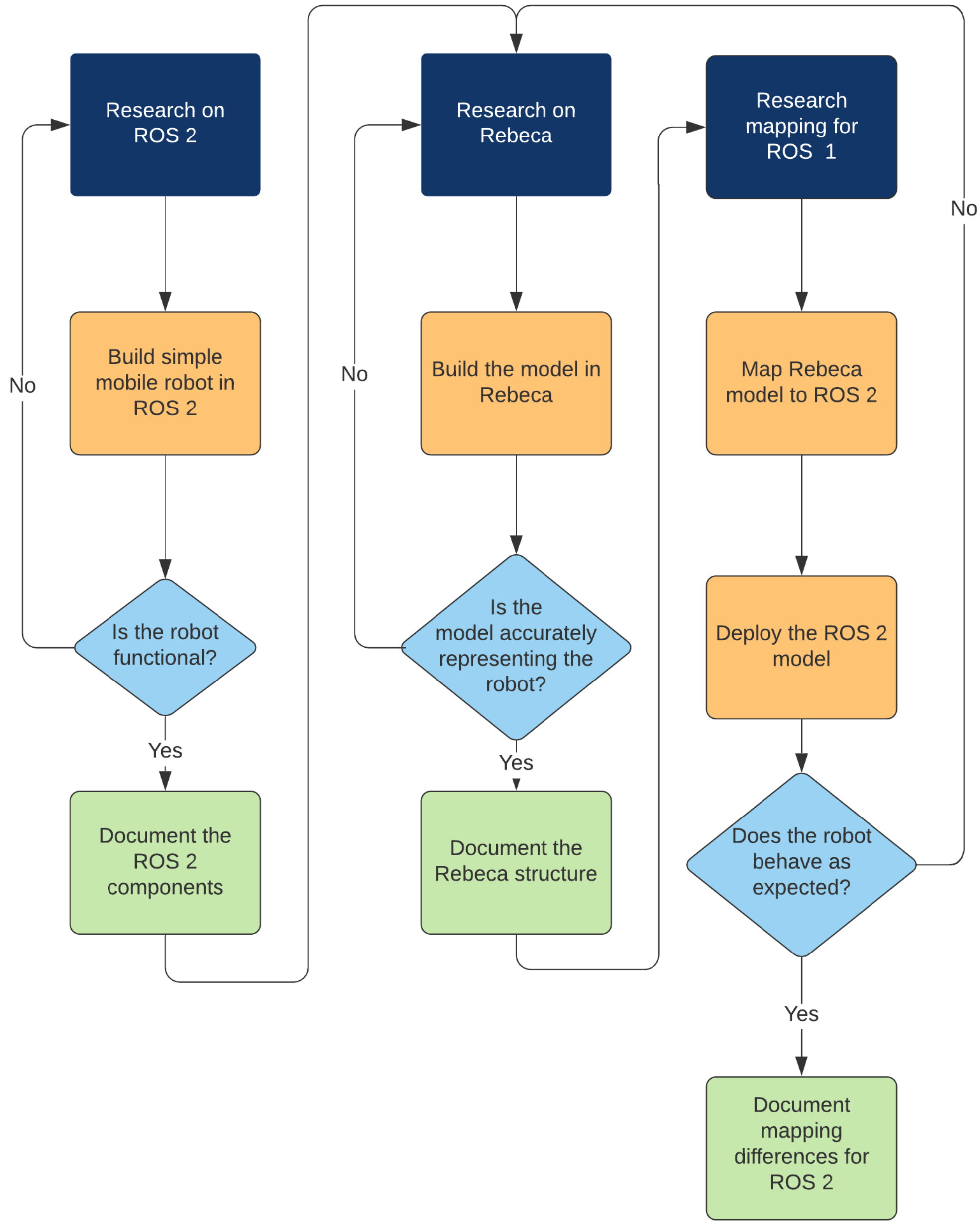


Figure 2: Case study process flowchart.

4.2. Robotic middleware

The Robotic Operating System (ROS) family of middleware already has been under research for purposes of modelling and verification [27]. In order to build on top of that knowledge, we decided to proceed with this type of middleware.

The robotic middleware used in our method was ROS 2. The ROS 1 robotic middleware is utilised in many projects, some of which are described in Section 3 "Related works". The ROS 2, however, is the next successor middleware to the applications which have ROS use-cases. The last planned distribution for ROS 1 is ROS Noetic, but the ROS team at OpenRobotics claim that OpenRobotics "can't commit to continue investing resources into future ROS 1 releases past Noeti" [39]. ROS Noetic planned end of line is 2025, so the companies using ROS have a strict deadline for migrating to ROS 2. Therefore, more and more developers will start using the new version of ROS.

4.3. Simulation environment

To answer research question 3, regarding the robot deployment, we need to select the deployment target, i.e. a simulation environment. We decided to choose Gazebo due to its tight integration with ROS 2. Another important benefit of Gazebo is that it does not require source code changes of the robot software. In fact, we can model and implement the robot, independent of the simulation environment. During deployment, we only need to run gazebo-managed ROS 2 resources at specific addresses that robot software is expecting.

4.4. Modelling and verification

The modelling language used in this research is Rebeca and the verification tool is RMC (Rebeca Model Checker). ROS 2 is based on asynchronous message passing, and Rebeca is well-suited for modeling this kind of communication. Accordingly, Rebeca models can closely resemble the programming model of the robotic applications. Rebeca is also based on actor model, which can align well with the distributed architecture of ROS 2 programs in which there is no central controlling entity.

5. Ethical and Societal Considerations

Current work is scoped to the modelling and verification software development practices. There is no direct connection to people, communication, economic issues involved, therefore, current contribution does not involve any ethical issues.

Regarding the societal considerations, mobile robots introduce many hazards that are related to property or health damage. In the context of this thesis, a method to decrease the probability of safety hazards was proposed, which may be beneficial in safety-critical scenarios.

6. From Model to Deployment: Proposed Method

As indicated in Figure 3, the method proposed in this thesis to develop safe software for mobile robots includes the following steps:

1. **Define the movement scenario.** This step represents defining the core high level objective of the robot, i.e. the movement tasks it is supposed to perform for the users.

2. **Investigate the ROS 2 components of mobile robots.** In order to adequately model the problem space, one should be familiar with the typical architecture of the real world mobile robot being modelled. To gain that knowledge, we have investigated the components that are required for mobile robots to perform the tasks defined in the previous step.

3. **Develop the Rebeca model for the scenario.** Keeping the architecture of the system in mind, we carefully model the behaviour and structure of a mobile robot in such a way that new architecture elements can be introduced later and that multi-robot verification is possible.

4. **Verify properties on the model.** To make sure that the modeled scenario is safe and will not lead to a collision, in this step we define a set of safety properties for the model and verify them using RMC. Additionally, we make sure to include the correctness properties in order to be confident that our model performs the expected task.

5. **Map the Rebeca model to ROS 2 implementation.** Having the Rebeca model developed and verified, we transform the model to its ROS 2 equivalent using a mapping algorithm.

6. **Deploy ROS 2 implementation to a robot.** The ROS 2 software generated in the previous step is then deployed on the Gazebo simulation environment and the issues not found during the modelling and verification phase are documented.

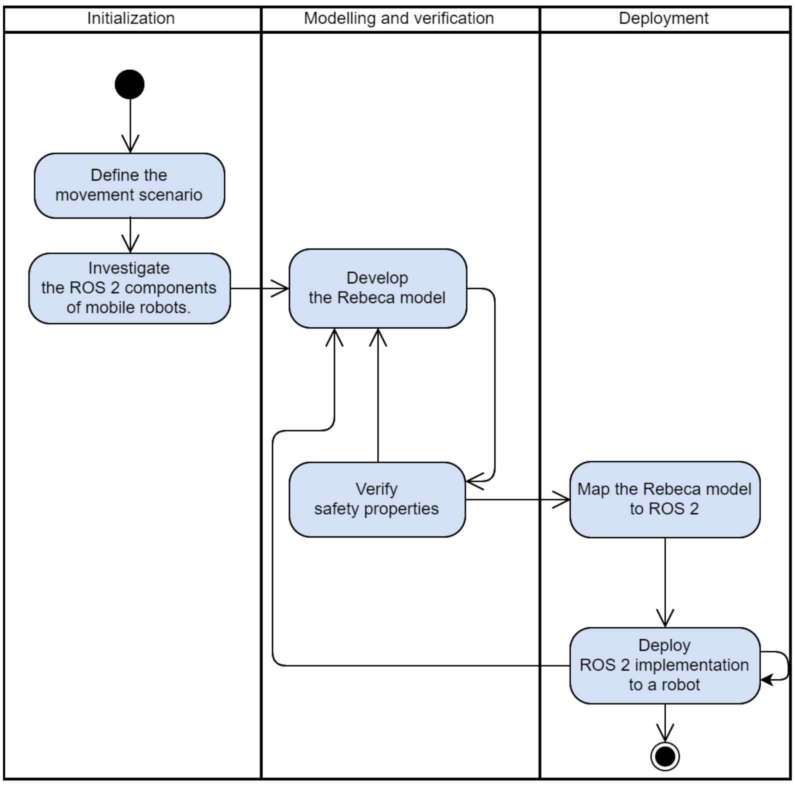
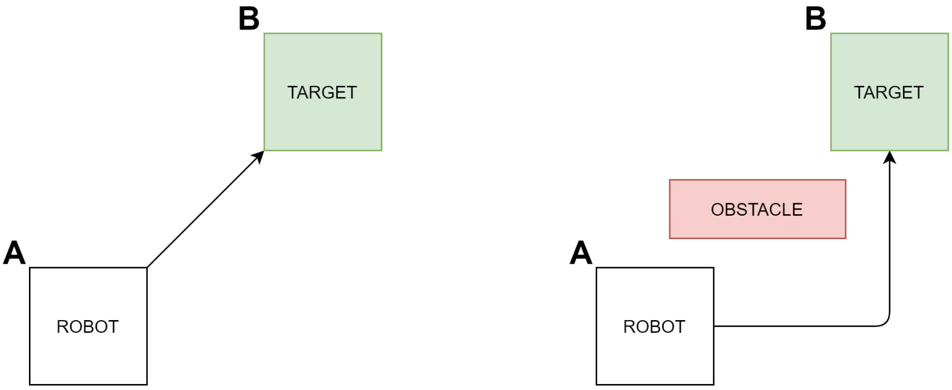


Figure 3: Activity diagram for the method to develop safe software for mobile robots.

In the following, we elaborate each of these steps by illustrative examples.

7. Movement scenario

In this section, we introduce the movement scenario we decided to base our case study upon. There are two points: A and B. The point A is the starting position of the robot. The task of the robot is to move to point B (see Figure 4a. The simple scenario is straightforward and poses no safety risks. However, as soon as we add one obstacle (see Figure 4b), the possibility of a collision starts to pose a safety risk.



(a) Simple scenario (b) Obstacle scenario

Figure 4: Movement scenario illustration

The scenario contains static obstacles only. The dynamic obstacles can be represented by adding more robots to the scenario which are all moving from one point to another. The high level goal of the thesis is to develop robotic software that accomplishes the task to reach the target without accidents.

8. A Robot Model in ROS 2

As discussed earlier, the main building blocks of ROS 2 applications are nodes, topics, services and the messages passed between these blocks. In the next sections, we are going to introduce the main components built on top of these blocks which are necessary in the movement scenarios we are considering.

8.1. Movement command

The ROS 2 component which is essential in movement scenarios, is a ROS 2 topic which accepts movement commands. Usually, it is called */cmd_vel*, which can be interpreted as 'command velocity'. That topic accepts a message of type *Twist* which consists of two 3D vectors: a linear and angular speed [40] the robot is expected to maintain. A ROS 2 node should listen for messages in the */cmd_vel* topic. It translates the expected angular and linear speed into rotational speed of the wheels by applying appropriate amount of force using specific actuators. However, the node responsible for handling */cmd_vel* messages is usually provided by the manufacturer of the robot. In fact, the robot programmer should not be concerned with implementing the actual mechanical movement.

8.2. Odometry topic

Any path-finding algorithm requires knowledge about the current state of the robot. In order to support that requirement, we assume that the robot has a way to determine its position and angle relative to the predefined points A and B. In other words, the robot should support odometry. Odometry [41] is a way to use several sensors to estimate positional and rotational changes of robot's position over time.

In ROS 2, this can be achieved by subscribing to the */odom* topic. This will allow the robot to receive periodic updates about current position. These messages should be produced by a node which knows the exact structure of the hardware and utilizes it to produce the estimation. Similarly to the movement command topic, that functionality is often provided by the manufacturer.

8.3. Laser sensor

Safe movement means that any obstacle should be avoided in order to prevent collisions that could result in accidents. The robot has to have an ability to detect obstacles in time to prevent possible collisions. We have implemented obstacle detection by using laser sensors.

The laser sensor works by spraying rays of light from the front side of the robot in different directions (see Figure 5). The laser sensor can be identified by the starting angle, the amount of rays it is producing and the distance between rays. Knowing the positional number of the ray, we can easily identify the angle. The output of each ray is either the distance to the nearest obstacle, or infinity if no obstacle is observed until the limit of laser's reach.

ROS 2 provides laser outputs in a topic, we decided to call that topic */laser_scan*. The manufacturer's ROS 2 node uses the hardware and produces messages of type *LaserScan*. These messages contain all the information needed to identify each ray and its output described earlier.

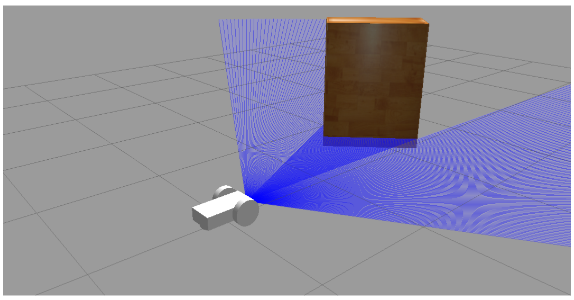


Figure 5: Blue rays of the laser sensor in Gazebo simulator.

8.4. Move topic

By providing means to control a robot in real time, we can make the robot flexible. Since the robots are tasked with some high level objectives, they need to receive those objectives in some way to change their behavior without recompiling source code.

To achieve this goal, one can introduce a separate topic for robot commands. We are studying mobile robots, therefore the main high level objective is to reach a specific position in the real world. That can be implemented in ROS 2 by introducing a */move* topic. The robot listens to the messages published there and changes its path-finding strategy to accommodate for the new goal.

The difference with the Movement Command topic described in Section 8.1 is that the Move topic handles high-level goals. It registers the final target of movement, in contrast to the Movement Command which handles low-level velocity changes.

9. Modeling Robots in Rebeca

The view of our Rebeca model is presented in Figure 6. To separate the hardware model from the code which is under control of the software developer, we decided to build the model based on three parts: Environment, Robot Hardware and the Robot Software. The respective model parts have one or more reactive classes implementing desired behavior. In the following subsections, we present how we modelled the specific components and interactions in Rebeca.

9.1. Publish-subscribe mechanism

ROS 2 utilizes not only direct message passing (between a service and a consumer), but also the publish-subscribe mechanism (using ROS 2 topics). For example, Robot Software subscribes to sensor information which is generated in Robot Hardware. However, Rebeca supports direct message passing only.

In order to provide support for modeling the publish-subscribe mechanism in Rebeca, we have defined two reactive classes *Publisher* and *Subscriber* (see Listing 3). Subscribe can be modelled as a single message pass to a message server *messagesTopic* representing a topic. Then, that message server will be responsible for publishing messages to the caller.

ROS 2 publishers often publish messages periodically with a certain rate. In order to model that, the *Publisher* starts a recursive message server which periodically (utilizing features of Timed Rebeca) sends messages to the consumer of the topic and then queues itself with a delay, using function $after()$.

```
reactiveclass Subscriber(2) {
 knownrebecs {
  Publisher p;
 }
 Subscriber() {
  subscribe();
 }

 void subscribe() {
  p.subscribe();
 }

 msgsrv messagesTopic(int message) { }
}

reactiveclass Publisher(2) {
 msgsrv subscribe() {
  self.periodicSend((Subscriber) sender);
 }

 msgsrv periodicSend(Subscriber s) {
  s.messagesTopic(1);
  periodicSend(s) after(5);
 }
}
```

Listing 3: Publish-subscribe relationship in Rebeca

9.2. Environment

The Environment part of the model in Figure 6 represents everything not under direct control of the robot. It includes any entity that the robot can interact with via sensors and actuators. Then, Robot Hardware part models the interfaces of the sensors and actuators based on the information available in the Environment part and provides the functionality to the Robot Software part. The Robot Hardware and Robot Software reactive classes are logically grouped into Robot Internals to showcase that a single robot is represented by two reactive classes. To model multiple robots

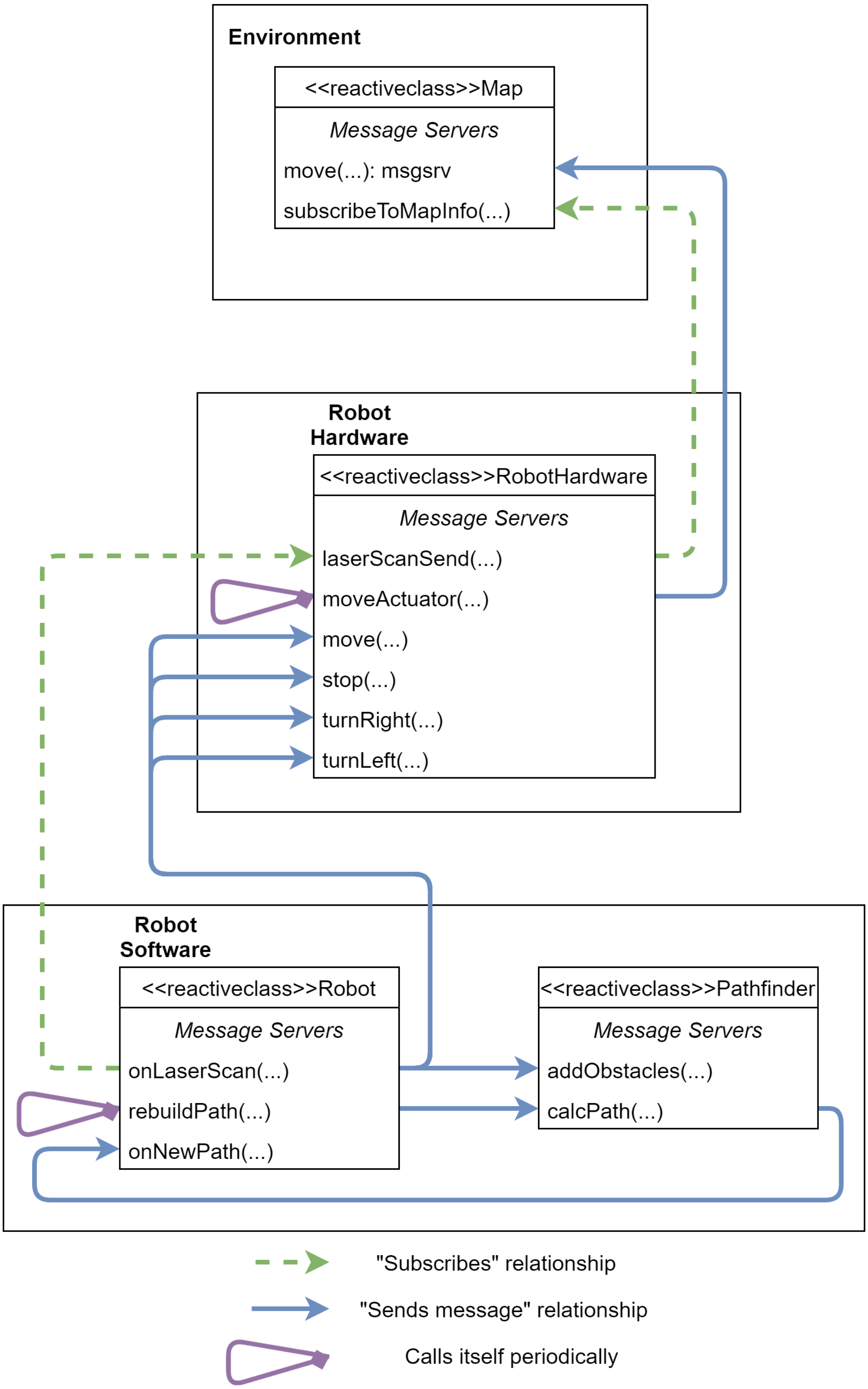


Figure 6: The components view of the mobile robot model.

interacting with the same environment, the engineer would only need to create copies of reactive classes in Robot Internals group which would interact with the same Environment.

To be more specific, in case of mobile robots, the Environment can be represented by a Map reactive class. The Map contains the information about all obstacles that the robot can collide with and all other robots present. From the Map's perspective, all the robots and obstacles are the same, so we will call both of them "obstacles". The main responsibility of the map is to store the obstacle information, present that obstacle information to Robot Hardware and give the Robot Hardware the ability to indicate movement (which means changing the obstacle information).

9.3. Robot Hardware

As soon as we modelled the Environment as a Map, we have all required information and functionality to model the sensors and actuators.

Movement. Starting with the movement actuator, we identified the following required operations:

- turnRight
- turnLeft
- startMoving
- stop

Identified operations are the public interface of the Robot Hardware reactive class. The robot will use the mentioned operations to move around.

To model the delay between the actual movement and the command execution, the movement is modelled as a periodic event which is triggered on a timer. Upon each 'tick', the message server checks the current state (the rotation, whether the rotation needs to change, the position, whether the robot is stopped or moving) and decides on the next action (see Listing 4). And then, if the position changes, it sends that information to the Map reactive class.

```
msgsrv moveActuator() {
 if(nextRot != rot) {
  prevRot = rot;
  rot = nextRot;
 }
 else if(moving) {
  int dx, dy;
  if(rot == NORTH) {
   dx = 0;
   dy = 1;
  }
  else if(rot == EAST) {
   dx = 1;
   dy = 0;
  }
  else if(rot == SOUTH) {
   dx = 0;
   dy = -1;
  }
  else if (rot == WEST) {
   dx = -1;
   dy = 0;
  }

  self.prevPosX = posX;
  self.posX += dx;
  self.prevPosY = posY;
  self.posY += dy;

```

```
    mapRebec.move(prevPosX, prevPosY, posX, posY);
  }
  ...
```

Listing 4: The message server in the Rebeca code modelling the Movement actuator

Laser sensor. To function properly, the robot needs the ability to interact with its sensors. Under our assumptions, the robot requires the laser (or ray) sensor only. The next modelling task is to represent the distance to the nearest obstacles by using the information available in the Map reactive class.

Mathematically, a ray is a straight line. In the mobile robot's case, the ray is identified by the starting point and angle relative to the robot's rotation. The straight line can be represented by a linear equation, as indicated in equation 1.

$$y = kx + b \tag{1}$$

where:

k = representation of angle
y = y coordinate
x = x coordinate
b = vertical bias

From mathematics we know that the k variable is the tangent of the angle between the straight line and the positive direction of abscissa axis (X-axis). So, by changing that parameter from $tan(startingAngle)$ to $tan(startingAngle + delta * rayCount)$, we can represent the whole range of rays expected in a laser sensor. Of course, under the assumption that the robot is at the starting point $(0, 0)$ and it is facing north.

Rotational component of robot's position should influence the rays too. By applying a rotation with the angle corresponding to the robot's rotation, we can rotate the laser rays. From linear algebra, see the rotational matrix in equation 2 where the θ parameter is the angle of counterclockwise rotation. Then, we can use $\theta = -\frac{\pi}{2}..\pi$ to represent rotations from east to south.

$$\begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix} \tag{2}$$

Application of the robot's map position is then straightforward: we add the coordinates of the ray with the position of the robot. This operation displaces the ray from $(0, 0)$ to the current position of the robot.

And finally, we need to find the first collision alongside the ray we received from the above-mentioned mathematical operations. The solution is to probe different cells along the straight line representing the laser ray. And, at the moment we find the first cell which is occupied, we report that distance as the nearest obstacle.

Summarizing the algorithm to produce rays, the Robot Hardware reactive class generates a sequence of distances to the nearest obstacles in the direction of different angles relative to the robot by following the steps illustrated in Figure 7.

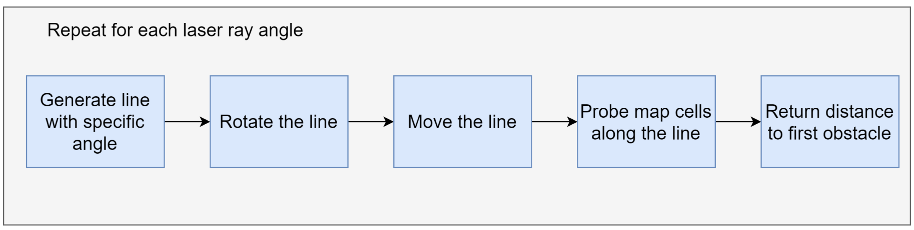


Figure 7: Algorithm to represent laser ray sensor data in the model.

Positional information. The last piece of data the mobile robot requires is the positional information. The odometry information is the position and the rotation of the robot. Additionally to the laser information, the Robot Hardware sends the odometry information as simple pieces of data.

9.4. Mobile robot behavior

By building the foundation of the model, we continue with implementing the actual logic of the robot, the section which the software engineer is responsible for. Recalling our high level objectives, the robot is required to move from point A to point B without colliding with anything.

The target position of the movement can be modelled by constructor arguments, because in the model we do not need to change the target. For flexibility purposes, we can also add a message server that changes the target position and resets the state of the robot

We believe that these are unnecessary details which add little value. That is because the change of the target can be easily performed in the *main* Rebeca section. Changing the target in the runtime is equivalent to changing the initial and target positions during model verification. Less complexity means less state space, so we decided to omit that feature in the model.

To illustrate the ability to change the software model without the need to change the hardware model, we have implemented several solutions to the problem: a naive solution, and an A*-based solution with support for obstacle detection and avoidance.

Naive approach

At first, we ignore the data from the laser sensor and move to the target disregarding any obstacles. The algorithm in that case is simple: with each iteration, the robot gets closer to the target position. Assuming the target position, or position B, is $(targetX, targetY)$, the algorithm is illustrated in Listing 5.

```
reactiveclass Robot(10) {
    knownrebecs { RobotHardware rh; }
...
 // Straightforward algorithm - just follow target, no collision avoidance
 msgsrv onLaserScan(int posX, int posY, int rot, boolean moving, double[10]
     laserDistances) {
  if(posX == targetX && posY == targetY) {
   if(!finish) {
    rh.stop();
    finish = true;
   }
  } else {
   boolean stop = false;
   if(targetY > posY) {
    if(rot != NORTH) {
     rh.turnRight();
     stop = true;
    }
   }
   else if(targetY < posY) {
    if(rot != SOUTH) {
     rh.turnRight();
     stop = true;
    }
   }
   else if(targetX > posX) {
    if(rot != EAST) {
     rh.turnRight();
     stop = true;
    }
   }
   else if(targetX < posX) {
    if(rot != WEST) {
     rh.turnRight();
     stop = true;
    }
   }

   if(stop) {
```

```
      rh.stop();
     }
     else if(!moving) {
      rh.move();
     }
    }
  }
}
```

Listing 5: Naive movement algorithm

Summarizing the algorithm, it detects the direction of the target point and rotates toward there. If the rotation is not required, it moves in that direction.

A*-based approach

Next, we are going to introduce some complex logic into the model. The algorithm can be described in two steps. Save all obstacles detected by lasers into the internal map of the robot and then build a path using the A* path-finding algorithm [42].

Following the principle of segregating components by design decisions [43], we introduce a $Pathfinder$ reactive class that encapsulates our path-finding implementation.

To find obstacles, we need to perform the opposite operation described in Figure 7 regarding modelling laser rays. In the Pathfinder's case, the information about the distance to the closest obstacle and the direction is known, but the reactive class needs to infer the exact coordinates of that obstacle.

Let's consider the triangle in Figure 8. Here, the ray of the laser is illustrated as the line AC. We know the exact length of that line from the laser sensor data. Additionally, we know the exact coordinate of dot C from odometry information as it is the position of the robot. To calculate the position of dot A, we need to know its offset of X and offset of Y from robot's position C.

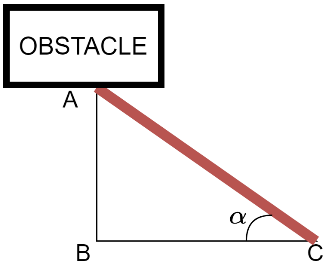


Figure 8: Laser ray with an obstacle.

Laser ray is indicated with a red line. Robot's position is C.

By constructing a right triangle with hypotenuse on AC, we can define the X coordinate offset as length of the side BC and the Y coordinate offset as length of the side AB. Also, we can get the angle of α by combining our knowledge of ray's angle and the rotation of the robot.

The next step is a matter of simple trigonometry (see equation 3). Now, as we know the offset from the dot C to dot A, the last step is to add the the position of dot C to the offset and we have the position of the sensed obstacle (see equation 4).

$$\begin{aligned} xOffset &= |AC| * cos(\alpha) \\ yOffset &= |AC| * sin(\alpha) \end{aligned} \tag{3}$$

$$(A_x, A_y) = (C_x, C_y) + (xOffset, yOffset) \tag{4}$$

The Rebeca implementation of the above algorithm is presented in Listing 6.

```
void addObstacle(double distance, double dYaw, int posX, int posY, int rot)
    {
    // calculate yaw from rotation rot and the laser ray angle dYaw
```

```
  ...
    double obsTargetX = posX + distance * cos(yaw);
    double obsTargetY = posY + distance * sin(yaw);

    int targetXIndex = round(obsTargetX);
    int targetYIndex = round(obsTargetY);

    if(targetXIndex < 0 || targetXIndex >= MAPSIZE ||
        targetYIndex < 0 || targetYIndex >= MAPSIZE)
    {
       // out of bounds
    } else {
     if(!map[targetXIndex][targetYIndex]) {
      self.changed = true;
      map[targetXIndex][targetYIndex] = true;
     }
    }
}
```

Listing 6: Obstacle detection in Rebeca

Now, with the exact coordinates of the obstacle, we can put it into the robot's internal map and indicate that the $Pathfinder$ needs to generate a new path based on the latest information.

There are three inputs to the stage of path generation: the map with detected obstacles, the robot position and the target position.

A* (or AStar) is a path-finding algorithm based on Dijkstra's algorithm, but improved for real-time use-cases. Broadly speaking, it changes the Dijkstra's algorithm from finding the 'best' path, to finding a 'good enough' path by prioritising path nodes closer to the target. By utilising this strategy, it greatly improves the runtime performance.

The implementation of A* heavily relies on tree structures and lists to represent path nodes. Unfortunately, Rebeca does not support neither of them, although, one can model them using arrays.

For lists, the following operations were implemented in order to realise A* in Rebeca:

- clear
- add
- remove
- find
- size

In the modelling scenario, there is no need for lists with ability to grow dynamically, so arrays with fixed sizes were used. The sizes of such arrays were chosen empirically in such a way, that they can hold data for most of the scenarios. The unused space in such arrays is marked with a special value of -1.

Additional issue was the absence of trigonometric functions in Rebeca. To overcome that, we introduced Taylor series approximations [44] of such values. Regarding the tree structures, they are modelled as a 2D array, where $v = arr[i][j]$ value corresponds to child-parent relationship of element at position (i, j) and element at position $(v\%MAP\_SIZE, v/MAP\_SIZE)$. The workaround was to represent a hierarchical structure.

Then, with all required tools at the disposal, the A* path-finding algorithm was implemented. After successful generation of the path, the $Pathfinder$ reactive class sends a message with the new path to the Robot Software reactive class.

Modelling in Rebeca enhances the debugging experience. Rebeca assertion mechanism allowed us to make sure that the model behaves appropriately. What the modelling engineer can do, is to insert $assert(false, “Amessagetodescribethedebugpoint”)$ to trigger a synthetic verification error in order to produce a counterexample. This counterexample shows all state transformations that led to a certain state. In case of the mobile robot scenario, one can insert such debugging assertion to fire when moving to a specific position. Then, the developer may inspect the path that the robot took to get there. If some state transitions look erroneous, the developer will be able to adjust the model.

Similarly to the debugging experience, during verification, the developer will see all the state transitions that led to an error. However, in this case, the developer does not choose the possible wrong state and/or transition, but such state is automatically detected by the RMC. Then, one can deduce the incorrect transition and fix the bug in the model.

Verification of safety properties and utilisation of debugging assertions allowed us to find and fix the following bugs and issues in our Rebeca model in different iterations:

- saving redundant data to state variables (running time of model verification grew)
- incorrect comparison signs in loop statements
- the path was followed in reverse (the robot tries to get to the endpoint first, instead of moving around obstacles)

This concludes the modelling phase for Robot Software, as all required functionality (moving, detecting obstacles, reaching target position) was modelled. The sensors and actuators were modelled with great precision in order to capture the behavior of these components. This allowed our Robot Software part to be modelled in a way that is very close to the target ROS 2 implementation, but with the advantages of model checking.

9.5. Multi-robot model

The structure of the model allows increasing the number of robots by instantiating more Robot Software and Robot Hardware reactive classes in the main method of Rebeca model. To model a multi-robot movement scenario, we instantiated the second robot in the model.

However, our attempt to do so resulted in failure. After investigation, state space explosion problem was identified as the reason. This means that the interaction of two rebecs with complex behaviors creates a huge state space which can not be verified in a reasonable period of time. To avoid that issue, instead of duplicating the same complex robot, we simplified the second robot's behaviour. Instead of finding path in real time, it moves along a predefined path, ignoring obstacle information (see Listing 7). We call the reactive class with that behavior $SimpleFollower$.

The introduction of $SimpleFollower$ solved the problem of state space explosion by abstracting away some unnecessary properties of the path-finding robot. However, this introduced an issue to the validity of the model. The idea behind multi-robot verification is to see whether the A* algorithm responds well to moving obstacles. Since the first robot is implementing A*, we argue that the main idea behind multi-robot model is preserved.

```
msgsrv move() {
 if(pathPosition < maxPathPosition) {
  if(!stop) {
   int nextPos = path[pathPosition];
   int nextX = nextPos % MAPSIZE;
   int nextY = nextPos / MAPSIZE;

   map.move(posX, posY, nextX, nextY);
   pathPosition++;
  }
  ...
 }
}
```

Listing 7: SimpleFollower - solution to space explosion problem of Multi-robot model

One of the reasons for the state space explosion is that we extract the pathfinding functionality into a separate reactive class (See Pathfinder reactive class in Figure 6). This introduces many additional states because the Robot reactiveclass needs to communicate with the Pathfinder reactive class by the means of passing messages.

There are several possible solutions which were not implemented in the scope of this work. The issue could be fixed from the side of Rebeca modelling language by introducing functionality to extract behavior into classes which communicate by calling methods of each other, without making them fully-fledged Actors. In this case, Robot reactiveclass will access Pathfinder state and methods without the requirement of passing messages, which will reduce the state connected to the message queue. Another solution is to utilise the Rebeca **@priority** construction. In general, when RMC encounters multiple messages arriving simultaneously, it explores all the possible message order permutations in order to correctly verify the model. The **@priority** construction states the exact order of received messages by sorting the message servers and the rebecs by priority. This will prune some of the state-space and may allow running the multi-robot model without other model changes.

10. Model Checking the Models or Verifying Properties

In order to verify the modelled robotic software, we extract the safety properties for the developed models.

We added the assertions for the safety properties to the *Map* reactive class. The Listing 8 shows that each movement performed on the map asserts that the target position does not have an obstacle, therefore, preventing collision.

In addition to safety, we are interested to address the starvation problem. In order to do that, we want to make sure that at each point, except the initialization phase of all of the rebecs (which is represented by *isFirstState* variable definition), the robot is not stopped at one place. We introduce a *isRotatingOrMovingWhenNotFinished* assertion to make sure that the progress is being made (see Listing 9).

Finally, to verify that the high level objective is achieved, we check that the latest robot position is the target position specified in robot's objectives. For that, *isOnTargetPositionWhenFinished* assertion is introduced (see Listing 9).

```
msgsrv move(int xFrom, int yFrom, int xTo, int yTo) {
 int prev = map[xFrom][yFrom];

 if(prev != OBSTACLE) {
  assertion(false, "Expected the FROM coordinate to be an obstacle");
 }

 int to = map[xTo][yTo];

 if(to == OBSTACLE) {
  assertion(false, "COLLISION: Expected the TO coordinate not to be an obstacle");
 }

 map[xFrom][yFrom] = 0;
 map[xTo][yTo] = OBSTACLE;
}
```

Listing 8: Safety assertion in Map model

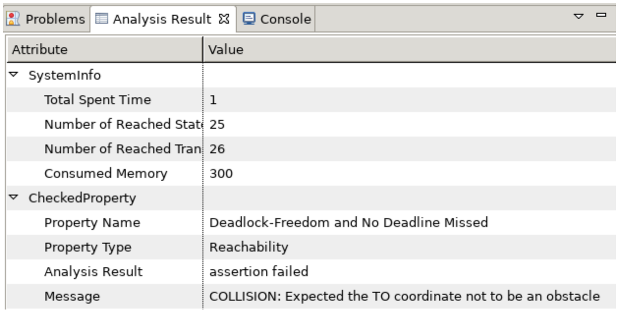
```
property{
 define {
  isOnTargetPosition = rh.posX == r.targetX && rh.posY == r.targetY;
  finish = r.finish;
  hasMoved = rh.rot != rh.prevRot || rh.posX != rh.prevPosX || rh.posY != rh.prevPosY;
  isFirstState = rh.isFirstState;
 }

 Assertion {
  isRotatingOrMovingWhenNotFinished: finish || isFirstState || (!finish && hasMoved);
  isOnTargetPositionWhenFinished: !finish || (finish && isOnTargetPosition);
 }
}
```

Listing 9: Properties verified in Rebeca model

Now, we can run the RMC with the assertions in place. There are two target models: the naive implementation and the A* pathfinding implementation.

Starting with the naive model, the algorithm works well while there are no obstacles. However, if we put an obstacle on the path of the robot, Rebeca Model Checker will show us that the safety assertion fails (see Figure 9 for the output of RMC and the counterexample).

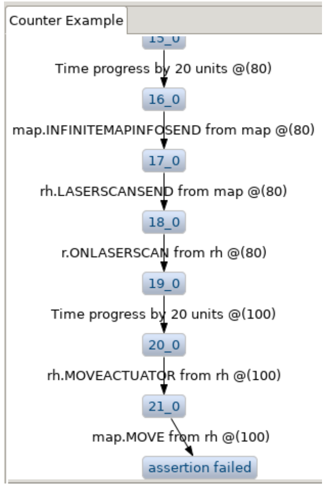
Continuing with the A* model, the RMC reports no failures, as indicated in Figure 10. The amount of states examined is considerably higher due to increased complexity of the model.



Problems | Analysis Result | Console

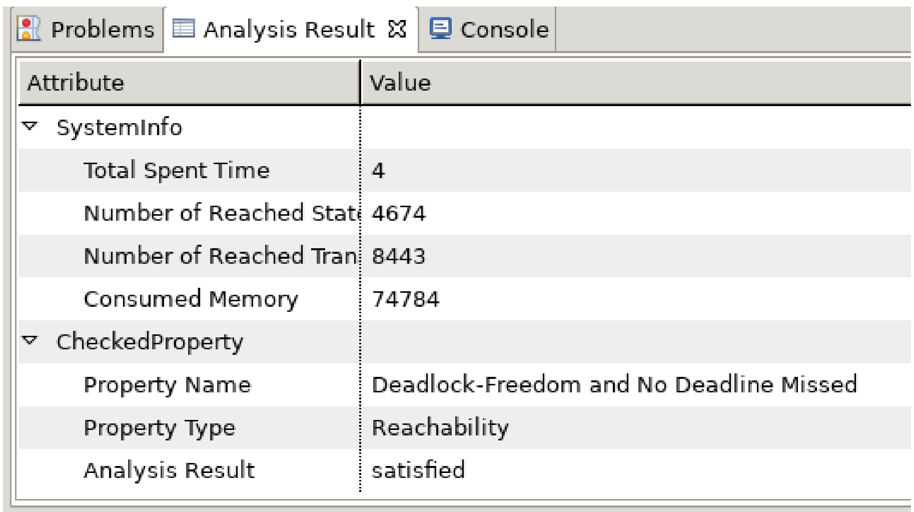
Attribute	Value
SystemInfo	
Total Spent Time	1
Number of Reached Stat	25
Number of Reached Tran	26
Consumed Memory	300
CheckedProperty	
Property Name	Deadlock-Freedom and No Deadline Missed
Property Type	Reachability
Analysis Result	assertion failed
Message	COLLISION: Expected the TO coordinate not to be an obstacle

(a) RMC error



(b) Safety counterexample

Figure 9: Failed safety verification of naive model



Problems | Analysis Result | Console

Attribute	Value
SystemInfo	
Total Spent Time	4
Number of Reached Stat	4674
Number of Reached Tran	8443
Consumed Memory	74784
CheckedProperty	
Property Name	Deadlock-Freedom and No Deadline Missed
Property Type	Reachability
Analysis Result	satisfied

Figure 10: Successful verification of A* model

11. Mapping from Rebeca to ROS 2 and the Deployment

11.1. Mapping from Rebeca model to ROS 2

Overall, the mapping algorithm consists of the following steps:

1. For every reactiveclass in the Robot Software part of Rebeca model create a ROS 2 node
   (a) For each statevar create a private field
   (b) For every method create a corresponding private method in the class
   (c) Transform Rebeca constructor to class constructor
   (d) For each message server create a topic subscription to the topic with the name of the message server
   (e) For message server parameters create a ROS 2 message type
   (f) For each send message operation to the specific message server publish a message to the corresponding topic
2. Ignore Rebeca code in Robot Hardware and Environment parts of the model
3. Create a class with all Rebeca environment variables as static data

11.2. Differences with ROS 1 mapping

Next, we compare generic Rebeca to ROS 1 mappings defined in RoboRebeca [27] and Rebeca to ROS 2 mappings within the context of mobile robots.

Robot reactive classes. Robot reactive classes (in this work we call them Robot Software) can be mapped in the same way as in ROS 1, disregarding minor syntax differences. These reactive classes will become ROS 2 nodes which have the same semantic value, as in ROS 1.

Environment variables. ROS 2 does not provide a "parameter server", therefore, one needs to create such an entity manually if needed. Since the scenarios defined in 7 did not require dynamic reconfiguration, all variables can be translated to compiler $define$ directives. This will allow different ROS 2 nodes to access the same variables, however, as in Rebeca, no global modification will be possible.

State variables. The statevars mapping rule did not require changes. ROS 2 supports node parameters which can be used to initialize the values representing the state. That will allow to support different configuration of the node mapped from a reactive class. Then, these values will be saved to the state of the node - private variables of the corresponding C++ class.

Controller. ROS 2 has no reason to contain the central piece called controller. ROS 2 nodes are decentralized and require no central control entity, so this concept becomes redundant.

Port. The port entities were used to communicate with the central server. With its deprecation, the same could be said about port message servers.

Unannotated reactive classes, methods and message servers. These reactive classes can be mapped to ROS 2 nodes in the same was as annotated ones. Message servers will transform into one subscriber and a number of publishers. Each publisher will be mapped from each individual reactive class that sends a message to the particular message server. This mapping did not change from the previous research.

Rebeca deadlines. Although, $after()$ and $delay()$ statements still have no direct transformations to ROS, the introduction of QoS (Quality of Service) in ROS 2 allowed to map the $deadline()$ statement. QoS configuration of a topic includes a message lifetime property, which purges the message from the queue if the message is older than specified. Therefore, the statement will map to a subscription with an appropriate QoS message lifespan configured.

@Class reactive classes. For performance reasons, some reactive classes conversion to ROS 2 nodes may hinder real-time performance. For this reason, this additional mapping rule was introduced. These reactive classes should be mapped not to ROS 2 nodes, but to C++ classes with a synchronous interface. For this class, all state variables are mapped to private fields and all message servers are mapped to methods.

However, since we are considering a specific movement scenario with specific required ROS 2 capabilities, we can define more concrete mapping for those capabilities.

@LaserSensor message server and odom data. We carefully designed the interface of this method to represent the actual data structures used in ROS 2 components. This message server is mapped to a subscriber to a hardware ROS 2 topic providing laser sensor data (see Section 8.3). The method body in this case will be run for each reading of sensor data. Additionally, this message server will create a subscription to odometry data, described in Section 8.2.

Robot Hardware move and rotate commands. These commands can be mapped to a publisher of an hardware ROS 2 topic */cmd_vel* responsible for movement data, introduced in Section 8.1. The model did not take into account the complexities of the linear and angular velocities, so the software developer will need to add that manually to the generated message body.

11.3. Deployment of the mapped mobile robot

The target simulator platform for deployment is the Gazebo simulator. However, after the mapping process, we had to perform multiple manual adjustments to the code. The issues that we had to fix became evident during deployment process. The Gazebo simulation with the robot, laser sensor and an obstacle illustrated is depicted in Figure 11.

Real coordinates.

Our model dealt with integer x and y coordinates which allowed us to map them to internal map coordinates straight away. However, the simulator and real world often use real coordinates. For that, we implemented a conversion between a map coordinate and real coordinate. In our case, we decided that each 1 x 1 meter square of real world will contain exactly 10 x 10 map cells. Effectively, we divided each meter into 10 map cells.

Robot dimensions.

In the modelling phase, we assumed that the robot is an entity with an infinitely small length and width. However, in real world, the robot has a width and length which should also be calculated in the algorithm responsible for collision avoidance. This issue was easily determined using the output of the visualization described in the Section 11.4.

To fix this issue, we decided to assume that an area equal to the dimensions of the robot around each obstacle is non-reachable for the robot. For each obstacle determined by the laser sensor, we additionally mark the area of that size around that point as an area with obstacles.

Discovered safety issue: rolling over

An additional safety issue was discovered after fixing previous problems. A sharp rotation angle, given the appropriate composition of the robot, may result in unstable footing. This will introduce shakiness that can result in robot rolling over itself. In order to fix that, we added a monitoring variable which makes the robot stabilize when it starts shaking.

This issue is hard to catch in modelling environment, as it is introduced by physical composition of the robot and properties may vary.

11.4. Additional software

In pursuit of better observability of the robot in the simulation environment, we produced some additional pieces of software.

Robot command. Additionally to the software of the robot, we had to build a simple graphical user interface (GUI) in order to give the robot an objective: the target coordinates (see Figure 12). This GUI provides the ability to send (x, y) coordinates to the Move topic of the robot. The robot interprets a message on this topic as the target position of its movement. The program can work with multiple robots deployed in the simulation environment by providing the user with an ability to choose any robot currently deployed.

Visualization. For quicker feedback loop, we introduced visualization of ROS 2 Robot behaviour by utilising RVIZ 2 - a visualization tool in ROS 2 environment. Visualization is often provided with components, so it became a de-facto standard in ROS 2 environment. We graphically put all perceived obstacles and the path on a map (see Figure 13 which corresponds to the Gazebo configuration showed in Figure 11 with input command specified in Figure 12). This was implemented by continuously sending obstacle and path information from the robot ROS 2 software to the RVIZ 2 tool. The green dots on Figure 13 represent the path that the robot is currently taking. The black dots represent the cells which the robot marked as obstacles. Additionally, robots odometry information is presented as a red arrow.

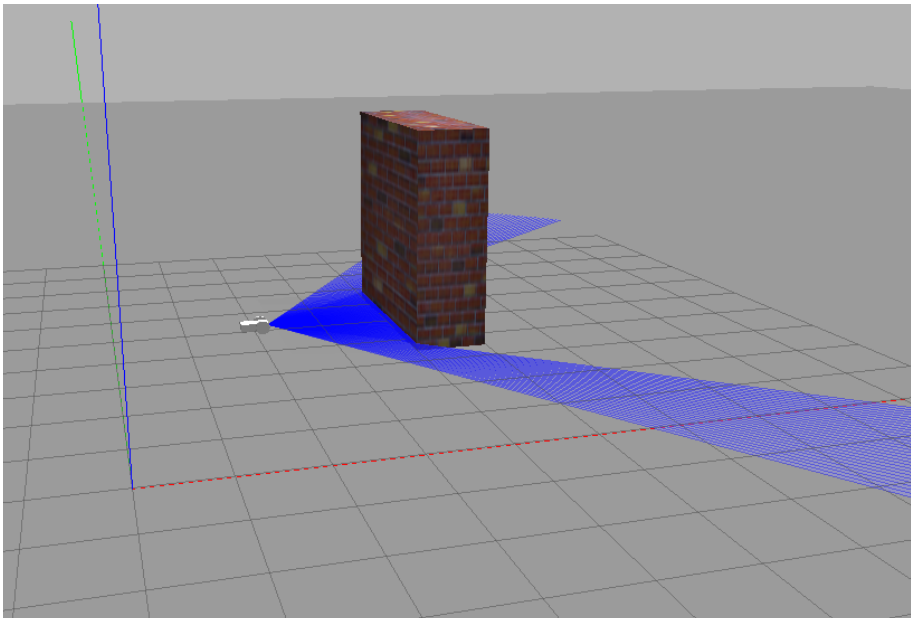


Figure 11: Gazebo simulation configuration

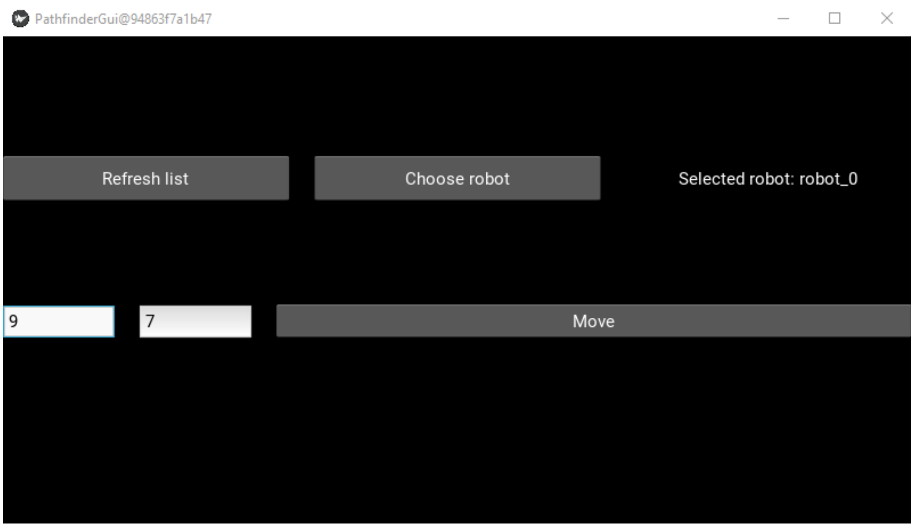


Figure 12: Robot command user interface.

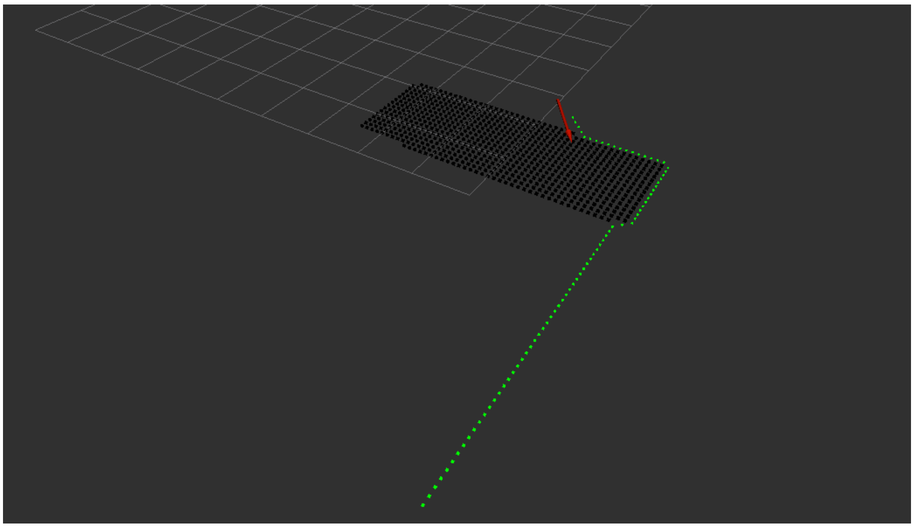


Figure 13: RVIZ2 obstacle (black dots) and path (green dots) visualization.

12. Results and Discussion

12.1. Result w.r.t. research question

We have assessed our results by enumerating the research questions and mapping them to contributions in the work.

RQ 1: What ROS 2 elements are involved in robot movement scenarios and how can we model them in Rebeca?

We have identified the following ROS 2 elements: movement command, odometry, laser sensor, a command receiver and the software which uses the above to realise the high level goals - robot software.

In order to model these, we divided the components of the model into Environment, Robot Hardware and Robot Software parts and modelled them in Rebeca (see the components view in Figure 6 in Section 9). This components division can be generalized to any robotic software and used as a basis of modelling.

Then, we presented how to model the ROS 2 components in Rebeca in Sections 9 and 9.4. The movement command, odometry and the laser sensor are not specific to the kinds of robots we are looking at in this work Therefore, the models can be reused for other robots in order to verify properties specific to their domain. Additionally, we proved that by separating hardware and software components, we made our model flexible by showing how to model different algorithms based on the same sensors and actuators.

There are two downsides of modelling with such an amount of detail: the state-space explosion problem and the absence of a standard library of functions in modelling languages. The model should be monitored carefully and checked often in order to keep the number of states and transitions to a practical amount. For example, if the RMC runs for an unreasonable amount of time, this is a symptom of the state space being too big. Either the Rebeca model contains some bugs, or the model is too detailed. The standard library problem can be worked around by introducing approximate and/or limited implementations of such functions. In our model, we implemented limited operations of a list, implemented some approximations of trigonometric functions and modelled tree structures with the array data structure. Although this issue makes the model less similar to the final ROS 2 code, the mapping from such structures to the actual constructions in the target programming language proved to be simple and straightforward.

RQ 2:How do changes in ROS 2 affect the existing mapping rules from Rebeca to ROS?

The changes in the mapping were described in Section 11.2. Overall, we found changes to be specific to the redundancy of the central server. Additionally, we introduced an additional mapping rule to accommodate to real-time systems performance requirements.

The mapping changes are relevant to all types of robotic software built on top of ROS 2. But, the limitation of this work is that it presents no automatic solution for the mapping process.

RQ 3: What problems can arise when deploying the ROS 2 code mapped from the Rebeca model?

We discovered the following issues in Section 11.3: real and integer coordinates mismatch, lack of robot size representation in the model and the effect of unstableness while attempting to perform some movements.

Mostly, these issues are related to the level of abstraction we chose to develop our model which resulted in omission of certain physical properties. The lack of robot size representation may be added to the future models, but other issues are too complex to handle in modelling environments. Formal verification of the model works best when the model assumptions are tested and verified on the real world environment. These assumptions are limitations of our developed model.

12.2. Future work

Based on this contribution, future work may include overcoming the limitations. For example, the model of the robot may be enhanced with the concept of robot size or dimensions. A practical continuation would include implementing the automatic mapping based on the mapping changes proposed in this work. Also, future work can explore modelling non-deterministic interactions.

Moreover, the contribution can be put into another context. For example, instead of focusing on mobile robots, another interesting problem is to verify manipulator robots, which are used in manufacturing settings. Those kinds of robots also have safety issues which may be useful to formally verify.

13. Conclusions

In this work, we propose a method to develop safe software for mobile robots. This method includes a modelling approach for ROS 2 components and mobile robots. Based on the developed model, safety properties of a mobile robot are extracted and verified. After verifying the model, an appropriate deployment artifact of the robot is prepared and put into a simulation environment. Discrepancies between the model and the real world which may harm robot's performance are reported.

Based on that method, we perform a case study on a specific movement scenario. In the case study, by following the proposed method, we are able to mitigate some safety issues by addressing them early: in the verification phase of the method. By applying formal verification techniques early in the development process, we remove faults from the software and make the robot safer.

References

[1] F. Ciccozzi, D. Di Ruscio, I. Malavolta, P. Pelliccione, and J. Tumova, "Engineering the software of robotic systems", in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 507–508.

[2] L. Capucha and B. Rohal-Ilkiv, "Software architecture of mobile robotic systems: A case study", in *2016 Cybernetics Informatics (K I)*, 2016, pp. 1–5.

[3] H. Zhang and L. Zhang, "Cloud robotics architecture: Trends and challenges", in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 362–3625.

[4] S. García, "Effective engineering of multi-robot software applications", in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 454–455.

[5] S. Sakakibara, "The latest robot systems which reinforce manufacturing sector", in *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, vol. 2, 2003, 2878–2883 vol.2.

[6] M. A. Diftler, C. J. Culbert, R. O. Ambrose, R. Platt, and W. J. Bluethmann, "Evolution of the NASA/DARPA Robonaut control system", in *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, vol. 2, 2003, 2543–2548 vol.2.

[7] Yu-Cheol Lee, Christiand, Seung-Hwan Park, Wonpil Yu, and Sung-Hoon Kim, "Topological map building for mobile robots based on GIS in urban environments", in *2011 8th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, November 2011, pp. 790–791. DOI: 10.1109/URAI.2011.6146018.

[8] R. Barber, J. Crespo, C. Gómez, A. C. Hernámdez, and M. Galli, "Mobile Robot Navigation in Indoor Environments: Geometric, Topological, and Semantic Navigation", in *Applications of Mobile Robots*, E. G. Hurtado, Ed., Rijeka: IntechOpen, 2019, ch. 5. DOI: 10.5772/intechopen.79842. [Online]. Available: https://doi.org/10.5772/intechopen.79842.

[9] J. Kim, Y. Choi, J. Lee, and S. Hong, "Floor-to-floor navigation for a mobile robot", in *2013 10th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, October 2013, pp. 362–363. DOI: 10.1109/URAI.2013.6677387.

[10] O. Holland, "The first biologically inspired robots", *Robotica*, vol. 21, no. 4, pp. 351–363, August 2003, ISSN: 0263-5747. DOI: 10.1017/S0263574703004971. [Online]. Available: https://doi.org/10.1017/S0263574703004971.

[11] ISO Central Secretary, "Road vehicles — Functional safety — Part 6: Product development at the software level", en, International Organization for Standardization, Geneva, CH, Standard ISO 26262-6:2018, 2018. [Online]. Available: https://www.iso.org/standard/68388.html.

[12] ——, "Robots and robotic devices — Safety requirements for industrial robots — Part 1: Robots", en, International Organization for Standardization, Geneva, CH, Standard ISO 10218-1:2011, 2011. [Online]. Available: https://www.iso.org/standard/51330.html.

[13] M. Y. Jung and P. Kazanzides, "An architectural approach to safety of component-based robotic systems", in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 3360–3366.

[14] Masayuki Murakami, "A safety failover subsystem for intelligent mobile robots", in *2007 International Conference on Control, Automation and Systems*, 2007, pp. 2493–2498.

[15] G. Beltrame, E. Merlo, J. Panerati, and C. Pinciroli, "Engineering safety in swarm robotics", in *2018 IEEE/ACM 1st International Workshop on Robotics Software Engineering (RoSE)*, 2018, pp. 36–39.

[16] C. Cheng, M. Rickert, C. Buckl, E. A. Lee, and A. Knoll, "Toward the design of robotic software with verifiable safety", in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, vol. 1, 2009, pp. 622–623.

[17] P. Trojanek and K. Eder, "Verification and testing of mobile robot navigation algorithms: A case study in spark", in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2014, pp. 1489–1494. DOI: 10.1109/IROS.2014.6942753.

[18] E. Gjondrekaj, M. Loreti, R. Pugliese, F. Tiezzi, C. Pinciroli, M. Brambilla, M. Birattari, and M. Dorigo, "Towards a formal verification methodology for collective robotic systems", in *In Proceedings of the 14th International Conference on Formal Engineering Methods*, November 2012, pp. 54–70. DOI: 10.1007/978-3-642-34281-3_7.

[19] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock, "Robochart: Modelling and verification of the functional behaviour of robotic applications", *Software and Systems Modeling*, January 2019.

[20] W. Visser, K. Havelund, G. Brat, and Seungjoon Park, "Model checking programs", in *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, Sep. 2000, pp. 3–11. DOI: 10.1109/ASE.2000.873645.

[21] J. M. Atlee and J. Gannon, "State-based model checking of event-driven system requirements", *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 24–40, January 1993, ISSN: 2326-3881. DOI: 10.1109/32.210305.

[22] (2020). Rebeca modeling language, [Online]. Available: https://rebeca-lang.org (Accessed on January 10, 2020).

[23] (2020). Afra, [Online]. Available: https://rebeca-lang.org/alltools/Afra (Accessed on January 11, 2020).

[24] (2020). RMC: Rebeca model checking engine, [Online]. Available: https://rebeca-lang.org/alltools/RMC (Accessed on January 21, 2020).

[25] M. M. Jaghoori, A. Movaghar, and M. Sirjani, "Modere: The model-checking engine of rebeca", in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC '06, Dijon, France: Association for Computing Machinery, 2006, pp. 1810–1815, ISBN: 1595931082. DOI: 10.1145/1141277.1141704.

[26] A. Jafari, J. J. S. Nair, S. Baumgart, and M. Sirjani, "Safe and efficient fleet operation for autonomous machines: An actor-based approach", in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18, Pau, France: Association for Computing Machinery, 2018, pp. 423–426, ISBN: 9781450351911. DOI: 10.1145/3167132.3167382.

[27] S. Dehnavi, *RoboRebeca: a New Framework to Design Verified ROS-based Robotic Programs*, Thesis, DiVA, 2019. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-44702.

[28] W. Kowalczyk, M. Przybyła, and K. Kozłowski, "Control of a mobile robot and collision avoidance using navigation function - experimental verification", in *2015 10th International Workshop on Robot Motion and Control (RoMoCo)*, 2015, pp. 148–152.

[29] (2020). ROS 2 Overview, [Online]. Available: https://index.ros.org/doc/ros2/ (Accessed on January 10, 2020).

[30] Open Source Robotics Foundation. (). ROS on DDS, [Online]. Available: https://design.ros2.org/articles/ros_on_dds.html (Accessed on May 10, 2020).

[31] O. S. R. Foundation. (2020). About Quality of Service settings, [Online]. Available: https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/ (Accessed on May 10, 2020).

[32] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator", in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan, Sep. 2004, pp. 2149–2154.

[33] (2019). Gazebo/TIAGo/SmartSoft Scenario [RobMoSys Wiki], [Online]. Available: https://robmosys.eu/wiki/baseline:scenarios:tiago_smartsoft (Accessed on May 10, 2020).

[34] G. B. Banusić, R. Majumdar, M. Pirron, A.-K. Schmuck, and D. Zufferey, "Pgcd: Robot programming and verification with geometry, concurrency, and dynamics", in *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS '19, Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 57–66, ISBN: 9781450362856. DOI: 10.1145/3302509.3311052. [Online]. Available: https://doi-org.ep.bib.mdh.se/10.1145/3302509.3311052.

[35] V. Germanos and E. L. Secco, "Formal verification of robotics navigation algorithms", in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, 2016, pp. 177–180.

[36] A. Tahir, K. Saghar, H. B. Khalid, U. Shadab Butt, U. S. Khan, and U. Asad, "Formal verification and development of an autonomous firefighting robotic model", in *2019 International Conference on Robotics and Automation in Industry (ICRAI)*, 2019, pp. 1–6.

[37] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects", in *Proceedings of the International Conference on Frontiers in Education : Computer Science and Computer Engineering FECS'13*, QC 20131210, CSREA Press U.S.A, 2013, pp. 67–73, ISBN: 1-60132-243-7. [Online]. Available: http://www.world-academy-of-science.org/worldcomp13/ws.

[38] P. Runeson, Ed., *Case study research in software engineering: guidelines and examples*. Hoboken, N.J: Wiley, 2012, ISBN: 9781118104354.

[39] D. T. @. OpenRobotics. (2018). Planning future ROS 1 distribution(s) - General - ROS Discourse, [Online]. Available: https://discourse.ros.org/t/planning-future-ros-1-distribution-s/6538 (Accessed on May 10, 2020).

[40] (2020). ros2/common_interfaces: A set of packages which contain common interface files (.msg and .srv)., [Online]. Available: https://github.com/ros2/common_interfaces (Accessed on April 15, 2020).

[41] M. Ben-Ari and F. Mondada, "Robotic motion and odometry", in *Elements of Robotics*. Cham: Springer International Publishing, 2018, pp. 63–93, ISBN: 978-3-319-62533-1. DOI: 10.1007/978-3-319-62533-1_5. [Online]. Available: https://doi.org/10.1007/978-3-319-62533-1_5.

[42] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths", *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[43] D. L. Parnas, "On the criteria to be used in decomposing systems into modules", *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, December 1972, ISSN: 0001-0782. DOI: 10.1145/361598.361623. [Online]. Available: https://doi-org.ep.bib.mdh.se/10.1145/361598.361623.

[44] M. Abramowitz and I. A. Stegun, *Handbook of mathematical functions with formulas, graphs, and mathematical tables*. US Government printing office, 1948, vol. 55.

Appendices

A. Code of the final Rebeca model

```
// Environment variables
...

reactiveclass Map(15) {
 statevars {
  int[10][10] map;
 }

 Map() {
  setObstacles();
 }

 msgsrv move(int xFrom, int yFrom, int xTo, int yTo) {
      ... // Moves the robot from one position to the other on the map
 }

 msgsrv subscribeToMapInfo() {
  infiniteMapInfoSend((RobotHardware)sender) after(msgDelay);
 }

 msgsrv infiniteMapInfoSend(RobotHardware h) {
  h.laserScanSend(map);

  infiniteMapInfoSend(h) after(msgDelay);
 }

 msgsrv subscribeToMapInfoFromFollower() {
  infiniteMapInfoSendToFollower((SimpleFollower)sender) after(moveDelay);
 }

 msgsrv infiniteMapInfoSendToFollower(SimpleFollower f) {
  f.onMap(map);

  infiniteMapInfoSendToFollower(f) after(moveDelay);
 }

 msgsrv setInitialPosition(int x, int y) {
     ... // Sets initial position of the robot
 }

 void setObstacles() {
    ... // Sets obstacles on the map
 }
}

reactiveclass RobotHardware(10) {
 knownrebecs {
  Robot robot; Map mapRebec;
 }

 statevars {
  int posX; int posY; int rot; boolean moving; int nextRot;

  // variables for assertion
  int prevPosX; int prevPosY; int prevRot; boolean isFirstState;
```

```
 }

 RobotHardware(int x, int y, int r)
 {
  posX = x; posY = y; rot = r;
  initMap();

  moving = true; isFirstState = true; nextRot = rot;

  moveActuator() after(moveDelay);
 }

 void initMap() {
  mapRebec.setInitialPosition(posX, posY);
  mapRebec.subscribeToMapInfo();
 }

 // Send laser data to robot software
 msgsrv laserScanSend(int[10][10] map) {
  double[10] dists;
  ... // Calculates the distances of rays 0..9 to the nearest obstacles

  robot.onLaserScan(posX, posY, rot, moving, dists);
 }

 // Move the actual robot (change pos)
 msgsrv moveActuator() {
  if(nextRot != rot) {
   ... // Handles rotation changes
  }
  else if(moving)
  {
   ... // calculates nextPos
   mapRebec.move(prevPosX, prevPosY, posX, posY);
  }

  self.isFirstState = false;
  moveActuator() after(moveDelay);
 }

 msgsrv move() {
  moving = true;
 }

 msgsrv stop() {
  moving = false;
 }

 msgsrv turnLeft() {
  ... // Handles rotating to the left
 }

 msgsrv turnRight() {
  ... // Handles rotating to the right
 }
}

reactiveclass Pathfinder(50)
{
 statevars {
     int targetX; int targetY;
```

```

      boolean[10][10] map; boolean changed;
  }

  msgsrv setTargetPosition(int x, int y) {
   ... // Changes the target position
  }

  msgsrv calcPath(int xPos, int yPos, int rot) {
   if(changed) {
    ... // Calculates the new path via A*
    ((Robot)sender).onNewPath(path); // Sends the path to the robot
   }
  }

  msgsrv addObstacles(double[10] obstacles, double[10] angles, int obstacleCount, int posX, int posY, int rot) {
   ... // Adds obstacles to the internal map
  }
 }

reactiveclass Robot(10) {
  knownrebecs {
      RobotHardware rh; Pathfinder pf;
  }

  statevars {
   int curX; int curY; int curRot;
   int targetX; int targetY;
   boolean finish;
   int[30] path; int curPathIdx;
  }

  Robot(int x, int y) {
   ... // Set target position
   self.setTargetPos(); // Set the target position of the Pathfinder
   self.rebuildPath() after(pathfinderDelay); // Rebuild path periodically
  }

  msgsrv setTargetPos() {
   pf.setTargetPosition(self.targetX, self.targetY);
  }

  msgsrv rebuildPath() {
   pf.calcPath(curX, curY, curRot);
   self.rebuildPath() after(pathfinderDelay);
  }

  msgsrv onLaserScan(int posX, int posY, int rot, boolean moving, double[10] laserDistances) {
   if(!finish) {
    ... // Updates the positional information
    ... // Extracts the obstacle information from laserDistances


    // Sends the obstacle information to the Pathfinder
    if(obstacleCount > 0) {
     pf.addObstacles(obstacles, angles, obstacleCount, posX, posY, rot);
    }

    ... // Move according to the path generated by Pathfinder
```

```
  }
 }

 msgsrv onNewPath(int[30] newPath) {
  // Save the new path from the Pathfinder
 }
}

reactiveclass SimpleFollower(20) {
 knownrebecs {
  Map map;
 }

 statevars {
  int[4] path; int pathPosition; int maxPathPosition;
  int posX; int posY;
  boolean stop;
 }

 SimpleFollower() {
  ... // Define path and position for the SimpleFollower
  startMoving();
 }

 void startMoving() {
  map.setInitialPosition(posX, posY);
  map.subscribeToMapInfoFromFollower();
  move() after(2 * moveDelay);
 }

 msgsrv onMap(int[10][10] mapInfo) {
  ... // Check if next position will not contain an obstacle, stop otherwise
 }

 msgsrv move() {
  if(pathPosition < maxPathPosition) {
   if(!stop) {
    ... // Update position and move to the next predefined path checkpoint
   }
   self.move() after(2 * moveDelay);
  }
 }
}

main {
 Map map():();

 Pathfinder pf():();
 Robot r(rh, pf):(5, 0); // Set target path to (5, 0)
 // Initialize robot at position (0, 0) facing EAST
 RobotHardware rh(r, map):(0, 0, EAST);

 SimpleFollower sf(map):();
}
```