Mälardalen University
School of Innovation, Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Software Engineering
(DVA501, 30.0 credits)

Model-based development & verification of ROS2 robotic applications using Timed Rebeca

Hong Hiep Trinh
hth21001@student.mdu.se

Examiner: Antonio Cicchetti
Mälardalen University, Västerås, Sweden

Supervisor: Marjan Sirjani & Fereidoun Moradi
Mälardalen University, Västerås, Sweden

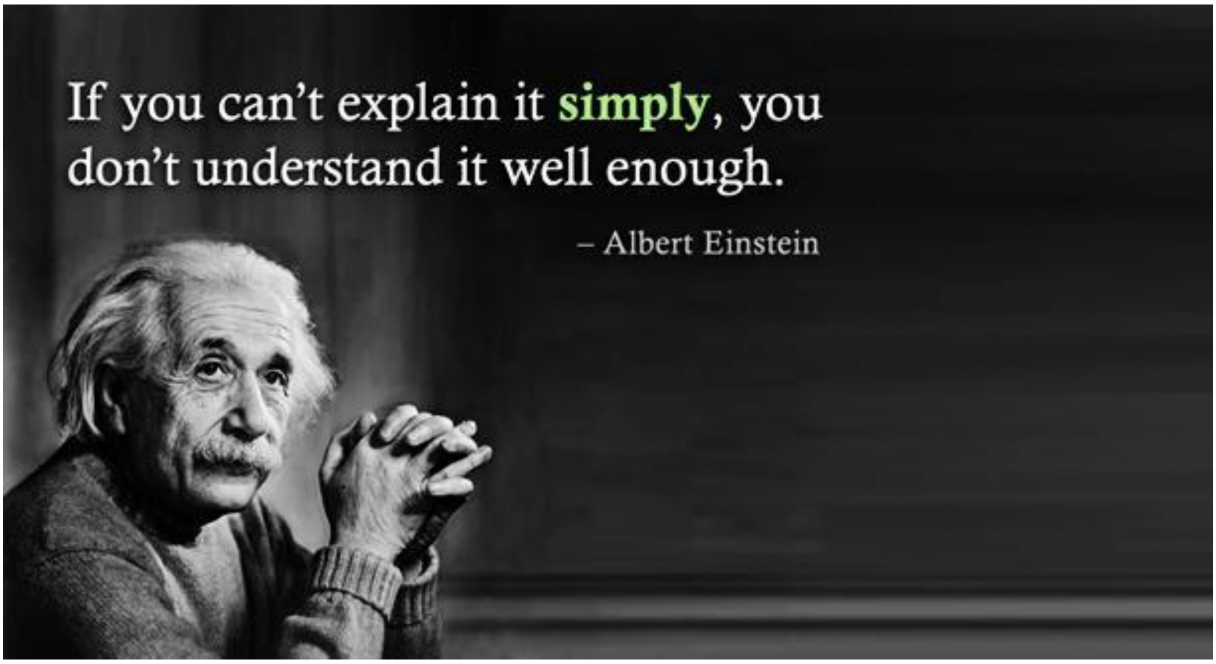
Company Supervisor: N/A

June, 2023

Abstract

ROS2 is an increasingly popular middleware framework for developing robotic applications. A ROS2 application basically is composed of nodes that run concurrently and can be deployed distributedly. ROS2 nodes communicate with each other through asynchronous interfaces; they reside in memory and wait to respond events that circulate around the system during the interactions between the robot(s) and the environment. Rebeca is an actor-based language for modelling asynchronous, concurrent applications. Timed Rebeca added timing features to Rebeca to deal with timing requirements of real-time systems. The similarities in the concurrency and message-based asynchronous interactions of reactive nodes justify the relevance of using Timed Rebeca to assist the development and verification of ROS2 applications. Model-based development and model checking allow quicker prototyping and earlier detection of system errors without the requirement of developing the entire real system. However, there are challenges in bridging the gaps between continuous behaviours in a real robotic system and discrete behaviours in a model, between complex computations in a robotic system and the inequivalent programming facilities in a modelling language. There have been previous attempts in mapping Rebeca to ROS, however they could not be put into practice due to over-simplifications or improper modelling approaches. This thesis addresses the problem from a more systematic perspective and has been successful in modelling a realistic multiple autonomous mobile robots system, creating corresponding ROS2 demonstration code, showing the synchronization between the model and the program to prove the values of the model in driving development and automatic verification of correctness properties (freedom of deadlocks, collisions, and congestions). Stability of model checking results confirms design problems that are not always detected by simulation. The modelling principles, modelling and implementing techniques that are invented and summarized in this work can be reused for many other cases.

Keywords: *model-based development, model-based prototyping, automatic verification, formal verification, model checking, ROS2, robotics, Timed Rebeca, autonomous mobile robots*


If you can't explain it simply, you
don't understand it well enough.
– Albert Einstein

Acknowledgement

After nearly five months of continuous hard work, almost without any weekend, I have finally overcome all the difficulties of this really complex thesis project and delivered break-through results. Modelling requires thinking abstractly while robotic development requires deep technical know-hows. The road to investigate a new research area, to find the right direction to solve unsolved problems is steep and bumpy, coming with not just one but multiple dark tunnels to cross, multiple narrow bottlenecks to get through. At times, even I thought I could not make it, but finally I did it.

First of all, I thank the Almighty God for blessing me with all the physical and mental strengths to stay always strong and resilient to get through these two years of many hardships and turbulences: the deadly covid19 pandemic, a breath-taking schedule & many administrative gates for a non-EU student in Sweden, an unexpected war and high inflation rate.

I want to thank my wife – Thi Thu Ha Nguyen - and the kids - Kien & Khiem - for having always been standing by me in all the ups-and-downs in life. Without them, I did not have such strong motivation to make this long and far-away journey.

I also want to thank professor Jan Carlson, the programme coordinator at MDU, for giving me the right push at the right time to catch up with the thesis selection process, and for his help with other administrative tasks.

I would like to take this chance to thank my old and new good friends for sharing interesting discussions and helping me out with various matters in daily life and academic work: Lan Kieu, anh Phuong Le, Hung Tran.

Finally, I would like to thank my supervisors: professor Marjan Sirjani and doctoral student Fereidoun Moradi, for their encouragement and valuable comments throughout the project.

Done in Västerås, Sweden from January till June 2023

Hong Hiep Trinh

In memory of my beloved grandmother, Nguyễn Thị Mông (1924-2004)

Acronyms & Terminology

Term	Meaning
AMR	Autonomous Mobile Robot
IDE	Integrated Development Environment
LIDAR	Light Detection and Ranging
OOP	Object Oriented Programming
RMC	Rebeca Model Checker
SLAM	Simultaneous Localization and Mapping
URDF	Unified Robot Description Format
XACRO	An XML macro language
w.r.t	With regards to
Q&A	Questions and Answers

Table of Contents

1. Introduction

Verification is defined by IEEE as "the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements." [IEEE 1012-2016]

Formal verification is "the use of mathematical techniques to ensure that a design conforms to some precisely expressed notion of functional correctness" [23, Per B. 2005]. Formal verification involves developing a mathematical model of a system and proving that certain properties (like safety, reachability, liveness) are not broken under all possible operational scenarios.

Model checking is a formal verification method in which the system state is expressed by a combination of state variables and each correctness property is expressed as a logical expression of the state variables. The state space is finite yet normally huge. The model is compiled and activated with different sets of state variable values by a model checker software until a property fails or all combinations have been tried, proving that the expected properties hold under all circumstances. Model checking allows automatic verification of correctness properties of safety-critical reactive systems [25, Edmund, chapter 21].

Rebeca is an actor-based language for modelling asynchronous, concurrent applications. Timed Rebeca added timing features to Rebeca to deal with timing requirements of real-time systems [4, Marjan S. 2014]. Rebeca has an OOP-like syntax and comes with a model checker software (RMC) and an IDE (Afra) to help build and check models conveniently, without requiring explicit knowledge of formal verification.

The Robot Operating System (ROS) is a set of software libraries and tools to ease the development of robotic applications by abstracting lower-level interactions with the operating system and hardwares, providing a framework for constructing a robotic software system. ROS was started in 2007 and has evolved to current version of ROS2 [22]. The software controlling a robotic system, which may comprise multiple robots working together, has to be guaranteed in terms of safety and security. Deploying & testing a robotic system on real devices and environments are costly while exhaustive testing to cover all situations is impossible. Modelling and model checking help to early & theoretically verify the correctness of system design and implementation code, henceforward there is a need to apply model checking to robotic applications. The asynchronous, concurrent nature of Rebeca's reactive objects and ROS2's nodes are alike, promising a possibility of fitting Rebeca models to a ROS2 application's context.

There have been works on mapping Rebeca to ROS2 to create formally verified robotic programs:

- (Saeid, 2019) [2]: the approach is to model a simplistic robotic system using Rebeca modelling language, then continuously verifying and revising the Rebeca model until all verifications are passed, then programmatically converting the verified model to ROS2 code so that the generated ROS2 code is guaranteed to create a formally verified robotic system. In short, it is Rebeca-driven development of ROS2.
- (Kostiantyn S., 2020) [3]: the approach is basically similar to [2], with some improvements. Typical components of a robotic system comprising multiple mobile robots are modelled in Rebeca; the created Rebeca model is then verified and manually converted to ROS2 code following similar rules; the ROS2 code is then deployed to a simulation platform to test. Gaps were identified, showing that it was not straightforward to model many parts of a robotic system in Rebeca (like robot dimensions, smooth speed changes, path-finding algorithm) and manual workarounds were needed to bridge the gaps.

However, limitations in either robotics knowledge or modelling techniques have prevented these studies from being applicable to more general robotic applications. There are still unanswered questions of in which ways Rebeca can bring benefits to robotic development - ROS2 in particular - and how to adopt it with regards to the development and verification of ROS2 robotic applications. This thesis is an attempt to readdress the problem from a new perspective, to break the shadow of many unknowns, to make it right.

The structure of a robotic software is very complex, involving more than purely logical processing inside the application code (like robot physical structure & kinematic features, environment map & obstacles, navigational algorithms, control algorithms). On one hand, overly-simplified modelling can lead to incorrectness of the model, skipping important real-world interactions & sophisticated algorithms (information loss); on the other hand, overly-detailed modelling causes the model to lose its abstraction and be no longer usable for model checking purposes. This is a new area, there are many unknown things that need to be discovered and understood gradually. Therefore, we explore the problem carefully, incrementally, yet attainably.

The nature of this research is investigating, exploratory, so things will get clearer and more affirmative along the way. This report does not only document the final findings but also our journey in exploring and working the way out. The knowledge lies in both the results and the process.

2. Background

To understand the research context, it is essential to have some foundation knowledge of related subjects: model checking, Rebeca, robotics & ROS2. Readers who are already familiar with these subjects may skip this section. However, there are focus points that we have emphasized to narrow down towards our research problems.

I utilized ChatGPT service (https://chat.openai.com/chat) to get answers to my purposeful guiding questions. The approach of interviewing an AI engine to exploit online knowledge-base is *nouvelle* and very efficient for pulling out target knowledge of interest, building a broad understanding of basic concepts, which is not new. The following annotations are used:

- [ChatGPT]: for answers that can be taken straightforward from ChatGPT.
- [ChatGPT*]: for answers that need to be further adjusted, corrected.

2.1. Model checking

What are basic concepts in model checking?
[ChatGPT] “Model checking involves several basic concepts, including:

- Model: A model of the system being verified, represented in a formal language that can be analyzed automatically.
- Properties: Requirements or desired behaviors specified for the system, also represented in a formal language.
- State space: A representation of all possible states the system can be in. The model checking algorithm explores this space to check if the system satisfies the specified properties.
- Transition system: A representation of the behavior of the system over time, including the transitions between states.
- Temporal logic: A formal language used to specify the properties or requirements of the system, allowing for the expression of temporal relationships between events and states.
- Verification: The process of determining whether the system model meets the specified properties, by exploring the state space and evaluating the temporal logic formulas.
- Counterexample: An execution of the system that demonstrates that the model does not satisfy a given property. The model checking algorithm returns a counterexample if the property is not satisfied.”

What is a model checker software and how does it work?
[ChatGPT] “A model checker software is a tool that automatically checks the correctness of a model of a system, by comparing it against a set of desired properties or requirements. The model checker software works by:

- Representing the system as a model: The system is modeled using a formal language, such as a state transition system, that can be analyzed automatically.
- Specifying properties: The desired properties or requirements for the system are specified using a formal language, such as temporal logic.
- Exploring the state space: The model checker software explores the state space of the system to determine if it meets the specified properties. This is done by evaluating the temporal logic formulas for each state in the state space, to determine whether they are satisfied.
- Reporting results: If the system model meets the specified properties, the model checker software returns a positive result. If the properties are not satisfied, the model checker software returns a counterexample, which is an execution of the system that demonstrates that the model does not satisfy the property.
- Debugging and fixing the model: If the model checker software returns a counterexample, the user can use this information to debug and fix the model, and re-run the model checker to see if the properties are satisfied.

Model checker software is used to verify the correctness and reliability of hardware and software systems, and can help identify bugs and problems before the systems are deployed."

What is a state variable? How do they relate to correctness properties? Why do not most model checkers work on continuous values?
[ChatGPT] "A state variable is a variable that represents the state of a system. In model checking, the state of the system is represented by a set of state variables, each of which represents some aspect of the system's behavior.

Correctness properties, such as safety and liveness properties, are specified in terms of the values of the state variables. The model checker software determines if the system satisfies the correctness properties by exploring the state space of the system and evaluating the values of the state variables in each state.

Model checkers work on discrete values because they explore the state space of a system by systematically checking all possible states. Continuous values cannot be systematically checked in this way, so model checkers typically work with discrete approximations of the continuous values. For example, a model checker may work with discrete intervals to represent continuous values, and check the correctness properties for the system in each interval. This approach can be used to achieve good approximations of the behavior of systems with continuous values, but it can also lead to over-approximation or under-approximation of the true behavior of the system."

Is there any model checker software that can work on continuous values?
[ChatGPT*] "Most traditional model checkers are designed to work with finite-state systems, and do not directly handle continuous values. However, there are several approaches that can be used to extend model checking to continuous systems, including:

1. **Discretization**: One approach is to discretize the continuous system, treating it as a finite-state system. This involves dividing the continuous space into a finite number of regions, and defining the transitions between these regions.

2. Hybrid systems: Another approach is to use a hybrid systems model checker, which can handle both discrete and continuous behavior. Hybrid systems model checkers typically use numerical methods to solve differential equations, and can also use symbolic techniques to handle discrete events.

These are some of the main approaches used to extend model checking to continuous systems, and there are ongoing research efforts to develop more advanced techniques for the verification of continuous systems."

What are the differences between simulation and modelling?
[ChatGPT] "Simulation and modeling are related concepts, but there are several key differences between them:

- Purpose: Simulation is used to study and understand the behavior of a system, while modeling is used to represent a system in a way that can be analyzed and used to make predictions or decisions.

- Representation: Simulation typically uses a detailed, numerical representation of a system, including the relationships between its components and the values of its inputs and outputs. Modeling, on the other hand, uses a simplified, abstract representation of a system, which may not include all the details of the real system.

- Approach: Simulation involves running the representation of the system through a series of time steps, in order to study its behavior over time. Modeling, in contrast, involves representing the system in a way that makes it amenable to analysis, such as through the use of mathematical equations, state transition systems, or graph theory.

- Output: The output of a simulation is typically a set of data or graphs that describe the behavior of the system over time. The output of a model is typically a set of predictions or insights about the behavior of the system, which may be used to make decisions or take actions.

- Use: Simulation is often used in engineering, science, and finance to study the behavior of complex systems and to make predictions about their future behavior. Modeling is used in a wide range of fields, including computer science, physics, biology, and economics, to represent and understand complex systems."

This question is meaningful because we need to be always aware of that what we are doing is modelling, not simulating.

2.2. Rebeca

Rebeca is a modelling language that uses actors to model the functional behaviour of distributed and asynchronous systems [4, Marjan S. 2014]. Actors, which are encapsulated in instances of reactive classes (*rebec*) in Rebeca, run concurrently and maintain a local queue of messages to respond to events asynchronously. Each rebec has its own state variables and can interact with other known rebecs by sending messages via calls to message server interfaces. Rebeca comes with a toolchain to help software engineers intuitively create, build and check the model against definable properties.

I want to hereby highlight the nature & limitations of Rebeca that need to be aware of when it comes to modelling complicated robotic systems:

- Rebeca's syntax is Java-like, OOP-like but it is not Java, not OOP, it does not support OOP features like class inheritance (*w.r.t version in use*). Rebeca is a modelling language, not a programming language for implementation. The model checker translates a model defined in Rebeca language to a target programming language (C++) so that it can be compiled into an executable format to be tested.
- Rebeca does not support normal public method calls among reactive objects, each rebec should be considered as a separate process and can only interact with other known rebecs by sending asynchronous messages via calls to the other's message servers. A rebec's state variables are also only directly accessible locally, not to other rebecs.
- Rebeca does not support dynamic sized arrays, associative arrays, strings. State variables can only be a rebec, or a single value or a fixed-sized array of primitive types (*boolean*, *int*, *short*, *byte)*. The reason is that Rebeca's model checker can only do exhaustive testing with discrete values, not with an infinite continuitum. Structured type as C++ *struct* is not supported as well.
- Calls to commonly used mathematical functions (like $sqrt, abs, pow, cos, sin, tan, atan$) are not interpreted by RMC to equivalent functions in C++. Developers therefore have to create their own implementation if they need to use these functions.

Workarounds must be devised to overcome these limitations. Such understanding will help to find the right contexts to fit Rebeca into ROS2 applications and to develop assisting tools to bridge the gap between implementation and model.

2.3. Robotics

Mapping ROS2 to Rebeca firstly means modelling relevant robotic problems for verification purposes, and ROS2 is an implementation structure that realizes a robotic system in its architectural topology and communication mechanisms. Without sufficient understanding of the building blocks of robotic systems and typical robotic problems, the research mission can easily get out of track in terms of practicality and feasibility. In such view, ROS2 comes as a secondary information layer, the primary layer is the robotic problems that are relevant to be captured by the models.

This section provides readers with a quick introduction to robotics, its common terminology and typical problems. Again, ChatGPT was used to facilitate a Q&A session under our guiding questions.

What is robotics and what are its typical problems?

[ChatGPT*] "Robotics is the branch of engineering and computer science that deals with the design, construction, operation, and use of robots. Robots are often defined as programmable, multi-functional machines capable of sensing, reasoning, and acting.

Typical problems in robotics include:

- Perception: Robots must be able to sense their environment, interpret the information gathered from sensors, and make decisions based on that information.

- Motion planning and control: Robots must be able to move around their environment and manipulate objects in a safe, efficient, and accurate manner.

- Mechanic design: Robots must be designed with the appropriate mechanical structure and actuators to perform their intended functions.

- Software systems: Robots require sophisticated software systems to coordinate their sensors, actuators, and computation.

- Human-robot interaction: Robots must be able to interact with people in a way that is intuitive and safe.

- Navigation: Robots must be able to navigate autonomously in complex and dynamic environments.

- Task planning and decision making: Robots must be able to plan their actions, make decisions, and adapt to changing situations in real-time."

What are the most common types of robots?
[ChatGPT] "The most common types of robots are:

- Industrial robots: These are robots used in manufacturing and production settings, such as assembly line robots and welding robots.

- Service robots: Service robots perform tasks for individuals, such as cleaning robots and personal assistant robots.

- Medical robots: Medical robots assist in medical procedures, such as surgical robots and rehabilitation robots.

- Mobile robots: Mobile robots are capable of movement and can be autonomous or remotely operated, such as delivery robots and planetary rovers.

- Humanoid robots: Humanoid robots are designed to resemble human beings, both in appearance and behavior.

- Drones: Drones are unmanned aerial vehicles that can be remotely operated or autonomous, and are used for a variety of applications, including delivery, surveillance, and search and rescue.

- Collaborative robots (Cobots): Cobots are robots designed to work alongside humans, performing tasks in a collaborative manner.

- Agricultural robots: Agricultural robots perform tasks in agriculture, such as planting and harvesting crops."

What are mathematical models that are often used in robotic problems?
[ChatGPT] "Mathematical models are frequently used in robotics to describe and solve problems related to perception, motion planning and control, and decision making, among others. Some of the most common models include:

- Transformations: Robotics problems often involve transforming points, vectors, and frames from one coordinate system to another. This can be done using homogeneous transformations, rotations matrices, and Euler angles.

- Kinematics: Kinematic models are used to describe the motion of robots, including their forward and inverse kinematics, and Jacobian matrices. These models are used to determine the configuration of the robot and to plan its motion.

- Dynamics: Dynamic models are used to describe the forces and torques acting on the robot, and to predict its behaviour under different conditions. They can be represented using Lagrangian and Newtonian formalisms, and involve the use of mass matrices, gravitational forces, and friction models.

- Control: Control models are used to regulate the motion of robots, including position control, velocity control, and impedance control. They often involve the use of feedback control loops and linear quadratic regulators (LQR).

- Probabilistic models: Probabilistic models are used to represent uncertainty in robotic problems, such as Monte Carlo methods, Kalman filters, and particle filters. These models are used for perception and decision making, and provide a probabilistic representation of the robot's state and environment."

It is note-worthy that: [ChatGPT] "
- Dynamic modeling in robotics is the process of mathematically representing the motion of a robot and its components, taking into account the interactions between them and the effects of forces, torques, and energy. It aims to predict the robot's behavior and performance under different conditions, and is used for control and planning purposes.
- Kinematic modeling in robotics, on the other hand, focuses solely on the description of the robot's motion without considering the forces and torques involved. It involves specifying the relationship between the robot's joints and end-effector, and is used to determine the configuration of the robot for a given task or to plan its motion.

Dynamic modeling takes into account the forces and energy involved, while kinematic modeling only focuses on the motion."

What are common navigation problems and algorithms in robotics?
[ChatGPT] "Navigation is a fundamental problem in robotics, and there are many different problems and algorithms related to it. Some of the most common include:

- Localization: Localization refers to the problem of determining the robot's position in its environment. This can be done using various methods such as simultaneous localization and mapping (SLAM), Markov localization, and particle filters.

- Mapping: Mapping involves building a representation of the robot's environment, and can be done using various methods such as occupancy grids, topological maps, and semantic maps.

- Path planning: Path planning involves finding a safe and efficient path for the robot to follow from its starting position to its goal. This can be done using various methods such as Dijkstra's algorithm, A* algorithm, and rapidly-exploring random trees (RRTs).

- Obstacle avoidance: Obstacle avoidance involves ensuring that the robot avoids obstacles while navigating its environment. This can be done using various methods such as potential fields, velocity obstacle methods, and reactive navigation.

- Motion planning: Motion planning involves finding a feasible and safe motion for the robot to execute. This can be done using various methods such as motion primitives, sampling-based motion planning, and trajectory optimization."

What is occupancy grid and how does it help to solve navigational problems and collision avoidance problems in robotics?
[ChatGPT] "An occupancy grid is a representation of a robot's environment that encodes information about the presence or absence of objects in the environment. It is a two-dimensional grid of cells, where each cell can be either occupied or free. The occupancy grid is generated by mapping the environment and updating the grid cells based on sensor readings and other information available to the robot.

Occupancy grids are commonly used to solve navigation and collision avoidance problems in robotics, as they provide a simple and efficient way to represent the robot's environment and reason about it. For example, they can be used to generate a map of the environment that the robot can use for path planning. By using an occupancy grid, the robot can determine which cells are occupied by obstacles and which are free, allowing it to plan a path that avoids collisions.

In addition, occupancy grids can also be used to solve the localization problem by providing a probabilistic representation of the robot's state in the environment. This can be done by updating the occupancy grid based on the robot's sensor readings and using Bayesian techniques to estimate its position."

How is a robot's physical structure represented and how are their shape transformations modelled in robotics?
[ChatGPT] "A robot's physical structure is typically represented using a kinematic model, which describes the geometry of the robot and its joints. The kinematic model consists of a set of interconnected bodies, referred to as links, which represent the robot's links, and joints that connect these links together. The joints can be either revolute, which allow for rotation about a single axis, or prismatic, which allow for translation along a single axis.

Shape transformations in robotics are modelled using homogeneous transformations, which are used to transform points, vectors, and frames from one coordinate system to another. Homogeneous transformations can be represented as matrices and are used to describe the position and orientation of the robot's links and joints in space.

Forward kinematics is the process of determining the end effector's position and orientation given the joint angles and the kinematic model of the robot. Inverse kinematics, on the other hand, is the process of determining the joint angles required to achieve a desired end effector position and orientation.

These models allow robots to be represented and manipulated mathematically, enabling robots to be controlled and their motion to be planned and optimized."

What are common types of sensors that a mobile robot uses and how do they work?
[ChatGPT] "Mobile robots commonly use various types of sensors to perceive and interact with their environment. Some of the most commonly used sensors include:

- Lidar: Lidar (Light Detection and Ranging) uses lasers to emit pulses of light and measure the time it takes for the reflected pulses to return. This information is used to generate a 3D point cloud representation of the environment, which can be used for obstacle detection, mapping, and localization.

- Camera: Cameras are often used for visual perception and navigation. They can be used for tasks such as object recognition, depth estimation, and tracking.

- Ultrasonic sensors: Ultrasonic sensors use high-frequency sound waves to detect obstacles and measure distances. They are commonly used for collision avoidance and wall following.

- Infrared sensors: Infrared sensors detect changes in temperature, which can be used to detect obstacles or to track the position of other robots or objects.

- IMU (Inertial Measurement Unit): IMUs are used to measure the orientation and linear acceleration of the robot, which can be used for navigation and dead reckoning.

- GPS (Global Positioning System): GPS is a satellite-based navigation system that provides the location of the robot in a global coordinate system. GPS is often used in outdoor robots for navigation and mapping."

What are common communication protocols used in robotics?
[ChatGPT*] "Communication protocols are used in robotics to allow robots to exchange information and coordinate their actions. Some of the most common communication protocols used in robotics are:

- TCP/IP: The Transmission Control Protocol/Internet Protocol (TCP/IP) is a widely used communication protocol for data exchange over networks, including the Internet. TCP/IP is often used in robotics to allow robots to communicate with other devices and systems, such as computers, sensors, and actuators.

- ROS (Robot Operating System): ROS is an open-source robotics middleware that provides a standardized communication infrastructure for robotics systems. It commonly uses a publish/subscribe messaging model to allow robots to exchange information and coordinate their actions.

- MQTT (Message Queue Telemetry Transport): MQTT is a lightweight communication protocol that is commonly used in the Internet of Things (IoT) for machine-to-machine (M2M) communication. It is often used in robotics to allow robots to communicate with other devices and systems, such as sensors, actuators, and cloud services.

- CAN (Controller Area Network): CAN is a communication protocol that is widely used in the automotive and industrial control industries. It is often used in robotics for low-level communication between the different components of a robot, such as the microcontroller, sensors, and actuators.

- RS-232/RS-485: RS-232 and RS-485 are serial communication protocols that are often used in robotics for communication between different devices, such as sensors and actuators."

What are typical architectures used by robotic systems?
[ChatGPT] "Robotic systems typically have a number of different components that work together to achieve their goals. The architecture of a robotic system refers to the overall design and organization of these components, including their relationships and communication pathways. Some of the most common architectures used by robotic systems are:

1. Distributed architecture: In a distributed architecture, the various components of the robotic system are spread out and communicate with each other over a network. This allows for increased flexibility and scalability, as well as improved fault tolerance.

2. Client-server architecture: In a client-server architecture, one component acts as a central server and coordinates the actions of the other components, which are referred to as clients. This is a common architecture for large, complex robotic systems, as it provides a centralized point of control and reduces the amount of communication required between components.

3. Peer-to-peer architecture: In a peer-to-peer architecture, the components of the robotic system communicate directly with each other, without the need for a central server. This can provide increased flexibility and scalability, but also requires more sophisticated communication and coordination algorithms.

4. Hybrid architecture: Many robotic systems use a hybrid architecture that combines elements of the above architectures to take advantage of the strengths of each.

The specific architecture of a robotic system is influenced by a number of factors, including the size and complexity of the system, the requirements for communication and coordination, and the desired level of fault tolerance. The choice of architecture can have a significant impact on the overall performance and effectiveness of the robotic system, so it is important to choose an architecture that is well-suited to the specific requirements of the application."

As robotics is a large & complicated inter-disciplinary field, getting familiarized with its terminology is essential for understanding the work of this thesis project.

2.4. ROS2

A ROS2 robotic software system can be modelled as a network of distributed nodes which run concurrently and interact with each other through different inter-process asynchronous messaging mechanisms (publisher-subscriber topics, services, actions) to achieve some robotic goals [6, ROS2 online docs]. A ROS2 node can be programmed in C++ or Python or Java, utilizing the underneath ROS2 libraries.

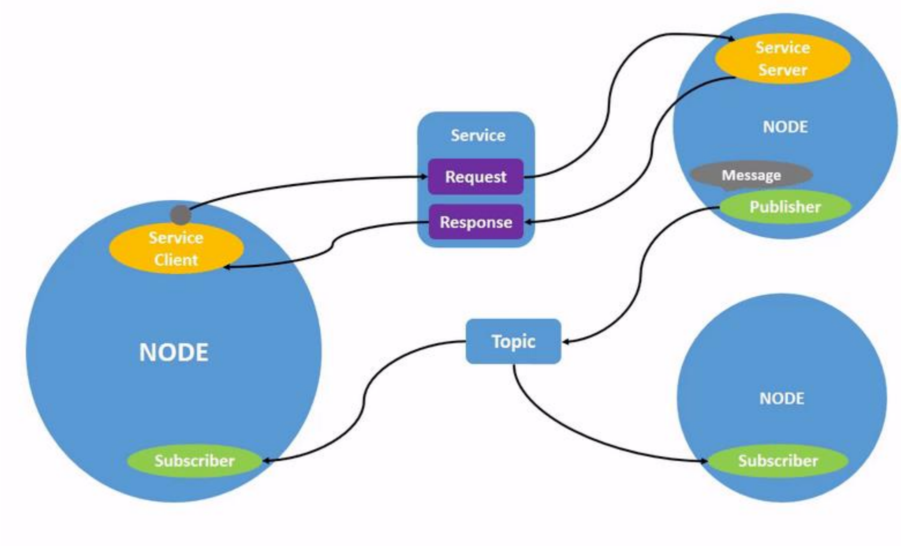


Figure 1: A ROS2 node graph (source: docs.ros.org)

The major communication mechanisms of ROS2 nodes are different in these ways:

1. **Topics**: a topic is a named bus/channel where one or many nodes broadcast data to the channel and other nodes register to the channel to receive data. The subscribers do not need to know the publishers, they just need to know the name and data structure of the topic, they receive data whenever it is available on the subscribed topic. Topics are suitable for streaming data such as sensory data or image processing output or robot states, which need to be updated in real time. Topics can facilitate *1-to-many* or *many-to-many* asynchronous communication channels.

2. **Services**: Services are used for request-response communication between nodes, where one node (the client) makes a request to another node (the server) and receives a response. There may be multiple client nodes using the same service but a service is provided by only one server node. Services are used for discrete, well-defined tasks, such as asking the robot to move to a specific location.

3. **Actions**: Actions are used for long-running goal-oriented tasks that can be monitored and potentially canceled (preempted). Actions allow a node to request a task from another node and receive periodical feedback on the progress of the task. If the task takes too long and shows little chance of success, the caller node can actively ask the doer node to cancel it.

Each node has its own settings called ***parameters*** that can be get or set by other nodes through standard services.

For each event, a **callback** function is to be supplied to specify the behavior of the node when the event happens. For example, when subscribing to a topic, a node can provide a callback which will be triggered whenever the node receives a message on that topic. Similar events are when a service is called or done, when an action is started/done/cancelled or when a feedback is received, or when a new value is set for a parameter.

The concurrent, asynchronous nature of a ROS2 node matches with the same way an actor behaves in an actor-based model like Rebeca.

ROS2 uses a simplified description language, the interface definition language (**IDL**), to describe data structures used in communication interfaces (*.msg* files for topic messages, *.srv* files for services, *.action* files for actions). This description makes it easy for ROS tools to automatically generate source code for the interface type in several target languages. A node just needs to know the name of the topic/service/action and its message structure to interact with.

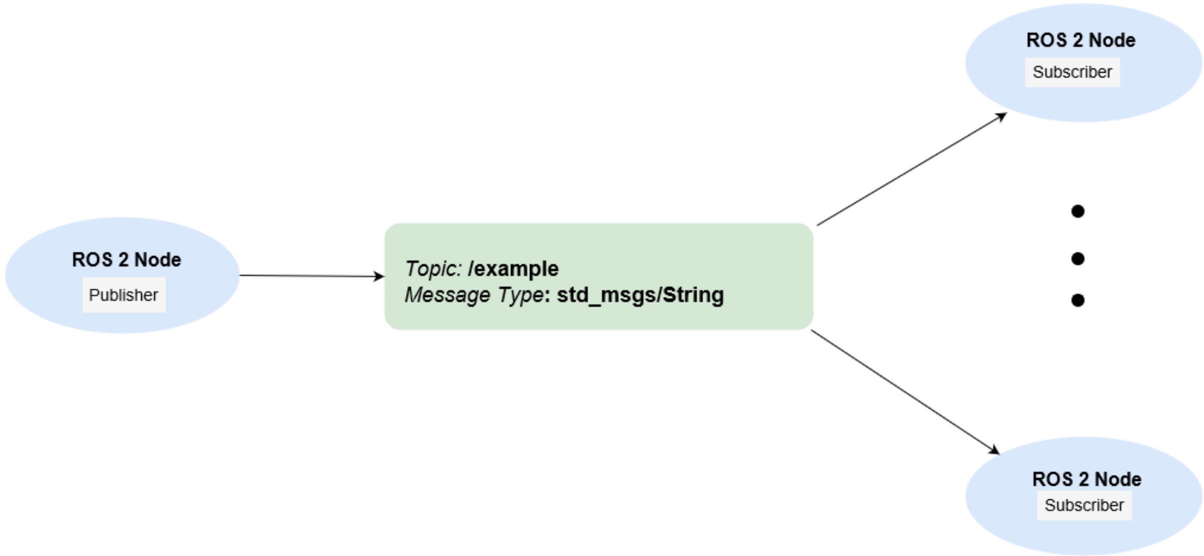


Figure 2: ROS2 nodes & topics (source: https://se.mathworks.com/help/ros/ref/ros2publisher.send.html)

Discovery is the process through which ROS2 nodes advertise about their presence and exposed interfaces so that nodes know how to talk to each other. Discovery of nodes happens automatically through the underlying middleware of ROS2. The ROS Discovery System is responsible for managing the registration and discovery of nodes, topics, services, and parameters.

When a node starts up, it will advertise the topics it publishes, the services it provides, the actions it offers, and the parameters it manages. This information is sent to the ROS Discovery System, which makes it available to other nodes in the same network. Nodes continue to periodically advertise their presence so that connections can be made with new-found entities, even after the initial discovery period. When a node goes offline, it also informs other nodes.

Other nodes can then query the ROS Discovery System to determine which interfaces are available and to learn about the topics, services, actions, and parameters offered by other nodes. For example, a node can use the `***ros2 topic list***` command to list all the topics available in the system, or use the `***ros2 service list***` command to list all the services available in the system.

In this way, the ROS Discovery System provides a centralized mechanism for nodes to discover and interact with each other, and enables nodes to dynamically adapt to changes in the system. This makes the system more flexible and scalable, and reduces the need for nodes to have prior knowledge of the interfaces exposed by other nodes.

The ROS2 CLI tool provides commands to extract information about the node topography, interfaces and message types, which are very helpful to analyze the architectural elements and communication flows of a ROS2 system.

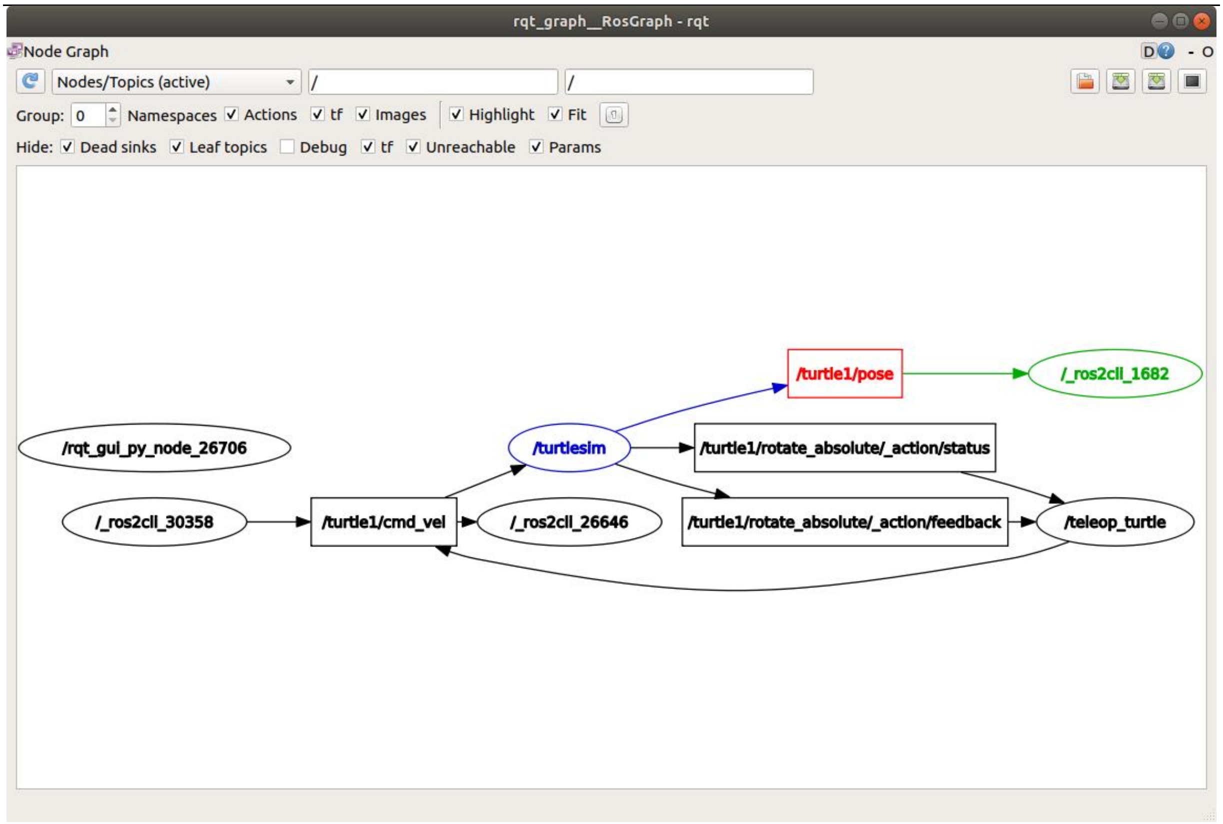


Figure 3: Node graph extracted and visualized by the "rqt_graph" tool (source: docs.ros.org)

Above is a quick look at ROS2, in the next parts we will look more closely at more specific problems like:

- How is mapping represented in ROS2?
- How does ROS2 handle obstacle detection, collision avoidance, path planning and replanning for mobile robots?

So that we can find if such robotic problems can be modelled in Rebeca. In short, the target to capture in modelling is the robotic problems that are solved in ROS2 architecture and interaction flows.

2.5. Coordinate transformations

Robotics heavily relies on mathematical models to represent the positions and movements of a mobile robot's relative location on a map or the different moving parts of a robotic arm. A location or an angle is always attached to a certain reference coordinate system, whether it can be the map frame or the robot's movement frame or others. During the process of devising algorithms for obstacle detections, collision avoidance, path planning and developing the model and the simulation code, I experienced many mistakes and bugs with coordinate transformations. Coordinate transformations are an essential part of a robotic system; as it is very easy to misunderstand and miscalculate relative coordinates in different reference frames, it is important to document it carefully. This section is mainly summarized from my own knowledge and experience in developing ROS2 code.

A navigation system in robotics contains these basic frames:

- The $world$ frame: fixed frame on earth
- The map frame: the frame attached to a floor map to pinpoint the location of a robot on the map. The map has a scale factor to scale down from real measurements.
- The robot's (movement) frame ($odometry$ frame): the dynamic location of a robot, which is denoted by a point (x, y) on the map and an orientation angle θ. The triple (x, y, θ) forms a 2D pose of the robot.
- The robot's $base\_link$ frame: the dynamic location of a robot's base, usually the same as the odometry frame.
- Other internal frames of the robot: laser base, wheel base, etc. denoting relative positions of the laser sensor or wheel from the $base\_link$ frame. For simplicity, we let the robots have only one frame: the $odometry$ frame, all other frames are made the same.

It is a repeated task to transform the coordinates of a point between frames – from the map to the robot frame, and vice versa – to know its relative positions to the map or the robot's current pose.

Given a point $P(x, y)$ in a $2D\ Cartesian$ coordinate system with origin O and axis OX and OY, a transformation T is done on the original OXY system to a new pose $O'X'Y'$, we need to know the new coordinates of P in the new system $P(x', y')$. It is noteworthy that the transformation is done on the origin and axises, not on the point in concern, the point is fixed.

There are only a few fundamental transformations: *shifting*, *rotating*, *scaling*.

Shifting (or translating)
Origin O is moved to a point $O'(a, b)$ in the old frame, the angle of the axises are unchanged. We have:

$$x' = x - a$$
$$y' = y - b$$

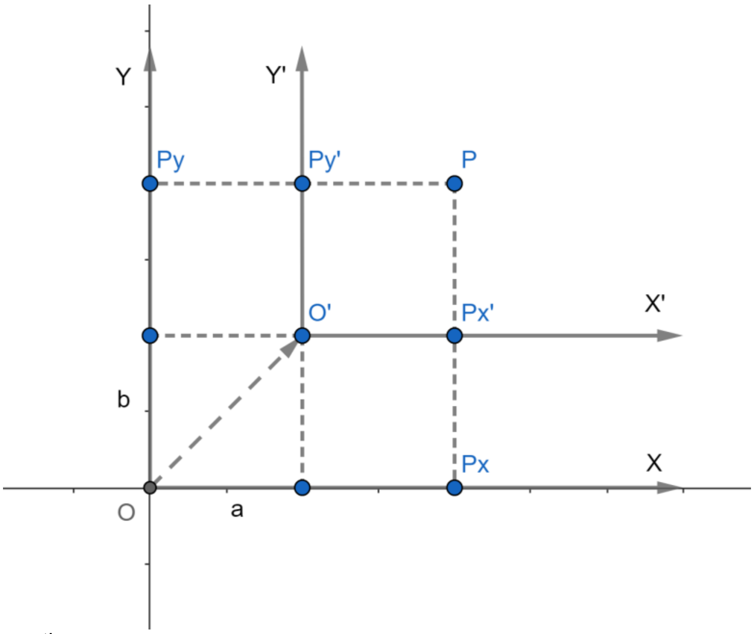


Figure 4: Shifting transformation

So the location of the old origin O in the new system is $(-a, -b)$.

In reverse direction, if we need to transform $P(x', y')$ in the new frame to $P(x, y)$ in the old frame, we need to shift $O'X'Y'$ to O which is at $(-a, -b)$ in the new system, then we apply the shifting formula:

$$x = x' - (-a) = x' + a$$
$$y = y' - (-b) = y' + b$$

Rotating
In a rotating transformation, origin O is kept fixed, the axises are rotated a $theta$ angle (by standard convention, positive angle = counter-clockwise, negative angle = clockwise).

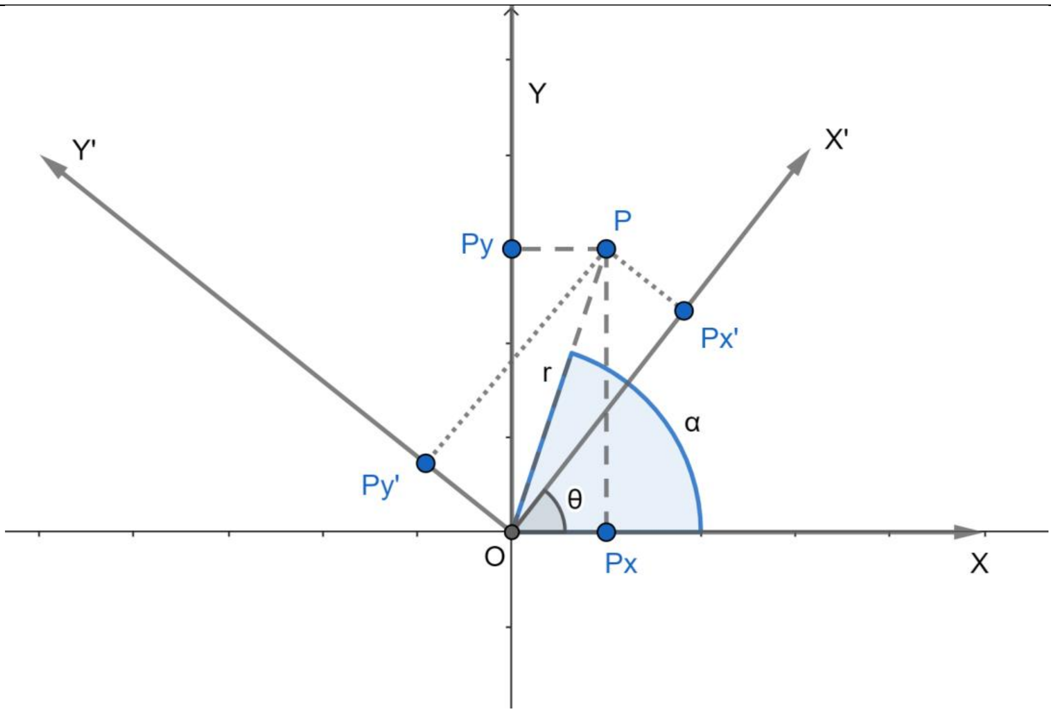


Figure 5: Rotating transformation

Let r *be the length of vector* $\overrightarrow{OP}$, α *be the angle of vector* $\overrightarrow{OP}$ *to X axis* , then $P(r, \alpha)$ is the polar coordinate of point P in OXY system, then $P(r, \alpha - \theta)$ is the polar coordinate of point P in the new system $O'X'Y'$. We have [13, Wikipedia]:

$x = r.cos\alpha$
$y = r.sin\alpha$

$x' = r.cos(\alpha - \theta) = r.cos\alpha.cos\theta + r.sin\alpha.sin\theta = x.cos\theta + y.sin\theta$
$y' = r.sin(\alpha - \theta) = r.sin\alpha.cos\theta - r.cos\alpha.sin\theta = y.cos\theta - x.sin\theta = -x.sin\theta + y.cos\theta$

Written in matrix multiplication format:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The matrix $\begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix}$ is called the rotation matrix. On the reverse way, when we need to convert coordinates from the new system to the old system, we rotate $O'X'Y'$ system a reverse angle $-\theta$, then we have:

$x = x'.\cos(-\theta) + y.\sin(-\theta) = x'.cos\theta - y.sin\theta$
$y = -x'.\sin(-\theta) + y'.\cos(-\theta) = x'.sin\theta + y'.cos\theta$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

It is a common mistake to take this reverse rotation matrix as the rotation matrix.

Some quick formulas for rotation at special angles:

- Rotating +90 degrees: $cos(+90) = 0, sin(+90) = 1$, so $x' = y;\ y' = -x$
- Rotating −90 degrees: $cos(-90) = 0, sin(-90) = -1$, so $x' = -y;\ y' = x$
- Rotating 180 degrees: $cos(180) = -1, sin(180) = 0$, so $x' = -x;\ y' = -y$
- Rotating +45 degrees: $cos(+45) = sin(+45) = 1/\sqrt{2}$, so $x' = (x + y)/\sqrt{2};\ y' = (-x + y)/\sqrt{2}$
- Rotating −45 degrees: $cos(-45) = 1/\sqrt{2}, \sin(-45) = -1/\sqrt{2}$, so:
  $x' = (x - y)/\sqrt{2};\ y' = (x + y)/\sqrt{2}$

Combined transformation
Supposed that a robot R is at pose (rx, ry, θ) relative to the map frame $O_m X_m Y_m$, we need to know how a point $P(x, y)$ on the map (like an obstacle) relatively appears to the robot's current position.

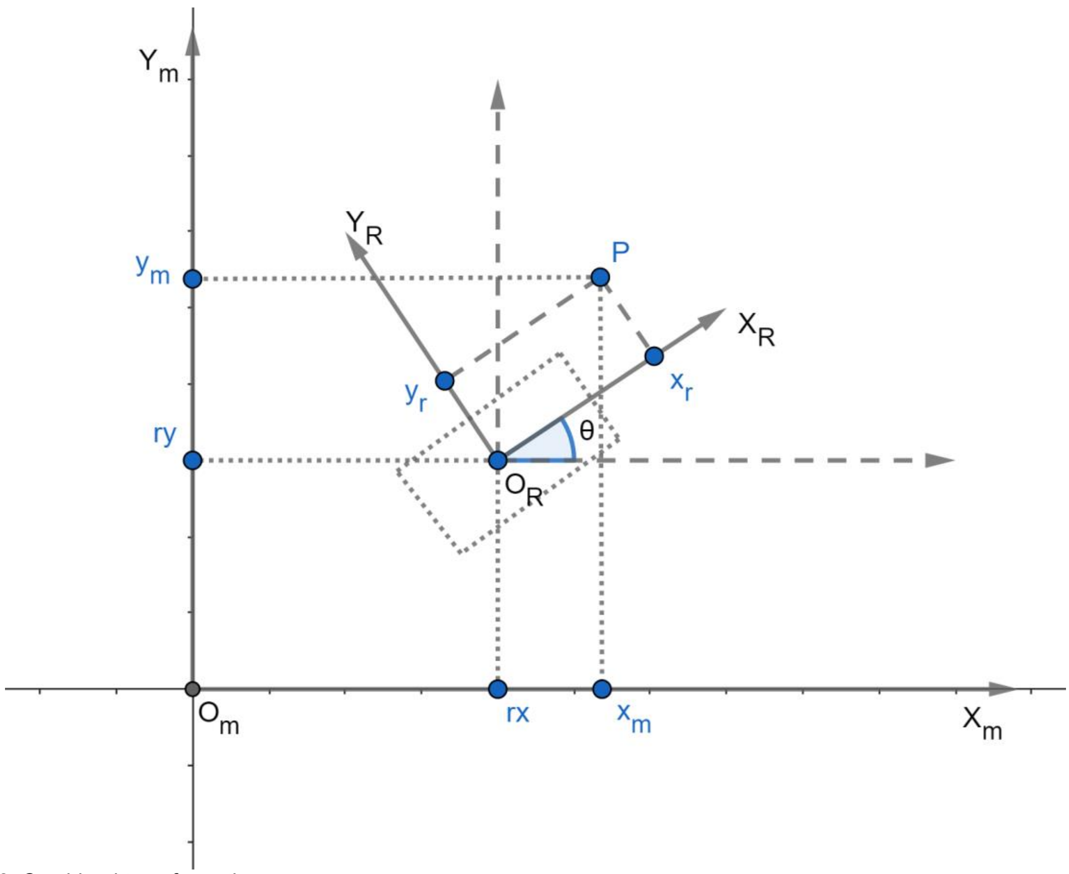


Figure 6: Combined transformation

To get that, we do two transformations:

- First, shift the origin to the robot's location (rx, ry): $x' = x - rx$; $y' = y - ry$;
- Then, rotate the axises to the robot's current angle θ:
  - $x' = (x - rx).cos(\theta) + (y - ry).sin(\theta)$
  - $y' = -(x - rx).sin(\theta) + (y - ry).cos(\theta)$

When converting a point $P(x', y')$ in the robot frame to the map frame, we do the reverse way:

- First, rotate the robot frame to straight direction (the robot frame's axis $O'X'$ is the same angle as the map's OX axis) (rotation angle is –θ):
  - $x = x'.cos(-\theta) + y'.sin(-\theta) = x'.\cos\theta - y'.sin\theta$
  - $y = -x'.sin(-\theta) + y'.cos(-\theta) = x'.\sin\theta + y'.cos\theta$
- Then, shift to the map origin O which is at $(-rx, -ry)$ in the rotated frame above:
  - $x = x - (-rx) = x + rx = x'.cos\theta - y'.sin\theta + rx$
  - $y = y - (-ry) = y + ry = x'.sin\,\theta + y'.cos\theta + ry$

The order and the direction of the transformations (positive or negative) are important to get a correct result.

Scaling
The map comes with a scale factor k, denoting a width on the map is equal to k units in reality. So, when converting from real world coordinates to map coordinates:

$$x_{map} = x_{world} / k$$
$$y_{map} = y_{world} / k$$

When converting from map to world:

$$x_{world} = x_{map} * k$$
$$y_{world} = y_{map} * k$$

The transformation operations can be programmed as class methods in an OOP code so that the basic transformations can be done more understandably, consistently, and conveniently.

3. Related Work

3.1. Rebeca to ROS mapping

Saeid, 2019 [2]: the work by Saeid can be regarded as a Rebeca-driven development of ROS (version 1), an analogy is Test Driven Development method where tests are created first and the implementation has to evolve to pass all the tests. The contribution of (Saeid, 2019) includes:

- How to model ROS constructs in Rebeca, including a central controller node, simple mobile robots and publisher-subscriber messaging mechanism.
- General Rebeca-to-ROS translation mechanisms and mark-up technique to generate ROS code from Rebeca.

However, as Rebeca is not designed for modelling robotic systems, Saeid's approach can only be demonstrated on overly-simplified scenarios of mobile robots ("toy problems" – *two robots are set up to move horizontally and vertically so that they cross, obstacle detection is simplified as returning "true" or "false" value – without tracking which path which speed each robot walks, we do not know whether they cross each other and where and when it may happen; and obstacle detection in real robotic systems is usually based on laser scanning*). Many essential robotic aspects are ignored, like:

- Mapping & path planning: a mobile robot usually uses a floor map of static obstacles, which can be prepared priorly or dynamically explored and constructed by a SLAM robot, and different sensors for detecting surrounded obstacles on its way. Navigation algorithms depends on mapping information of static and mobile obstacles.
- Physical aspects of the robots: mobile robots are not always simply depicted as non-dimensional, omni-directional dots to fit one square on the map and only move horizontally or vertically. Shapes and dimensions of the robots affect collision detection, especially for large robotic vehicles. Robotic arms have internal body transformations that affect its reach. Other physical factors of movement like velocity, acceleration & stopping distance are not considered as well.
- Complicated computations: it is not always possible to model the changes on state variables as predictable algebraic assignments ($x := f(x)$); there are many situations where the function f can be a complicated optimization model (like path planning, control algorithms) whose outputs are not predictable.

Kostiantyn, 2020 [3]: Kostiantyn went further with Saeid's approach by studying closer of typical components of a mobile robot system (laser sensor, map server and path finder) and modelling those in a more realistic way so that the model reflects more closely the actual interactions of the robot(s) and the environment in simulated deployment. Along the way, he documented the issues he encountered and how he managed to realize more complicated modelling in Rebeca. Valuable contributions of Kostiantyn are:

- Awareness of complexity of modelling geometric computations in robotic problems
- Understanding of obstacle detection and path planning algorithms, and ways to imitate in Rebeca
- Raising several programming gaps between Rebeca modelling language and ROS2
- Documenting some practices in developing and debugging Rebeca models

Kostiantyn was able to set up more complicated mobile robotic scenarios with a single robot or one robot and one simpler moving object. Nevertheless, what he did is closer to simulating, not modelling. The introduction of more computational complexity in a Rebeca model makes it more a replication of real behaviours, and less usable for model checking (e.g. *on-the-fly trigonometric calculations with real angles*, that's why the approach fails when it comes to two robots of the same complexity). Furthermore, he did not express an explicit understanding of the discrete nature in the models and the continuous nature in the real world. The model-reality gaps are coherent, and attempting to bridge such gaps raises the question of the relevance of modelling that way.

Our research is in the reverse direction – mapping ROS2 to Rebeca, which is more bottom up and more challenging as we have to be able to extract the right model(s) from real robotic applications for verification purposes. Less assumption can be made as the target is to make Rebeca model checking applicable to more generic & practical ROS2 applications.

3.2. Robotic problems & formal verification

Liu et al., 2020 [10]: they built a Simulink/Stateflow model for the RTPS (real-time publish and subscribe) protocol, which is the underlying protocol of the publish-subscribe communication mechanism in ROS2, and translate the model to a Timed Automata in Uppaal to verify the properties of the RTPS protocol. Though their work relates to robotics, it is more a protocol verification process that only needs to care about timing assumptions and state transitional conditions; all interesting robotic behaviours can be ignored. Timed Rebeca allows doing the same thing with its timing extension, and our target in this thesis is more than just timing verification, we aim to deal with more real-world robotic behaviours.

Matt Luckcuck et al., 2019 [8]: This article systematically surveys the state-of-the-art in formal specification and verification for autonomous robotics. As autonomous robotic systems depend on sophisticated software-based controlling and decision making and are often deployed to safety-critical scenarios, testing through simulated or real deployments is not enough to detect problems. Formal methods offer a stronger form of verification using mathematically-based techniques to ensure the correctness of, and provide sufficient evidence for the certification of, robotic systems. Most common approaches are:

- Model checking: besides Rebeca, there are UPPAAL, PRISM, NuSMV, SPIN, etc.
- Run-time monitoring: ROSRV is an example (https://github.com/cansuerdogan/ROSRV)
- Theorem proving
- Integrated formal methods
- Formal software frameworks & architectures

Of those, model checking is the most widely used approach to verifying robotic systems because of two main reasons:

1) model checkers are automatic
2) the concept of checking every state in a model to see if a required property holds is similar to exhaustive testing, thus relatively easy to understand and explain to stakeholders without formal methods experience

The paper also summarizes hindrances to applying formal methods to robotic systems development:

- Resistance to adoption: the perception is that applying formal methods is a complicated additional step in the engineering process, which prolongs the development process while not adding to the value of the final product.
- Lack of appropriate tools
- Lack of interoperability between formalisms: models or specifications of similar components are incompatible and locked into a particular tool. There is a need for a common framework for translating between, integrating different formalisms.
- Lack of clear guidance
- Formalizing the *last link*, the step between a formal model and program code: the program code can be automatically generated or manually written to reflect the created model; how to ensure that the model is represented by the code is a remaining question.

The summary study by Matt Luckcuck et al. offers us a broader and deeper understanding of formal methods and tools in robotics, and further reinforces the relevance of bringing Rebeca to robotic software development, ROS2 in particular. We need to be aware of making it usable, clearly guided and ensuring the "*last link*" synchronization between the model and the program code.

Filip Berlin & Sebastian Granath, 2021 [9]: a master thesis on improved obstacle detection & avoidance algorithms based on occupancy grid, A* path planning for an Automated Guided Vehicle for transporting luggage at the airport. In this context, we know about how an AGV robot behaves differently with regards to obstacle detection and avoidance (it tries to detect obstacles early and divert away to avoid a stop, because the robot is large and heavy (1.5 tons), a stop may cause its engine to stop and require a human supervisor to come to restart). This behavior is different from smaller, faster mobile robots in warehouses (warehouse robots can stop and resume movement easier). Their work helps to understand about navigational problems of real-world robots, and justifies the need of modelling such situations to realize practical use of model checking in robotics. Without sufficient knowledge in robotics, it is easy to come up to just dealing with of non-practical problems.

4. Problem Formulation

The purpose of this thesis project is to bring Timed Rebeca modelling language and model checker to ROS2 robotic applications as a usable toolchain to assist development and correctness verification. We further have an ambition of fully or partially automating the process so that ROS2 developers can use the toolchain with ease. On one hand, we avoid over-simplification of robotic scenarios, which results in unrealistic & unusable solutions; on the other hand, we avoid over-complication since robotic behaviours are very complex, the *model-reality gaps* do exist and we have to find the right contexts to fit the model in without losing modelling correctness. On the way of investigating and experimenting, following research questions are expected to be addressed:

- RQ1: What robotic problems can be modelled in Timed Rebeca and how to model them?
- RQ2: Which correctness properties of ROS2 robotic applications are suitable to verify with Rebeca model checking?
- RQ3: Can the Rebeca models be generated from ROS2 system design and implementation structures?
- RQ4: What are the achievements and unresolved challenges?

Due to the nature of a new research, the questions are treated more as initial expectations and hypotheses rather than actual targets, because we start from a lot of unknowns that need to be revealed gradually and our initial understanding of the problem need to be corrected and adjusted along the way of discovery.

5. Method

As the subject of applying Rebeca to ROS2 robotic problems is new while robotics is very complex and our initial knowledge of robotics is limited, we start from mostly unknown or little known matters and try to find the way out to break the shadow, gain more knowledge and come up to something that works. Defining what to do, what to limit is even very difficult in the beginning. The best suitable research method for our context is exploratory research.

What is exploratory research methodology? What are its approaches?
[ChatGPT] "Exploratory research methodology is a type of research that is used to explore a topic or problem in order to gain a better understanding of it. It is often conducted when little is known about a topic, and is used to generate ideas and hypotheses for further research. Exploratory research can take many forms, but generally involves gathering and analyzing data from a variety of sources in order to gain a more complete understanding of the topic.

There are several approaches to exploratory research, including:

1. **Literature Review**: This involves reviewing existing literature on the topic to gain a better understanding of what has already been studied and what gaps exist in current knowledge.

2. **Case Studies**: This involves analyzing a specific case or set of cases related to the topic of interest, in order to gain insights into potential factors and patterns that may be driving the issue.

3. **Focus Groups**: This involves bringing together a group of people with similar characteristics or experiences to discuss the topic in a structured setting, in order to gain insights into their perceptions, opinions, and attitudes.

4. **Surveys**: This involves gathering data from a large sample of people through a structured survey questionnaire in order to gain insights into their perceptions, attitudes, and behaviors related to the topic.

5. **Observation**: This involves observing individuals or groups in a natural or controlled setting to gain insights into their behaviors and interactions related to the topic.

These approaches can be used individually or in combination, depending on the nature of the research question and the available resources. The goal of exploratory research is to generate hypotheses and ideas that can be tested in subsequent research."

As pointed out by [12], "*an important aspect here is that the researcher should be willing to change his/her direction subject to the revelation of new data or insight. Such a research is usually carried out when the problem is at a preliminary stage. It is often referred to as grounded theory approach or interpretive research as it used to answer questions like what, why and how*."

Along the journey, we repeated multiple cycles of knowledge evolution which compose of following steps:
(1) forming hypotheses
(2) studying cases and literature, interviewing experts
(3) identifying the gaps between our hypotheses and observations
(4) adjust our understandings (including correction of misunderstandings, addition of unknown knowledge, reformulation of problem definition and approach)

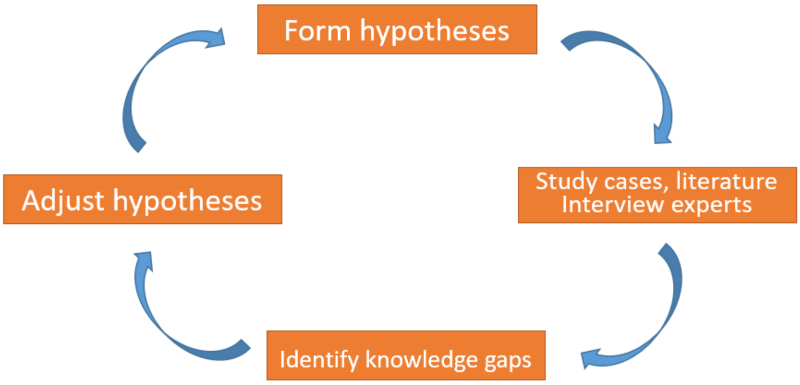


Figure 7: Knowledge evolution cycle

ChatGPT is used as a most available expert to put up with a lot of questions and get free answers from – which apparently need to be double-checked of correctness, yet are very helpful to give hints to find existing knowledge.

To achieve the research goals, the below process is followed:

1. Roll out cycles of knowledge evolution to define the problems and approaches
2. Study cases to draw out commonalities and formulate modelling patterns
3. Develop a working model to form a reusable baseline template
4. Develop ROS2 program to realize the model
5. Revise the model to synchronize with the code, test the synchronization
6. Summarize the achievements and unresolved challenges

This research is not supposed to be a complete, conclusive one as it is a ground-breaking attempt to cope with such complex matter – modelling (ROS2) robotic problems in Rebeca for model checking purposes. Its process and findings will pave the way for many future works to continue and expand.

6. Ethical and Societal Considerations

The research in this thesis aims to improve the formal verification process of ROS robotic software using Rebeca model checking. Formal verification is a proven method for early detection and prevention of faults, which reduces development cost and safety hazards by checking a simplified theoretical model of a system instead of real deployment. By sticking to empirical methodology the research results are evidence based, well documented and can be verified independently. There are no visible ethical & societal issues that are connected to the research conducting process and future adoption of its findings. Any use of proprietary information & materials was done through non-disclosure agreements signed with involved parties.

7. Exploratory directions

A possible direction is where a solution may be found, there is no guarantee that a solution can always be found there but it is where we have a chance. An impossible direction is one that there are no solutions or there exists a high barrier which is not possible to overcome, given current condition. Similar to an exploration of a new land, first we need to identify which direction to go; and if the selected direction leads to a dead end or a blocking barrier which is impossible to pass given the current constraints, then we may have to go back to switch to another direction.

7.1. Direction #1: Straightforward

This direction naturally comes as the easiest option to think of, yet the hardest or mostly impossible one – it can be compared to "*take the bull by the horns*". After looking at several ROS2 codebases, it is observed that:

- The straightforward code-to-code approach does not work when there is no ROS2 code at all – that means the robotic system is in conceptual design phase and we need to model the robotic problems in the design to early detect design faults and adjust parameters and processing logics.
- What we need to model is not the code, but the ROS2 robotic design that the ROS2 code implements. That means we always have to go to a higher abstraction level of the implementation concretes. **The code is seen as a manifestation of the system design**.
- Even when having a ROS2 code, we cannot do anything with it if we do not know what to model and what to verify. Modelling everything implemented in a code does not make sense, as it is not possible to do so and is just redoing what the code has done – which is not what modelling is supposed for. Modelling must be extracting a narrowed view from the system for a certain purpose.
- The term "code" can easily mislead attention to only code files in some programming language. This is a wrong understanding of information sources that need to be taken care of in modelling. A ROS2 codebase contains different resources in different formats that define the structures and behaviours of a robotic system. The C++ or Python code files define part of the processing logics, the behaviours. That's why we switched to use the terms "ROS2 system design and implementation structures" instead of "the code". "The code" is just a handy short term to refer to.

Having a tool to generate a Rebeca model out of a given ROS2 codebase is a most desired option, however it is just the final cover of sub-problems that have been solved underneath. It requires knowing for sure the patterns and conversion mechanisms, and a super powerful & smart parser - which is not doable in our starting position where we do not know yet the rules and given the limitations in terms of time and effort.

For those reasons I gave up the straightforward direction. Nevertheless, the investigation process helped build my understanding of the problem and gave hints to find solution in another way.

7.2. Direction #2: Use mark-up helpers

The idea is inspired by the technique used in automatic generation of API documentation. Purposeful mark-up commenting syntax is devised to facilitate the content generation, together with parsing of the code. This way the actual code is not affected because extra mark-ups are added only in comments. Below is an example used in a PHP API document generator:

```
/**
 * @api {get} ?action=:action&first=:firstname&last=:lastname&email=:useremail&phone=
 * @apiSampleRequest http://www.example.com/rest/callloop.php
 * @apiName Subscribe
 * @apiGroup Subscription
 * @apiDescription subscribes a user to the provided group
 * @apiParam {string="subscribe","unsubscribe","trigger"} action=subscribe API Action
 * @apiParam {String} [first] Optional first name
 * @apiParam {String} [last] Optional last name
 * @apiParam {String} [email] Optional user email
 * @apiParam {String} phone User's mobile phone number
 */
```

Figure 8: Markups in code for generating API document

However, on further investigation, I found that this approach, even if feasible, does not provide an overall solution to our problems because:

- It only helps to mark some entities (classes, properties, functions, function body)
- It is still a code-based approach, similar to direction #1. It does not work when there is no ROS2 code yet or when the code is untouchable.
- It is not very helpful as the node graph is made available already by ROS2 CLI tool.
- It may only help in case we need to map the processing logic in the callbacks. However, the translation will always need to go through a higher abstract level.

For those reasons, I also gave up the direction #2 but reserved it as an individual technique that may later be used.

7.3. Direction #3: Go through a model

7.3.1. Overall

Recognizing that the code is just a realization of a design idea, an architecture, a model – in other words, a higher abstraction of the system, I came up to the most feasible direction – that is going through a model.

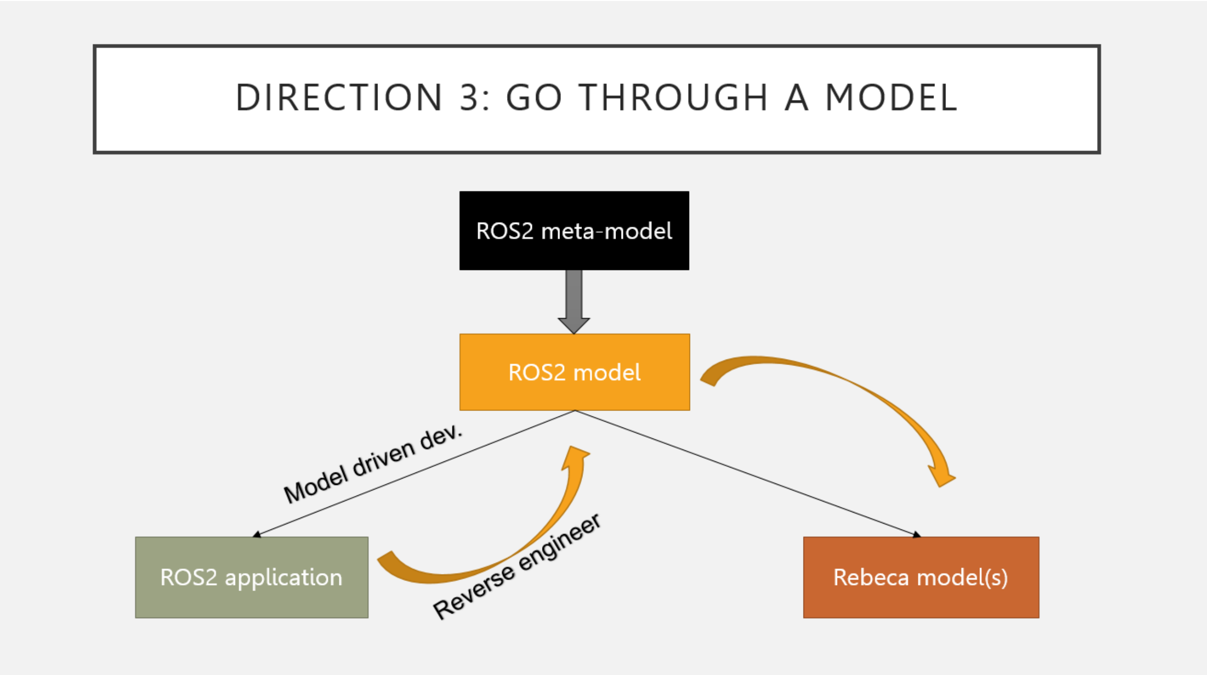


Figure 9: Direction #3

For a new code, it is model-driven development – from the model to the code. For an existing code where we do not have the model, it is a reversed engineering process in which the model is reconstructed from the code. Fortunately, the node graph of a ROS2 application is registered to ROS2 platform and can be extracted by using ROS2 CLI tool and visualized by “rqt” tool. The behavioural specifics are defined in the callbacks.

The mapping from ROS2 to Rebeca is **1-N relationship** – i.e. multiple Rebeca models can be created from a ROS2 application to extract different views for verification of different properties.

Within the scope of this thesis I do not develop the domain-specific language for modelling ROS2 applications – as it is a different big concern, I only document it here to show the full engineering flow from model-based development perspective.

What we have to model are the robotic problems in ROS2 applications, not the code. These problems can be:

- Obstacle detection and avoidance for single or multiple mobile robots, robotic arms.
- Collision-free reachability: a mobile robot can travel safely to its destination; a robotic arm can drive its end-effector to expected locations without collision with itself or other objects.
- Freedom from deadlocks: the robot(s) do not get stuck during the way to reach their destination.

- Timing, sequence requirements: an action must be achieved within some period of time, some states/actions/responses must or must not happen in some order.
- Concurrency problems: as ROS2 nodes react in an asynchronous, parallel, concurrent manner; there are possible problems of synchronization that can be efficiently detected by model checking.

When facing a robotic system, what we need to understand are the architectural design, the communication flows, the responses to events, so that we can model correctly the entities in the system and the ways they interact with each other to deliver desired system services. One separate model may be created to investigate and verify one robotic problem or some problems combined. It is not recommended to think of making a *for-all-purposes* model that can conveniently facilitate investigating all problems in one place, as if such model can be created it will become too much complicated for model checking. Anyway, the real system itself is the only for-all-purposes model. Trying to create such comprehensive model will result in a meaningless replication of the implementation code, which should be avoided.

7.3.2. Challenges & solutions

Though finding direction #3 as a most feasible way to go, I faced many challenges to start with:

- After checking several available ROS2 codebases (like the sample **turtlebot3** project or some others that can be found on github.com with “ROS2 mobile robot” keyword), I found that they are not usable for my purpose. Reasons are that they are packed with unneeded features which only overcomplicate the problem, or the parts that are of interest (behavioural algorithms) are not provided or not built in, or are not easy to modify and gain a thorough ownership of the code. Due to these reasons, I decided that I have to start from scratch – building a working model, creating a demonstration code, evolving and synchronizing between them – that also means the development work is doubled: modelling and implementing.

- Due to various limitations in programming and visualizing, it is difficult to code and troubleshoot robotic behaviours in Rebeca. To overcome these, I used a middle script to help prepare data, test similar codes & algorithms, visualize scenarios that happened in the model. The approach of using a script to bridge the gaps between real world implementation and modelling, instead of working directly and solely on the modelling language, has been proven to be an efficient approach.

- ROS2 is a middleware layer that wraps up underlying libraries to facilitate the easier creation of thinner applications atop. When modelling a ROS2 application, we have to model the whole stack, hardware data and physical interactions – for example: range detection sensory data, path planning algorithm, collision detection, movements. These things are complicated, if not handled strategically, they may result in ending up with overly-complicated, buggy, non-working and incorrect models.

- The former approaches done by Saeid and Kostiantyn should be avoided, instead of inherited, as they all led to dead ends due to over-simplification or over-complication. A new, different, more sensible approach must be invented. The complexity of the problem is high, while it must be fitted within the time frame and delivery constraints of a thesis project. On the one hand, I must avoid over simplification; on the other hand, I must put realistic simplifications and limitations to make the research progress forward and resultful within the given scope.

The first decision to make is to choose which robotic problems to model and verify. Again, the problems of collision avoidance, navigation with mobile robots are closest and interesting to study. Robotic arms are not considered in this project as their internal shape transformations in the 3D space are complicated to model given the current state of art. However, the lessons learnt in modelling mobile robots will be very useful to adapt to other robotic problems. This matter will be later discussed in the discussion sector.

8. Modelling autonomous mobile robots

8.1. Why autonomous, why multiple?

Autonomous robots are robots that can do their assigned jobs independently, without human control. The more autonomous a robot is, the more human-like thinking amount it has to do, the more complicated controlling & learning algorithms it has to depend on, thus the more leaning towards model-based development and formal methods for verifying their correctness. Autonomous mobile robots are untethered from wired power, use a set of sophisticated sensors, cameras, machine learning & artificial intelligence algorithms, to find their way through undeterministic environments with static and unexpected mobile obstacles which could be a fallen box, another similar mobile robot, a walking through human. Under such circumstance, they have to think what to do next – slow down, stop, wait, or reroute to avoid obstacles – very much similar to the way humans do when moving around. Obstacle detection, path planning & replanning, collision avoidance are common and complicated tasks that offer a large ground for applying model-based development and model checking.

I first look at modelling typical robotic problems with autonomous mobile robots. There are operation scenarios that involve one single mobile robot or multiple ones like multiple warehouse robots. In any case, a mobile robot has to deal with mobile obstacles which can be another mobile robot or walking around humans. A mobile robot can itself equally represent any mobile obstacle, that’s why modelling a system of multiple mobile robots also covers the scenarios of one single mobile robot. I therefore investigate straightly multiple mobile robots’ scenario, instead of single one.

Figure 10: Illustration of multiple warehouse robots.

(source: https://www.mhlnews.com/technology-automation/article/22054632/rise-of-the-warehouse-robots)

The modelling result and lessons learnt in this context will set a base for modelling other robotic interactions and provide reusable templates for other cases.

8.2. Modelling node architecture & interfaces

A node in ROS2 is equivalent to a *rebec* (reactive object) in Rebeca. They all have internal state variables and only interact with each other through asynchronous messaging mechanisms. Publish-subscribe topics, services, actions, callbacks in ROS2 can be mapped to message server *msgsrv* interfaces in Rebeca.

For a multiple mobile robots problem, two kinds of node need to be created:

- Robot node: which represents the "brain" of a mobile robot, taking inputs from sensors (LiDAR), processing sensory data (range detection) to navigate safe and efficient ways to reach its goal. Each robot is an instance of the robot node class with different working parameters.
- Map server node: a shared supporting and coordinating node which provides mapping feature, path planning service, simulating features (e.g. laser scan simulation), and traffic coordinating service to the robots. The map server can be thought of as the infrastructure with streets, buildings and traffic lights; while the mobile robots be self-driving cars to run around on top.

There are multiple ways to model pub-sub topics:

- The basic flow of a pub-sub topic is that when the publisher (sender) has data to send, it will broadcast the data to the channel (topic) and the registered callbacks of subscribers (receivers) are triggered when data is received. In Rebeca, we skip the "data sending" process, and let the sender call the callback of the receiver directly. In ROS2, the sender does not need to know the receiver, but in Rebeca, it has to know to trigger the callback of the receiver (through the caller object indicated by Rebeca's $sender$ keyword or passed explicitly in $msgsrv$ parameters).
- Periodic sending is modelled by a nested call inside the data sending $msgsrv$ of the sender with an "$after(sending\_rate)$" timing primitive.
- To model faithfully the *many-to-many*, *anonymous* relationship between publishers and subscribers in a topic, a separate topic manager class has to be created which maintains a list of subscribers and provides a sending $msgsrv$ to allow publishers send data to. However, this way of modelling is not used because it introduces an additional reactive class (topic manager), and there is only one publisher (map server node).

A service interface is modelled similarly:

- The service point is modelled as a $msgsrv$ of the server node, the message structure is defined in calling parameters, the client object is passed in the calling parameters so that the server can trigger its callback when the service is done.
- The callbacks (e.g. *on_service_triggered*, *on_service_done*) are defined as $msgsrv$ interfaces of the nodes (which can be either the server or the clients).

The below code extracts will help understand clearly:

```
reactiveclass PublisherNode{
	knownrebecs{
		ReceiverNode r1;
		ReceiverNode r2;
	}

	PublisherNode(){
		sendTopicData_topic1(r1,100);
		sendTopicData_topic1(r2,200);
	}

	//sending topic data
	msgsrv sendTopicData_topic1(ReceiverNode r, int rate){
		int data1,data2;
		//prepare data ...

		//send data
		r.onTopicData_topic1(data1,data2);

		//schedule next sending
		sendTopicData_topic1(r,rate) after(rate); //receiver node is passed so that multiple receivers are allowed
	}
}

reactiveclass ReceiverNode{
	knownrebecs{
		PublisherNode theDataProvider;
	}

	ReceiverNode(){
	}

	//topic callback
	msgsrv onTopicData_topic1(int data1, int data2){
		//define the behaviour here
	}
}
```

Figure 11: One-to-many pub-sub topic modelling

```
reactiveclass Topic(20) {
	statevars{
		int topicName;
		Node[5] subscribers;
		int countSubscribers;
	}

	Topic(int t){
		topicName = t;
		countSubscribers=0;
	}

	msgsrv subscribe(Node node){
		if (countSubscribers<5){
			subscribers[countSubscribers]=node;
			countSubscribers++;
		}
	}
	msgsrv unsubscribe(Node node){
		for(int i=0;i<countSubscribers;i++){
			if (subscribers[i] == node) {subscribers[i]=null;break;}
		}
	}
	msgsrv send(int[10] topicMessage){
		for(int i=0;i<countSubscribers;i++){
			if (subscribers[i]!=null)
				subscribers[i].onTopicMessage(topicName,topicMessage);
		}
	}
	msgsrv perodic_send(int rate){
		//prepare data to send
		int[10] topicMessage;
		send(topicMessage);
		perodic_send(rate) after(rate);
	}
}
```

Figure 12: Many-to-many pub-sub topic modelling

```
reactiveclass ServerNode(20){
	knownrebecs{
		ClientNode r1;
		ClientNode r2;
		//other clients
	}

	msgsrv doService_srv1(ClientNode client,int[10] request){
		int[10] response;
		//do service with input arguments in `request` and generate outputs in `response`

		//trigger client's callback on done
		client.onServiceDone_srv1(response);
	}
}

reactiveclass ClientNode(20){
	knownrebecs{
		ServerNode theServiceProvider;
	}

	ClientNode(){

	}

	msgsrv onServiceDone_srv1(int[10] response){
		//response on service done
	}

	msgsrv doSomething(int[10] request){
		theServiceProvider.doService_srv1(self, request);
	}
}
```

Figure 13: Service interface modelling

8.3. Map representation & occupancy grid

In ROS2, the environment map (floor map) can be scanned & constructed dynamically by a SLAM robot or prepared beforehand. A SLAM robot crawls around the floor, using its range detection sensors to measure distances to surrounding obstacles and construct the floor map in a PGM (portable gray map) image, where each cell is marked with a gray level value representing the occupancy status (from 0=black, surely occupied, to 255=white, surely unoccupied).

Map files are usually stored in a /maps folder in a ROS2 codebase as a pair of a PGM image file and a YAML file for describing the map (metadata):

/maps/floor1.pgm
/maps/floor1.yaml

The YAML file defines a few parameters for using the map:

- Resolution: scale of cell width to meters in reality
- Threshold values (*occupied_thresh*, *free_thresh*) to tell which gray level should be treated as occupied (dark enough = gray level ≤ *occupied_thresh*) or unoccupied (transparent enough = gray level ≥ *free_thresh*). Gray level (normalized) = raw gray value / 255 = a real number from 0.0 to 1.0.
- Map origin: $[x, y, \theta]$ pose of the map origin in the world's frame, θ is the rotation angle (0 = no rotation, often ignored). ROS2 uses a right-handed system (the quadrant I (x>0, y>0) is on the right, while in a more commonly used left-handed system it is on the left. This matter will be explained carefully in the next part). The map origin information is ignored as it is set in the code so that the view port is at the center of the map.

Example of an occupancy grid generated from a PGM map:

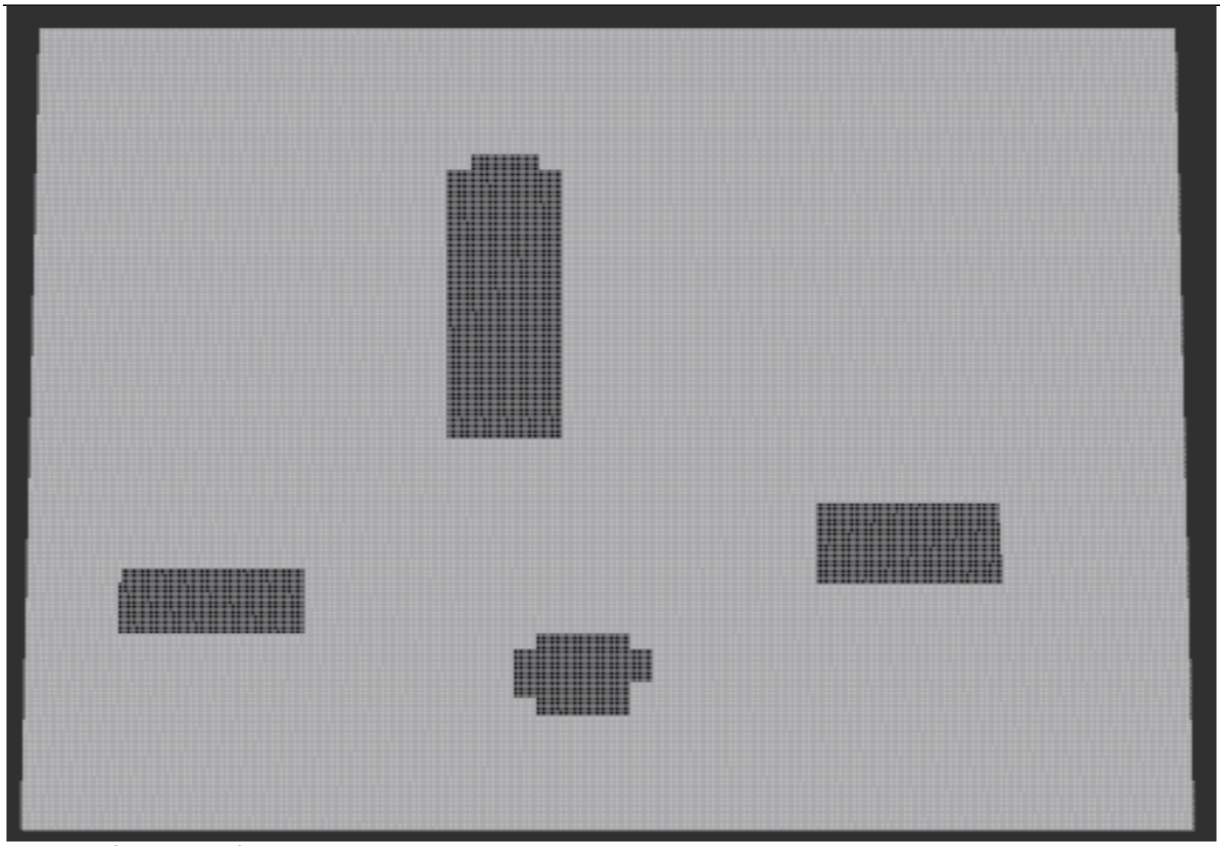


Figure 14: Screenshot of an occupancy grid displayed on RViz2

and a YAML file:

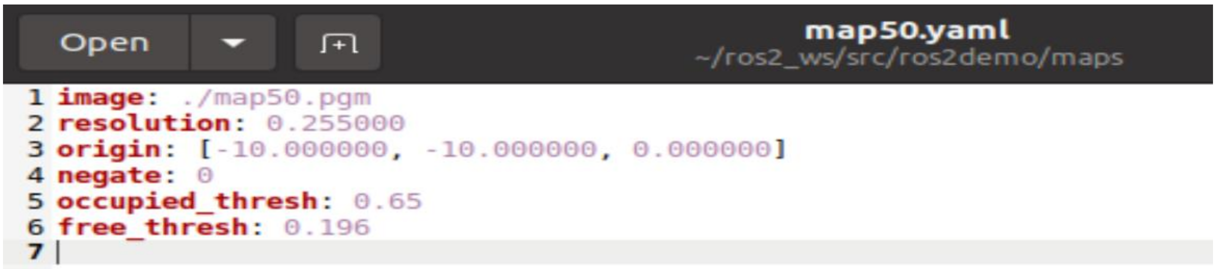


Fig.15: Screenshot of a YAML map metadata file

To represent the obstacles, the floor map is converted to an occupancy grid. Run-time positions of mobile robots need to be updated to this occupancy grid to facilitate dynamic obstacle detection & generation of *collision-aware* paths. To prevent the robots from getting too close to obstacles, the boundaries of obstacles ("footprints") are expanded with some extra cells ($= ROBOT\_WIDTH/2 + SAFE\_MARGIN$) so that the generated paths are shifted some distance from the obstacles.

I use the term "*collision-aware*" here, not "*collision-free*" because the path is only collision-free at the time it is generated, not *future-proof*, since mobile obstacles can move and appear unexpectedly in the future.

For external mobile obstacles (like a wandering human being or animal), we cannot tell them to report their location to this map server node; in real life the robots have to detect and cope with them dynamically. In modelling and simulating, we need to have a map server to keep track of the occupancy status so that we can model obstacle detection and path generation.

I use a middle script (written in PHP) to help generate the occupancy grid from the PGM map, and copy the generated array to Rebeca. This middle script also helps to prepare, visualize and troubleshoot other data and functions to bridge the gap between implementation language and Rebeca modelling language.

One note-worthy point here is that the map of static obstacles should be loaded into a local variable rather than a state variable, because it is unchanged data and of large size which will cause unreasonable explosion of the state space. The locations of mobile robots are updated dynamically to the grid; the map server node provides a service point for the robots to report their locations to.

8.4. Modelling robot pose

A moving robot has a location and a direction that it currently looks forwards (orientation), represented by a triple of (x, y, θ) where (x, y) is the coordinates relative to the map origin and θ is the angle of the robot's movement axis *Xr* relative to the *X* axis of the map (positive = anti-clockwise, negative = clockwise). These two components form the pose of the robot.

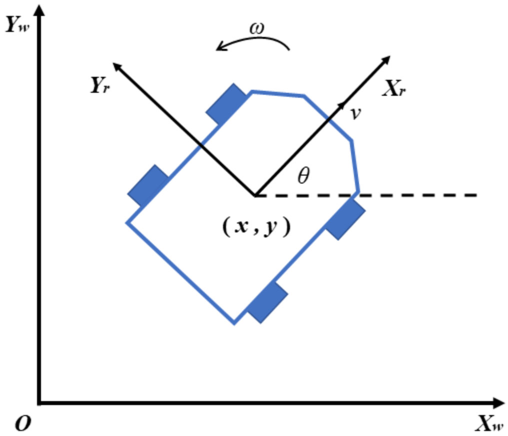


Figure 15: Illustration of a robot pose and velocities (source: https://www.mdpi.com/2076-3417/11/1/426)

We have to model both. In simplistic cases, a robot is supposed to move only horizontally or vertically, but this assumption is not enough to ensure modelling correctness as in reality most robots can turn to an arbitrary angle. Meanwhile, in model checking, we can only support discrete values, not continuously smooth values, which is known as the *model-reality gap*. How can we survive this gap and still capture enough information to guarantee the correctness of the model? *Discretization* is the strategy.

In an occupancy grid, it is enough to track only 08 possible directions: horizontally, vertically, and diagonally.

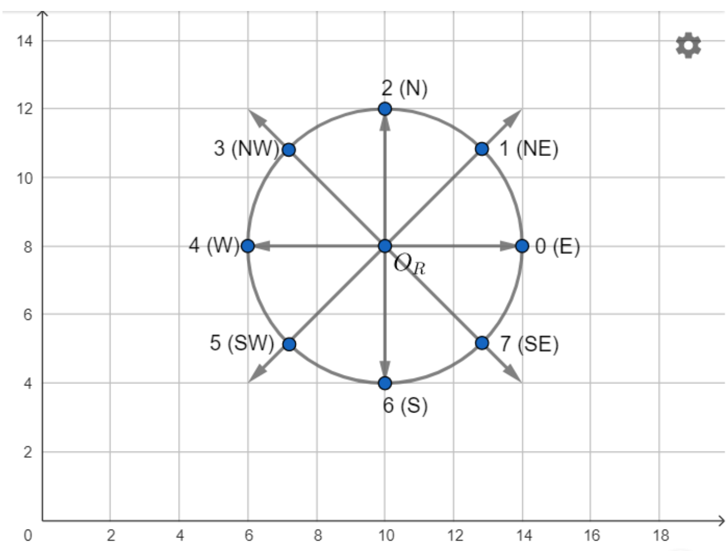


Figure 16: Discretizing robot directions

These 8 angles are multipliers of 45º and are very easy for trigonometric computations. This way we discretize the rotation from an arbitrary angle to only 8 known directions:

TABLE 1: DISCRETIZATION OF ROBOT DIRECTIONS

Direction	Angle	Name
0	0º	E
1	45º	NE
2	90º	N
3	135º	NW
4	180º	W
5	215º	SW
6	270º (or -90º)	S
7	315º (or -45º)	SE
8	360º	E (one full circle)

So a real-world robot pose represented by real numbers (x, y, θ) is mapped to $(rx, ry, rdir)$ with discrete integer values in the model. The direction value needs to be normalized to non-negative only as below:

```
int normalizeDir(int dir){
    dir%=8;
    if (dir<0) dir+=8;
    return dir;
}
```

After discretizing robot's directions to just 8 known directions, the translation between map coordinates and robot coordinates is easy and fast (because the trigonometric values of these 8 angles can be predefined):

```
/*
shifting:
        x' = x-a
        y' = y-b
rotating:
        x' = x*cos(theta) + y*sin(theta);
        y' = x*-sin(theta) + y*cos(theta);
*/
int[2] getNewCoord(int x, int y, int a, int b, int dir){
        dir = dir % 8;
        if (dir<0) dir+=8;

        //theta = 0,45,90,135,180,235,270,360
        //cos = x, sin = y, tan = y/x

        //use ENV for these constant arrays will cause the model not compilable, don't know the reason
        double[8] sins = {0.,SIN45,1.,SIN45,0.,-SIN45,-1.,-SIN45};
        double[8] coss = {1.,COS45,0.,-COS45,-1.,-COS45,0.,COS45};

        int[2] p;
        x-=a; y-=b;
        p[0] = round(x*coss[dir] + y*sins[dir]);
        p[1] = round(-x*sins[dir] + y*coss[dir]);

        return p;
}
```

Figure 17: Code extract of coordinate transformation function in Rebeca

To know the relative position of an obstacle (which is located at (ox, oy) in the map) to the robot's current pose $(rx, ry, rdir)$, we simply call:

$$int[2]\, p \;=\; getNewCoord(ox, oy, rx, ry, rdir);$$

8.5. Modelling the velocity

The two previous works by Saed and Kostiantyn all ignored the robot's velocity, which results in unfaithful modelling of real world behaviors since collision probability and avoidance strategy require consideration of the speeds of the robots. We need to track *where* and *when* robots can be to know the possible crossing sections. That's why speed has to be modelled.

There are two kinds of velocity:

- Linear velocity (in meters per second): indicates how fast a robot travels along a line.
- Angular velocity (in radians per second): indicates how fast a robot rotates.

In modelling, we can ignore the angular velocity, assuming that the time it takes a robot rotating to an angle can be ignored. To move to a point, a robot first rotates to the right angle towards that point (simply modelled by setting the direction to the desired value: $rdir = dir;$), then moves straightly towards that point. The direction of movement is already modelled as above; the magnitude of the linear velocity is converted to equivalent value on the occupancy grid (from *meters per second* to *number of cells per second*):

cell_speed = round(real_speed / map_resolution)
*real_speed = cell_speed * map_resolution*

So given map resolution is 0.05 (1 cell = 0.05 meters = 5 cm), *real_speed* = 0.5m/s → *cell_speed* = 0.5 / 0.05 = 10 cells/second.

We model the speed of the robot by converting to the time (in *milliseconds*) it takes the robot to travel over one cell by its edge (horizontally or vertically - directions 0, 2, 4, 6) or by its diagonal (directions 1, 3, 5, 7):

time_to_cross_edge = 1000 / *cell_speed* = 1000 / 10 = 100 ms
time_to_cross_diagonal = *sqrt*(2) * *time_to_cross_edge* = 1.4142 * 100 = 141 ms

Timed Rebeca offers timing primitives like $delay$() or $after$() that can facilitate modelling speed as consumed travelling time (with $delay$() the time period has to be a predefined constant, while with $after$() it can be a locally initialized variable – for this reason, $after$() is more preferable as it is more flexible to set different speed for each robot).

8.6. Robot dimensions

To some extent, a robot can be simply represented as a non-dimensional point on the map, but that simplification is only justifiable when the robot can fit into one grid cell and its dimensions can be ignored. In ROS2, the physical structure of a robot is defined in a URDF file (normally put in /urdf or /models folder).

URDF stands for Unified Robot Description Format, which is an XML format used to describe a robot model. It is commonly used in robotics for simulation, visualization, and control. The URDF file contains information about the robot's links, joints, sensors, and other properties. It can also include information about the robot's physical dimensions, such as its mass, inertia, and geometry.

To extract the robot dimensions from a URDF file, we can look for the "link" and "joint" elements in the file. The "link" element describes the physical properties of a robot's link, such as its mass and geometry. The "joint" element describes the relationship between two links and can provide information about the position and orientation of the links. By analyzing the "link" and "joint" elements in the URDF file, we can extract the robot's physical dimensions, such as its length, width, height, and weight.

In modelling, we only need to define two dimensions: $robot\ width$ and $robot\ length$. The robot's origin is assumed to be at the center. Two extra distances are useful to determine movability and obstacle avoidance:

- $SAFE\_MARGIN$: some extra margin to the sides of the robot so that it knows that it can move if there are no obstacles in the front zone.
- $STOP\_ZONE$: distance to its most-frontal point, if an obstacle appears within this stop zone, the robot will have to stop as the obstacle is too close. Outside the stop zone is the warning zone, the robot may have different ways of handling obstacles in different ranges.

These useful dimensions are depicted in the below drawing:

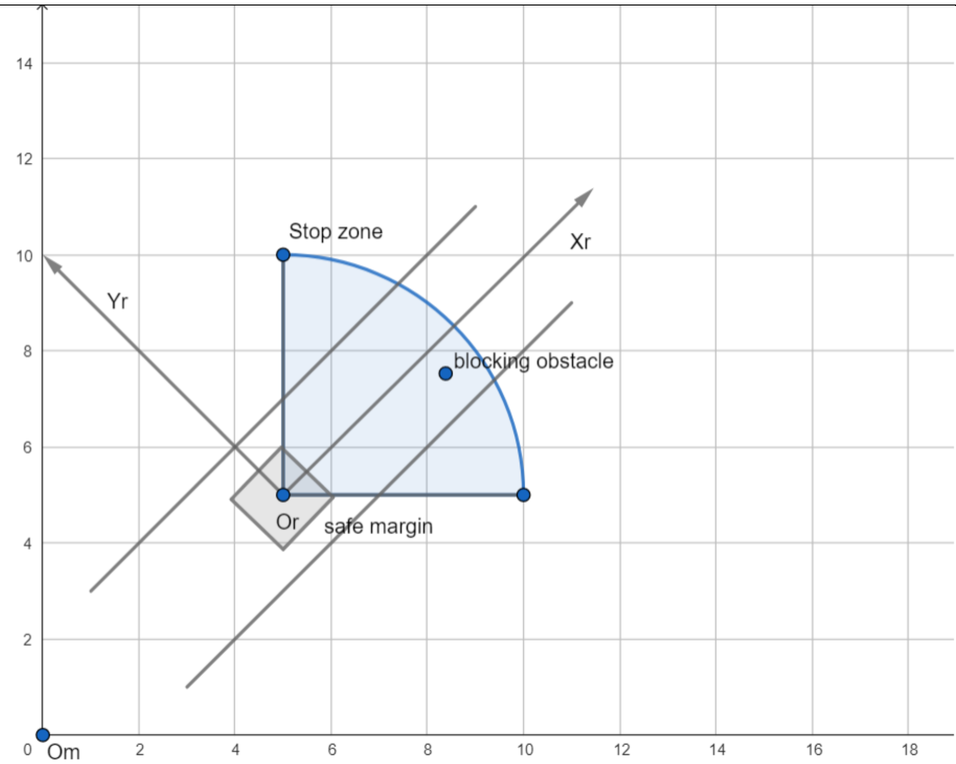


Figure 18: Robot dimensions, safe margin, stop zone, field of view

It is not true that any obstacle appears within a radius to the robot can block its way, it depends on the relative location of the obstacle to the robot's movement direction. So we can see that, if an obstacle is close to the robot but is outside the front area, the robot still can pass through it. The logic of detecting blocking obstacles depends on the developer's idea; these definitions are only what we devise. Some robot avoids stopping, giving priority to slowing down and rerouting to cut through the behind of a mobile obstacle (like the example of the AGV at the airport), others may give priority to safety by stopping if any obstacle appears within a radius. The way of responding to obstacles depends on the operational context, speed, braking distance, and other physical properties & priority of the robot. My approach in modelling is only one way to tackle.

8.7. Obstacle detection

ROS2 robots can use a variety of sensors for detecting obstacles, depending on the specific robot and the environment in which it operates. Some common types of sensors include:

- LiDAR (Light Detection and Ranging): LiDAR sensors use lasers to measure distances and create a 3D map of the environment. They are commonly used for mapping, localization, and obstacle detection.

- Cameras: Cameras can be used for visual detection of obstacles, such as people or other robots. They can also be used for object recognition and localization.

- Ultrasonic sensors: Ultrasonic sensors use high frequency sound waves to detect obstacles and measure distances. They are commonly used for short-range obstacle detection.

- Infrared sensors: Infrared sensors detect obstacles by measuring the heat emitted by objects. They are commonly used for proximity detection.

These sensors can be used in combination to provide more comprehensive obstacle detection and avoidance. For example, a robot may use a combination of LiDAR and cameras to detect obstacles in its environment, and use the data from these sensors to plan a safe path through the environment.

LiDAR or laser sensor is the most commonly used type of sensor for obstacle detection in ROS2. Here's how it works:

- Emitting laser pulses: The LiDAR device emits a series of laser pulses that travel at the speed of light towards the target object or surface.
- Reflection and detection: When the laser beam hits the target, it reflects back towards the LiDAR sensor. A portion of the reflected light is then detected by a highly sensitive photodetector.
- Time-of-flight calculation: The time it takes for the laser pulse to travel to the target and back to the sensor is recorded by the LiDAR device. This measurement is called "time-of-flight" and it determines the distance between the sensor and the target.
- Creating a 3D map: The LiDAR sensor emits laser pulses at different angles and positions to create a 3D map of the target object or environment. By combining these measurements, LiDAR systems can create highly accurate and detailed 3D models of any object or environment.

The figure below shows an example of a 90° LiDAR sensor when the robot is looking East.

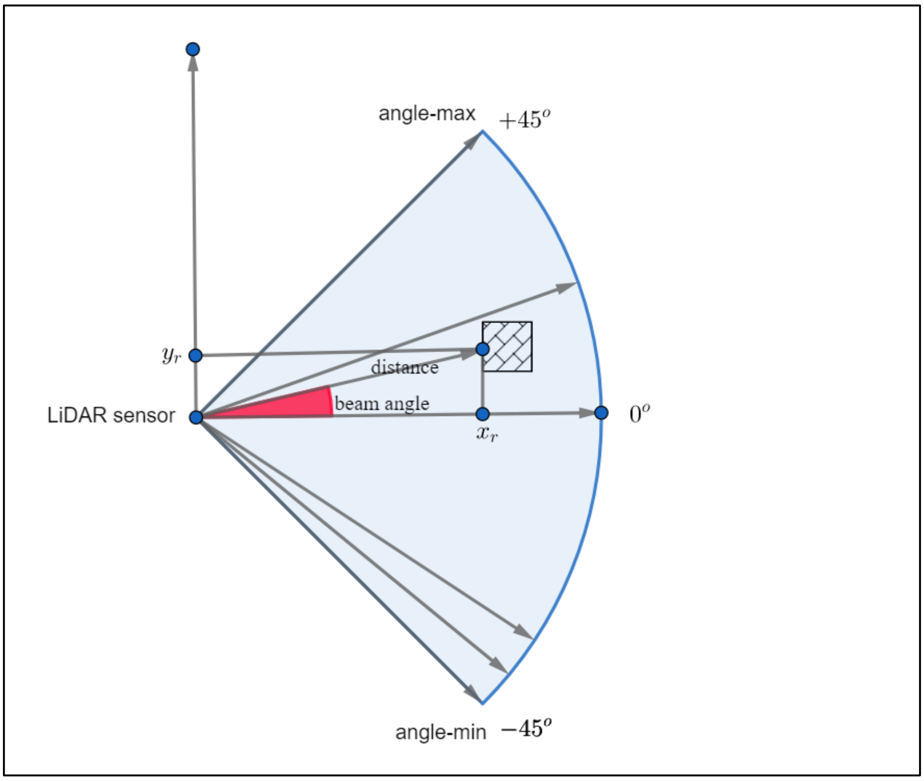


Figure 19: Laser beams in a LIDAR sensor

From the beam angle and detected distance, the coordinates of detected obstacle relative to the robot’s axises can be computed by:

$$xr = distance * cos(beam\_angle)$$
$$yr = distance * sin(beam\_angle)$$
$$(angle\_min \leq beam\_angle = angle\_min + i * angle\_increment \leq angle\_max)$$

ROS2 wraps laser scan data in **/sensor_msgs/msg/LaserScan** message structure (http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/LaserScan.html)

File: sensor_msgs/LaserScan.msg

Raw Message Definition

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Compact Message Definition

```
std_msgs/Header header
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Figure 20: Screenshot of LaserScan message structure (source: docs.ros.org)

The laser sensor broadcasts range data in **/laserscan** topic and ROS2 nodes subscribe to this channel to capture.

Modelling obstacle detection mechanism is essential to guarantee correct reflection of the way it works in reality.

There are a number of challenges in modelling such in Rebeca:

- Having to discretize real numbers to integers while making sure that it provides the same scanning result.
- Making geometric computations light-weighted so that the model checker can work, as laser scanning event is repeated periodically at high frequency (normally 10-30Hz ~ 100-33ms).
- Rebeca does not provide very basic mathematic and trigonometric calculation functions (like *sqrt*, *tan*, *sin*, *cos*). Efficient workarounds must be devised.
- Adapting to limitations in array size and fixed-size arrays in Rebeca (we found out that Rebeca can only handle one-dimension array of ~100 elements as function parameter or returned value, it generated unclear errors when using arrays of size over 100 or multiple dimensions).
- Providing detection data for convenient use in obstacle detection and avoidance algorithms in Rebeca, while facilitating the same decision making as in ROS2 code.

> It is not going to be 1-on-1 equivalence between data structures in ROS2 and Rebeca, but the same semantics is converted so that the same decision, the same algorithm can be made. In other words, it is *semantic equivalence*, not *literal equivalence*.

I have successfully modelled laser scanning in Rebeca this way:

- Angle increment: 2º is enough to detect two adjacent cells on the grid (*atan*(1.1) – *atan*(1) = 0.047 radian = 2.7º) (1º provides smoother scans but doubles the loop and storage size, while 3º step starts to be too wide that can result in skipping adjacent obstacles).
- **Precomputed outputs**: as in the discrete model, all possible angles are integers from 0 to 360º, so they can be precomputed using the PHP helper script and put into a cache (array) to look up so that all related trigonometric calculations can be used straightly, without having to calculate on the fly (this way the limitations of Rebeca are overcome and computations are much more rapid).
- Field of view: typically, a laser sensor can cover an angle of 120º – 270º, to achieve 360º view the robot can combine 03 sensors or use a rotating sensor. The robot's field of view is defined as a constant, and can only be a multiplier of the angle increment (like 120º, 180º, 270º, 360º).
- The maximum range should be just enough to detect near-enough obstacles that the robot needs to care about (like 20-50 cells), larger value results in longer loop while far-away obstacles are out of concern.

The algorithm for imitating laser scanning can be implemented as below (in PHP, adaption is needed to make it work in Rebeca):

```php
<?php
0 references
function simulateLaserScan($grid, $origin, $numAngles, $maxRange, $angleMin, $angleMax, $angleIncrement) {
  // Get the dimensions of the grid
  $rows = count($grid);
  $cols = count($grid[0]);

  // Convert the minimum and maximum angles to radians
  $angleMinRad = deg2rad($angleMin);
  $angleMaxRad = deg2rad($angleMax);

  // Initialize an array to store the ranges for each angle
  $ranges = array_fill(0, $numAngles, $maxRange);

  // Iterate over each angle
  for ($i = 0; $i < $numAngles; $i++) {
    // Calculate the current angle
    $angle = $angleMinRad + $i * deg2rad($angleIncrement);

    // Initialize the range for this angle to the maximum range
    $range = $maxRange;

    // Calculate the step size for each ray
    $step = min($rows, $cols) / $maxRange;

    // Iterate over each step along the current ray
    for ($j = 0; $j < $maxRange; $j += $step) {
      // Calculate the current position along the ray
      $x = $origin[0] + $j * cos($angle);
      $y = $origin[1] + $j * sin($angle);

```

Figure 21: (a) Code extract of laser scan simulation function

```
      // Check if the current position is within the grid
      if ($x >= 0 && $x < $rows && $y >= 0 && $y < $cols) {
        // Check if the current position is an obstacle
        if ($grid[floor($x)][floor($y)] == 1) {
          // Update the range for this angle
          $range = $j;
          break;
        }
      } else {
        // If the current position is outside the grid, set the range to the maximum range
        $range = $maxRange;
        break;
      }
    }

    // Store the range for this angle in the ranges array
    $ranges[$i] = $range;
  }

  // Return the ranges array
  return $ranges;
}
```
Figure 22: (b) Code extract of laser scan simulation function

For convenience, the scanning result in Rebeca is instead provided in 02 pieces:

- An array of (*x*, *y*) coordinates of the detected obstacles (relative to map origin): the range is computed by Euclid distance from the robot origin to the obstacle's location. The (*x*, *y*) coordinates are packed into one integer to use one-sized array instead of two. If we need to know the relative position to the robot's origin, we just need to translate the coordinates from map's origin to robot's origin.
- An array of corresponding beam indexes (so that we know the angle of the obstacle to determine which direction it may block)

The periodic scans are modelled as below:

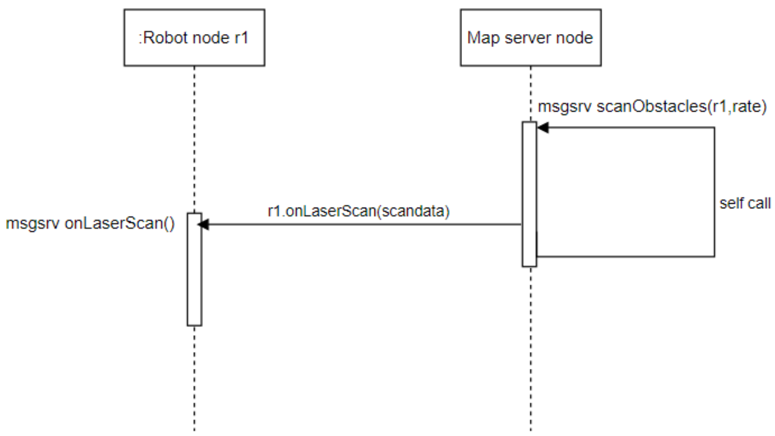


Figure 23: Laser scan events

```
reactiveclass MapServerNode {
    knownrebecs{
        RobotNode r1;
        RobotNode r2;
        //...
    }
    MapServerNode(){
        //start laser scans
        scanObstacles(r1,1,140,180,20);
        scanObstacles(r2,2,140,180,20);
        //...
    }

    msgsrv scanObstacles(Node robot,int rindex,int rate,int fov,int max_distance){
        //simulate range detection here
        int[101] scandata;

        //trigger the robot's callback on arrival of new laser scan
        robot.onLaserScan(scandata) deadline(rate+5);

        //schedule next scan, with previously assigned parameters
        scanObstacles(robot,rindex,rate,fov,max_distance) after(rate);
    }
    //...
}

reactiveclass RobotNode{
    knownrebecs{
        MapServerNode theMap;
    }

    msgsrv onLaserScan(int[101] scandata){
        //callback at arrival of new laser scan
    }
}

```

Figure 24: Code extract of laser scan event handling in Rebeca model

8.8. Path generation

A* is a popular and efficient path searching algorithm in games and robotics. ROS2 navigation components like NavFn, Smac 2D planner all implement A*-based algorithms to find paths on an occupancy grid. Unfortunately, path finding is an optimization problem whose solution cannot be easily guessed by a simple linear function. We therefore have to implement a similar path finding algorithm in Rebeca to be able to generate a walkable path from a starting point to a goal point for each robot. Under different contexts, the robots will encounter different situations in different timing to handle.

The path generation service is provided by the map server node so that a robot node can send a request to with its own situational parameters and get back the generated path in an asynchronous response. The service is modelled as a $msgsrv$ of the map node, and the callback is modelled as a $msgsrv$ of the robot node.

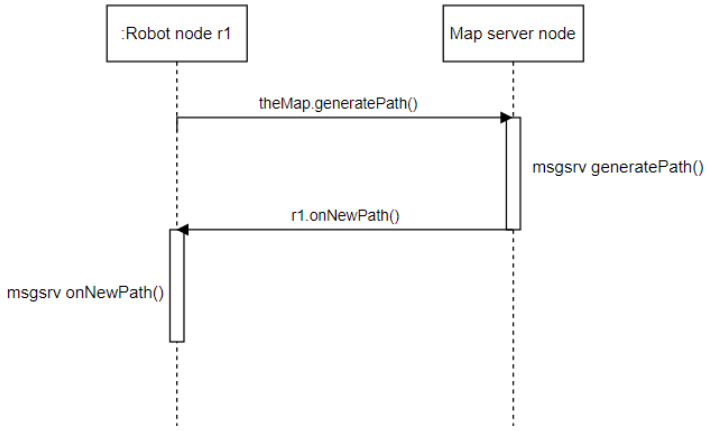


Figure 25: Path generation events

To generate collision-aware paths, following improvements have been introduced:

- Obstacle expansion: as mentioned, the footprints of the static obstacles are expanded to count for the dimensions of the robot so that the generate path is shifted away from the boundary and the robot does not get too close to hit.
- Dynamic locations of mobile obstacles (robots): locations of mobile obstacles are updated dynamically to the map so that the path avoids hitting them in their current locations.
- Consideration of future collisions: if another robot is moving, its frontal cells will be marked as occupied so that the current robot avoids cutting through the frontal area (otherwise it will soon end up encountering that mobile obstacle and have to stop again and replan the path).

8.9. Modelling robot movements

A mobile robot has 02 kinds of basic movements: rotation and linear motion. It combines these two movements to travel to a point. To be able to model, the robot's movement is simplified to two steps:

- First, it rotates to look towards the waypoint.
- Second, it moves in that direction to reach to the point, the angle is not changed during the linear movement.

The process in reality happens gradually, incrementally with real number representation of current location, movement speed, travelled distance. In a discrete model, this process happens abruptly by setting the angle or location to the desired value, the "process" is modelled as a delay same as the time it takes to accomplish the movement:

- Rotate: $rdir = getNextDir();$
- Move: $rx = nextpoint.x;\ ry = nextpoint.y;\ delay(time\_to\_travel);$

The rotation time is ignored (for simplicity); the time to move to a point can be put before or after setting the new location of the robot.

Following state variables are needed to track the robot's movements:

- A triple of $(rx, ry, rdir)$ to track current pose
- A queue of targets (destinations): the last one is the final goal to reach, others are intermediate targets.
- A queue of moves (waypoints) and a pointer to track current position in the queue.
- Variables to track waiting status and waiting time: $boolean\ isWaiting;\ int\ waits;$

There are two movement modes (set by $scan\_before\_move$ boolean variable)

- Scan before move (safety-first): this way, the robot waits for the arrival of a new laser scan before making next move. This mode is safer but slower. This mode is not absolutely safe, as while the robot moves, a collision still can happen.

- Move and scan in parallel: the robot moves on without waiting for a new laser scan, it uses the most updated scan result to make decision. This mode is less safe but faster, and requires fast enough scanning rate to catch up with movement speed.

Overall movement modelling scheme can be explained by below pseudo-code:

```
msgsrv onLaserScan(scandata):
    //update array of obstacles
    obstacles = scandata;
    //only update moving status when not reached target yet
    if (waiting_in_progress OR (scan_before_move && not reached target yet))
        updateMovingStatus(!scan_before_move);

msgsrv updateMovingStatus(boolean movenext):
    if (robot has reached to nearest target in queue){
        stop();
        targets.pop_back();
        if (!targets.empty()){
            theMap.generatePath(rx,ry,targets.get_back());
        }
    } else {
        //rotate robot to next way point;
        rdir = getNextDir();
        //check if the robot can move in this direction
        if (canMoveAtDir(rdir)){
            //calculate time to travel from current (rx,ry) location to next waypoint
             int travel_time = time to travel to next waypoint;
             //set robot location to next waypoint
             rx = nextwaypoint.rx;
             ry = nextwaypoint.ry;
             if (movenext) updateMovingStatus(movenext) after(travel_time);
        } else {
            stop();
            //start waiting or continue waiting if waiting is in progress
            isWaiting = true;
            waits++;
            //wait long enough
            if (waits > max_waiting_time){
                moves.clear();
                theMap.generatePath(rx,ry,targets.get_back());
            }
            //continue waiting, next update will be triggered by onLaserScan() event
        }
    }
```
Figure 26: Message servers for handling robot movements in Rebeca model

Detailed implementation needs to be checked in the model's code. The main idea to capture here is the circulation of events and the conditions to consider, the decision to make corresponding to each situation.

8.10. Collision avoidance & congestion resolution

The traffic problem of multiple autonomous mobile robots is very much similar to cars running on crossing streets in real life. To come to a collision avoidance strategy for the robots, I refer to the ways a human handles traffic in reality:

- When moving, I look forwards to the frontal area, if an obstacle appears in the way I am going to, I estimate the time/distance to encounter it: if it is too close, I stop to wait for it to pass; if it is still far away, I move on.
- In case after waiting long enough but the obstacle does not get out of my way, I try to find another way to skip the obstacle, often cutting through its behind to get through, and continue towards my destination.

The robot should learn from human's wisdom, as it is supposed to work autonomously, to make decisions on its own like a human.

So to make a move:

- First, the robot rotates to the direction towards the next turning point on the generated path (waypoint)
- Second, it checks among the most up-to-date array of obstacles, to see if any obstacle is blocking its frontal area
- If there is enough free space in the frontal area, the robot moves. Otherwise, it stops and starts waiting.
- If it has been waiting long enough, the robot tries to generate a new path to skip the obstacle and continue towards its goal.

The above logic leads to the use of following state variables and algorithms for the mobile robot:

- An array to keep track of the absolute positions of surrounding obstacles. This array is updated on the arrival of a new laser scan. The coordinates of obstacles are expressed in the map frame.
- State variables to track the waiting status:
    - $int\ max\_waiting\_time$: maximum waiting time (in Rebeca it has to be converted to maximum waiting rounds in terms of laser scan intervals because a clock function is not provided).
    - $boolean\ isWaiting$: a boolean variable to check if the robot is in progress of waiting
    - $int\ waits$: total waiting time so far. If it exceeds $max\_waiting\_time$, the robot will reroute its way.
- To check the relative positions of scanned obstacles to current pose of the robot, the absolute coordinates of the obstacles in the map frame (ox, oy) need to be converted to the robot's current frame (ox', oy') (using the above-mentioned knowledge of coordinate transformations). An obstacle is a blocking one if it lies in the frontal area within the movement lane:

$$is\_blocking\_obstacle\ = ox'\ >\ 0\ \&\&\ |oy'| \leq\ ROBOT\_WIDTH/2\ +\ SAFE\_MARGIN$$

  The free frontal space is the distance to the nearest blocking obstacle = $min(ox')$

- These functions will help the robot make decision:
    - $getNextDir()$: returns the direction to the next waypoint
    - $getObstacleAtDir(int\ dir)$: returns the distance to the nearest blocking obstacle in that direction dir

- To check if it can move in a direction, the free space in that direction must be longer than some safe distance (stopping distance):

$$canMoveAtDir(int\ dir)\ =\ getObstacleAtDir(dir)\ > (ROBOT\_LENGTH/2\ +\ STOP\_ZONE)$$

The figure below will help to understand the algorithm:

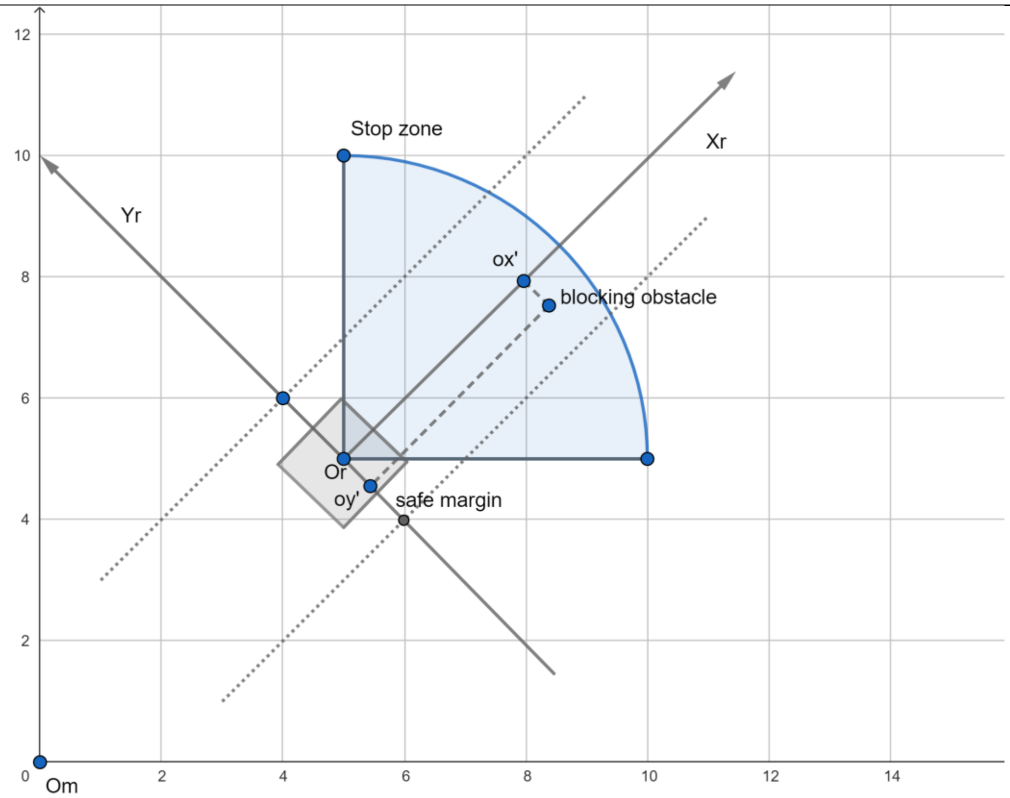


Figure 27: How to estimate nearest obstacle at a direction

Thinking that way has led to an efficient algorithm to avoid collision. During the process of implementing the algorithm, following interesting behaviors are observed:

- If the obstacle array is not updated fast enough to catch up with the moving speed, the robot will use outdated information which is no longer valid for the new situation and a collision with another robot is likely to happen. This agrees with the Nyquist-Shannon sampling theorem: "*The sampling frequency of a signal must be strictly greater than twice the highest frequency of the input signal*".

The laser scan rate therefore must be smaller than half of the time it takes to move a step. For example: if linear speed is $1.0m/s$, map resolution is $0.255(m/cell)$, then it takes $1000 * 0.255/1.0 = 255ms$ to cross a cell by its edge, then the laser scan rate must be $< 255/2 = 127ms$. If laser scan rate is fixed by the hardware (e.g. $140ms$), then $time\_to\_cross\_edge > 2 * laserscan\_rate$, so $linear\_speed < 1000 * map\_resolution / 2 * laserscan\_rate = 0.91m/s$

- In terms of waiting time: if $max\_waiting\_time$ is too short, the robot will be too hasty in rerouting while waiting a bit further is better. Unnecessary reroutings will extend the time to reach target and may cause the robots to get to a new crossing situation again. If $max\_waiting\_time$ is more than the time needed to wait for the other robot to cross, longer waiting will not help when the other robot keeps waiting as well (deadlock situation).

- **Mexican standoff**: [Wikipedia]: "*A Mexican standoff is a confrontation where no strategy exists that allows any party to achieve victory. Any party initiating aggression might trigger their own demise. At the same time, the parties are unable to extract themselves from the situation without suffering a loss. As a result, all participants need to maintain the strategic tension, which remains unresolved until some outside event or interparty dialogue makes it possible to resolve it.*". When the robots are cloned objects from the same class, they are coded to think and behave the same way, when two or more robots get too close to each other, they all see each other, stop and wait for the others to get out of their way. Rerouting does not help if the same path is generated or not found (as the way has been blocked). This situation is the same as a daily traffic congestion at a crossroad where there is no traffic light, no round-about, no traffic controller. That means, there must be a way to recognize and resolve a congestion situation – either by the robot itself or under the coordination of a traffic controller. The known mechanisms in solving congestions can be learnt and adopted: traffic light, roundabout, priority tickets, etc.

Fig.26: A Mexican standoff (source: https://en.wikipedia.org/wiki/Mexican_standoff)

I devised some way to resolve congestions (since the mission of this project is not to work out a very clever algorithm in obstacle avoidance and congestion resolution, it just needs to be something that works as a proof-of-concept so that the robots can get out of the deadlocks and reach their goals):

- When the robot waits enough, it starts to try to get out of the congestion situation.
- The robot moves back in another direction with least obstacles a number of steps (*backoff steps*), then asks the map server to generate a new path.
- Each time of rerouting, a counter of failures should be triggered. Each time a new path is generated, the counter is decreased, and each time a new path is not found, the counter is increased. When the counter exceeds some threshold, a deadlock situation should be alerted – that means the robot cannot get out of the (repeated) congestion(s) or cannot reach target (ways are blocked), despite keeping backing off and rerouting.

The following phenomena are observed in such way of resolving congestion:

- If the robots back off the same steps in the same direction each time (e.g. the reverse direction), the same paths will be generated and again they will get stuck again.
- **Heterogeneity** (dissimilarities, variations, differences) should be introduced to vary the robots' behavior patterns so that they do not behave the same way. That equals to some variations in each robot's working parameters (speed, maximum waiting time) and randomizations in backoff direction, backoff steps will lead to less chance of collision and more chance of getting out of congestion.
- Path generation: the shortest path is not always the best path if it leads to unresolved congestions or situations that end up in impossibility to get to goal. The optimization algorithm in path finding may end up in generating the same path every time. This observation opens up a research to find ways to improve the path generating algorithm (e.g. keep track of generated paths to avoid reusing, allow more parameters to differentiate optimization results, etc.)

8.11. Overcoming programming limitations of Rebeca

As mentioned, Rebeca is a modelling language, its code is interpreted to another compilable language (C++) to create an executable model. Only a few normal programming facilities are supported. I have to invent and apply a number of techniques to overcome the limitations of Rebeca:

8.11.1. Handling dynamic arrays

Rebeca only allows arrays with predefined and fixed size, which is very inconvenient. To allow dynamic-size data, there are several ways:

1. Mark the ending element with a special value: similar to NULL-terminated strings, the drawback of this method is that each time referring to the size of the array we have to traverse from the beginning until finding the terminator value, and the terminator value if chosen not carefully can collide with a possible ordinary value.
2. Use the first element to store the number of elements, the rest are the actual elements.

Method #2 is used as I found it to be more convenient and performant. For example:

```
int[101] a; //buffer size = maximum size + 1
//initialize
for(int i=0;i<101;i++) a[i]=0;

//append
a[0]++;
a[a[0]] = 100;

//pop last element
int last = a[a[0]];
a[0]--;
a[a[0]] = 0;

//loop, indexing starts from 1 to a[0]
for(int i=1;i<=a[0];i++){
    //a[i]
}

//check not empty
if (a[0]>0){
    //...
}
```

Figure 28: Code extract of handling dynamic arrays in Rebeca

8.11.2. Combining numbers

Rebeca does not support structured data with fields of different data types like a $struct$ in C++. It is only possible to store as a sequence of numbers of the same type. Very often in the model we have to pass a sequence of coordinates (x, y) (like waypoints in a path, array of obstacle coordinates). It is more convenient to encode each pair as one element to pass and decode to get back the individual numbers. Some simple encoding scheme is invented (works for integers < 1000):

- $(45,5)$ is encoded to 10051045, $(-45,-5)$ is encoded to 200520045: for each part, the first digit marks the sign (1 for positive, 2 for negative), the remaining 3 digits for actual value.
- Largest number can be encoded is 999, which is enough to handle supported grid size (100x100)
- $xy2idx(), idx2xy()$ are encoding and decoding functions in use.

8.11.3. Pre-calculated data

Rebeca does not support ordinary trigonometric functions (like sin, cos, tan) while robotics uses them very often in geometric and kinematic computations. As angles in the model are discretized to integer values from 0 to 360° only, we can precompute all trigonometric measurements associated with each angle and put them into an array to look up later, using the value of the angle as the indexing key. The creation of such is done through the middle PHP script.

Rebeca also does not support file handling, so the conversion from a map file to an occupancy grid is done through the middle script and output result is saved to a text file and copied to the Rebeca code.

The strategy of using a helper script to help with developing and troubleshooting the model code is proven to be an effective approach.

8.12. Correctness properties

For a system of multiple mobile robots, we are interested in ensuring that:

- There is no collision between two robots in any circumstance (collision freedom)
- All robots finally travel to their expected destinations (reachability, deadlock freedom)

Currently Afra IDE only allows defining each property as a $boolean$ expression of rebecs' state variables to assert in each executed state transition. The assertions can be defined in a .property file or directly in the code through $assertion()$ command.

Properties for verification are defined as follows:

- Collision freedom: A "software" collision happens when two robots are too close to each other (distance between them is smaller than some value = distance to fit two robots next to each other = $ROBOT\_WIDTH$). Assertion of collision freedom is done inside the $msgsrv\ updateRobotLocation()$ in the map server node instead.
- Reachability: a tolerance is given for each robot, a state variable (target2distance) helps to track the distance to its destination. When the robot is near enough to target, reachability is achieved. This way is more flexible than requiring the robot to be on the target precisely, because it works that way in ROS2 as well, in many situations the robot just needs to get close enough to target.
- Deadlock freedom: a failure counter is used to track how many times a robot fails to reroute, if it exceeds some threshold, it is likely that the robot gets stuck. This property is asserted inside the code, on the rerouting part. It can be manually recognized when the state space explodes, the model checking process does not stop after a long time.

```
ros2rebeca-5b.property
property{
    define {
        onTarget1 = r1.distance2target <=r1.target_tolerance; //r1.rx==r1.targetX && r1.ry==r1.targetY;
        done1 = r1.moves[0]>0 && r1.moves[0]-r1.moveidx<=r1.target_tolerance;

        onTarget2 = r2.distance2target <=r2.target_tolerance; //r2.rx==r2.targetX && r2.ry==r2.targetY;
        done2 = r2.moves[0]>0 && r2.moves[0]-r2.moveidx<=r2.target_tolerance;

        onTarget3 =  r3.distance2target <=r3.target_tolerance; //r3.rx==r3.targetX && r3.ry==r3.targetY;
        done3 = r3.moves[0]>0 && r3.moves[0]-r3.moveidx<=r3.target_tolerance;

        onTarget4 =  r4.distance2target <=r4.target_tolerance; //r4.rx==r4.targetX && r4.ry==r4.targetY;
        done4 = r4.moves[0]>0 && r4.moves[0]-r4.moveidx<=r4.target_tolerance;

        onTarget5 =  r5.distance2target <=r5.target_tolerance; //r5.rx==r5.targetX && r5.ry==r5.targetY;
        done5 = r5.moves[0]>0 && r5.moves[0]-r5.moveidx<=r5.target_tolerance;
    }

    Assertion{
        reached_all : (!(done1 && onTarget1) || !(done2 && onTarget2) || !(done3 && onTarget3) || !(done4 && onTarget4) || !(done5 && onTarget5))
        ^ (done1 && onTarget1 && done2 && onTarget2 && done3 && onTarget3 && done4 && onTarget4 && done5 && onTarget5);
    }
}
```

Figure 29: Property definition file

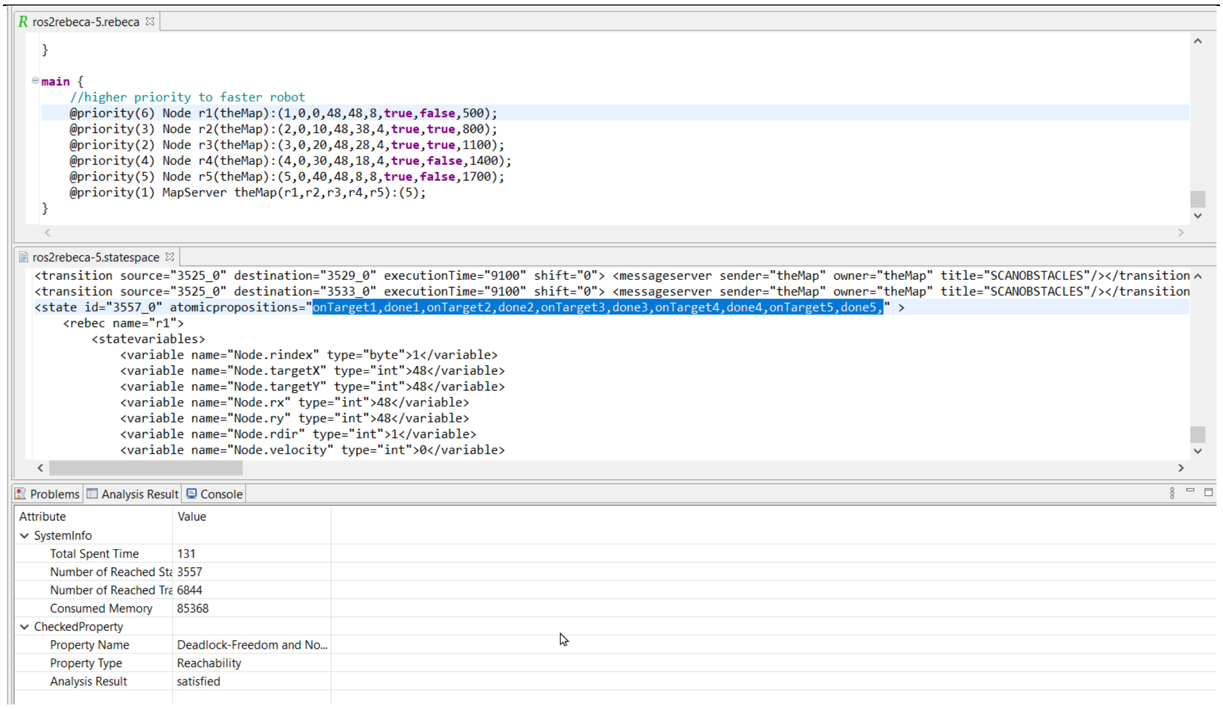


Figure 30: A working case of the first version of the model

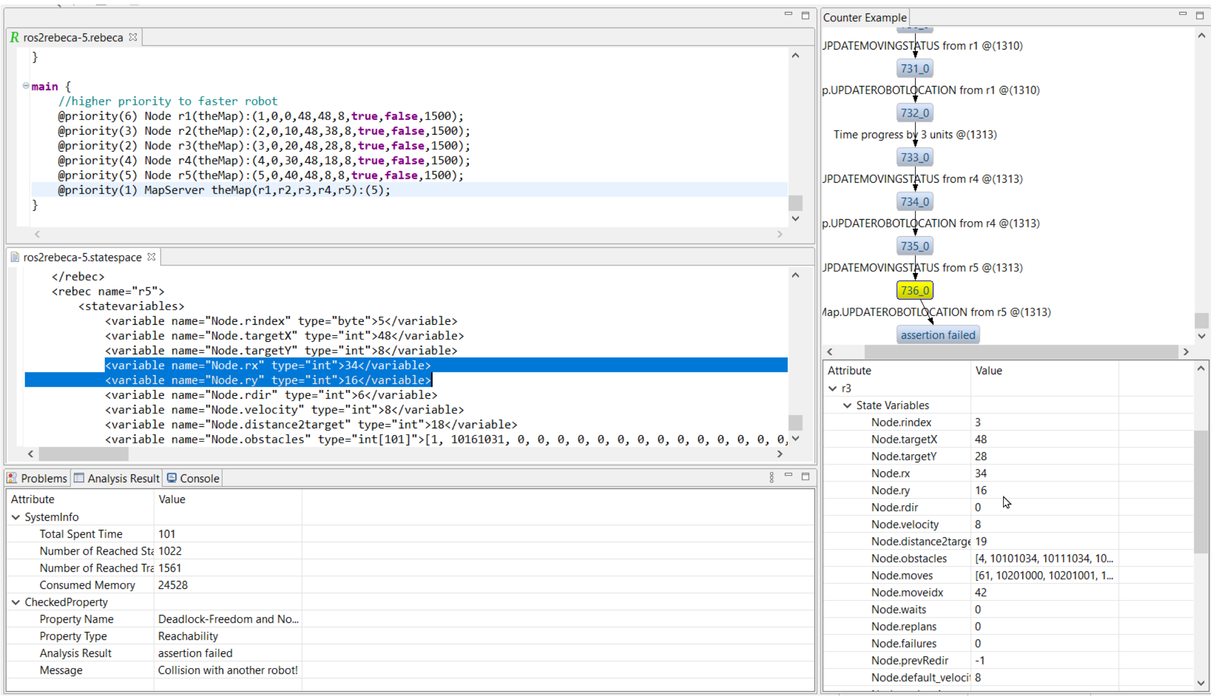


Figure 31: A counter-example in the first version of the model, where two robots $(r5, r3)$ collide.

9. Developing ROS2 code

Once a working model is achieved, I start to develop ROS2 simulation code to realize the model. It is actual model-based development in this phase. It is noteworthy that the created Rebeca model is a proof-of-concept version, it basically works but it is not perfect, not 100% correct yet. It is not yet precise enough to be a model for verification purpose, since verification requires 100% match between the model and the program. Later, once the program is in good shape, the model will be revised to match with the evolution in the program. In other words, the model and the code compensate each other in construction and evolution – the proof-of-concept model is used to help create the program code, and the evolved program code helps to recognize the flaws in the model, so that finally both the program and the model are corrected, improved, and synchronized.

The simulation code is developed using ROS2 Foxy version, Ubutu 20.04.5 LTS operating system installed on a VirtualBox virtual machine. Simulation is run on RViz2.

Multiple programming and definition languages are used to create the program:

- C++: for programming the nodes' code. Python can be more convenient but C++ provides better performance and is more used in industrial robotic applications.
- Python: for defining launch files.
- URDF and XACRO: for defining the robot's physical structure. The macro language (XACRO) allows to create parameterized template to generate different URDF files for different robots.
- XML: for defining the package list (package.xml)
- CMake build file: CMakeLists.txt

A ROS2 program is complex in structure with different kinds of information defined in different formats. It becomes even more complex when dealing with multiple robots instead of a single one.

9.1. Why RViz, not Gazebo?

There are two popular simulators in ROS2 which are used in parallel: Gazebo and Rviz (Rviz2 is version 2). Gazebo is for 3D visualization of the robots and the environment, while RViz is more for 2D visualization – showing the robots on a 2D map. Gazebo broadcasts **/laserscan** topic so that RViz can capture to visualize laser range detection.

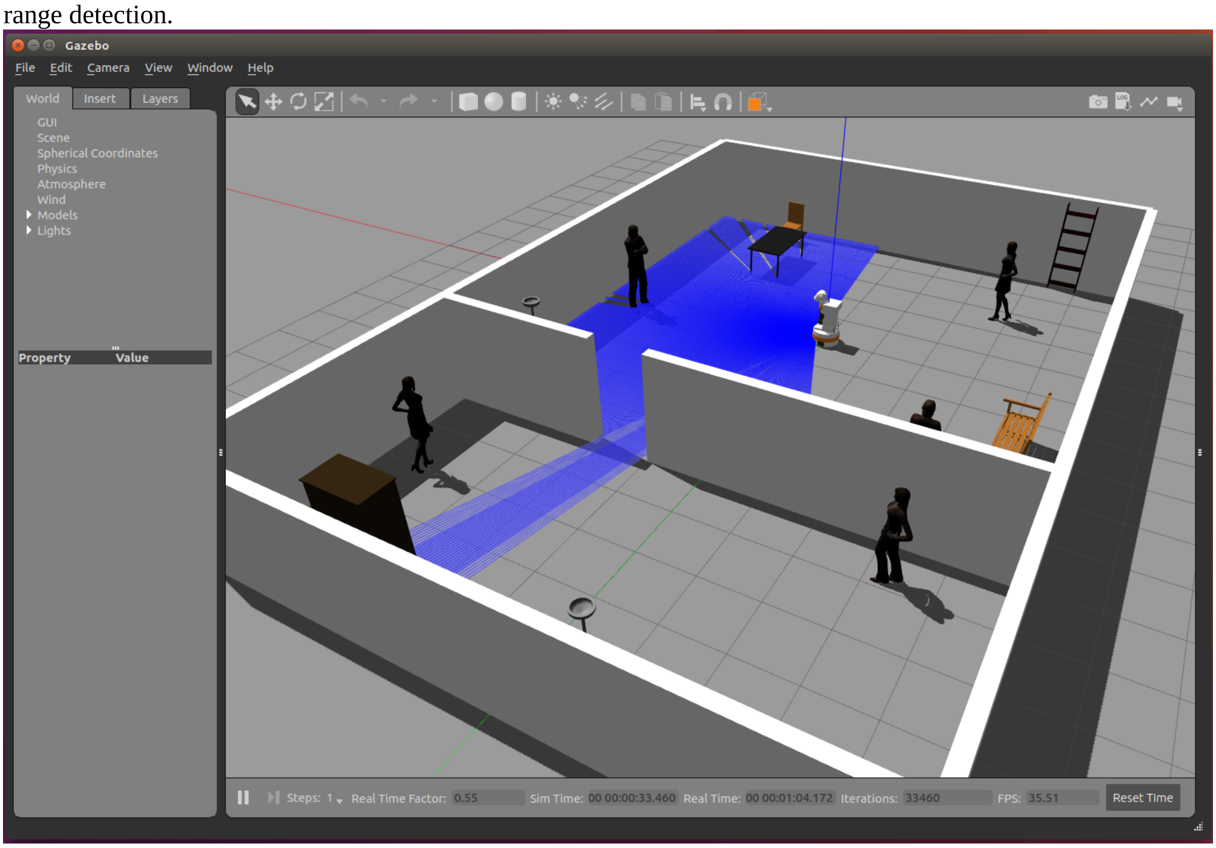


Figure 32: A Gazebo simulation (source: https://robmosys.eu/wiki-sn-02/baseline:scenarios:tiago_smartsoft)

RViz is a better fit for our purpose: showing mobile robots on an occupancy grid. The 3D visualization in Gazebo is eye-catching, making it look like a 3D game, but is excessive and heavy for our purpose – as we only need to care about the footprints (projected images) of the obstacles and the robots. RViz is also more light-weighted when it comes to visualizing multiple robots.

Without Gazebo, the laser sensing data has to be simulated by my own implementation inside the code. It is not easy but having everything integrated in RViz is a more self-contained, more compact solution.

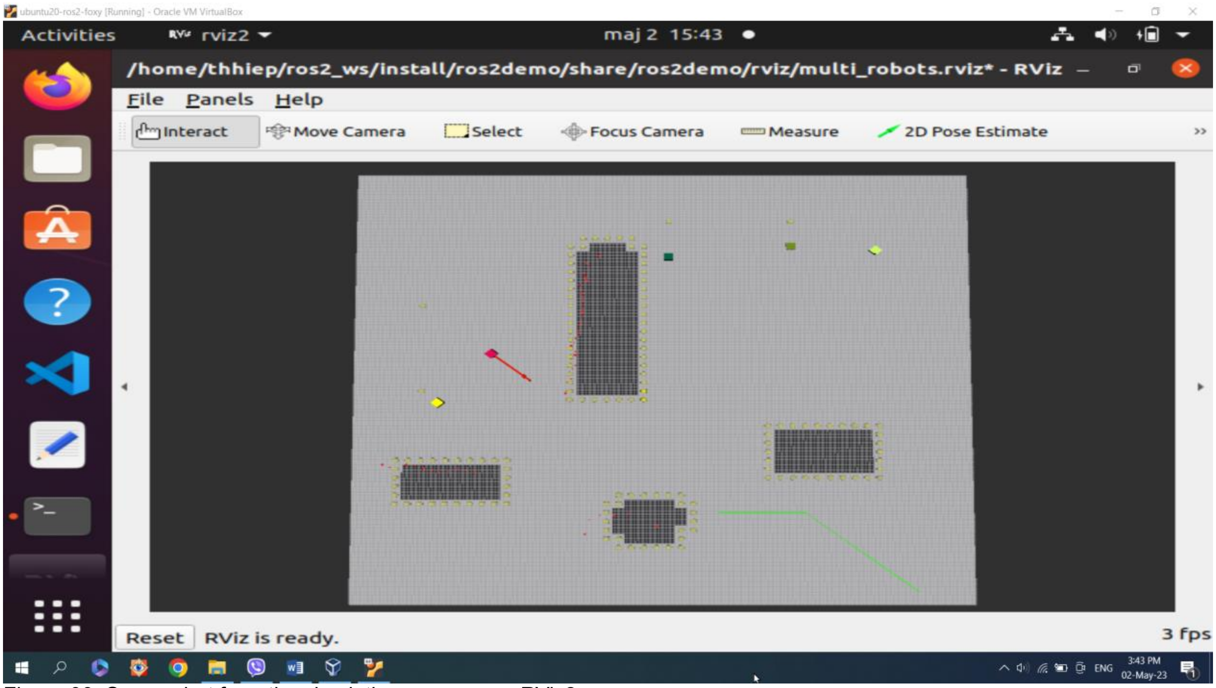


Figure 33: Screenshot from the simulation program on RViz2

9.2. Basic program structure

The ROS2 code implements two kinds of node as in the model: map server node and robot node. Basically, all communications in ROS2 are done through standard and custom pub-sub topics. RViz2 captures data in different topics to visualize on the graphical interface:

- Occupancy grid: the map server node needs to convert the PGM map to an occupancy grid and broadcasts this data frequently in a $nav\_msgs::msg::OccupancyGrid$ standard message.
- Robot description: the physical model of a robot defined in a URDF file is also broadcasted in a standard message by the built-in $robot\_state\_publisher$ package in a named channel for each robot ("$/r1/robot\_description$")
- Standard frame transformations: each robot has to broadcast standard transformations ($world \rightarrow map \rightarrow odometry \rightarrow base\_link$) so that RViz2 can visualize the robot dynamically while it is moving.
- Laserscan: simulated laser scan data for each robot is wrapped in $sensor\_msgs::msg::LaserScan$ messages and broadcasted through a named channel for each robot ("$/r1/laserscan$")
- Path: each robot sends a request to the map server through a service ("$/r1/generate\_path$") to ask for a path and get the generated path in response, the generated paths are also wrapped in $nav\_msgs::msg::Path$ messages and broadcasted through a named channel for each robot ("$/r1/path$") so that RViz2 can visualize the generated paths.
- Markers: additional visualizations to allow troubleshooting.

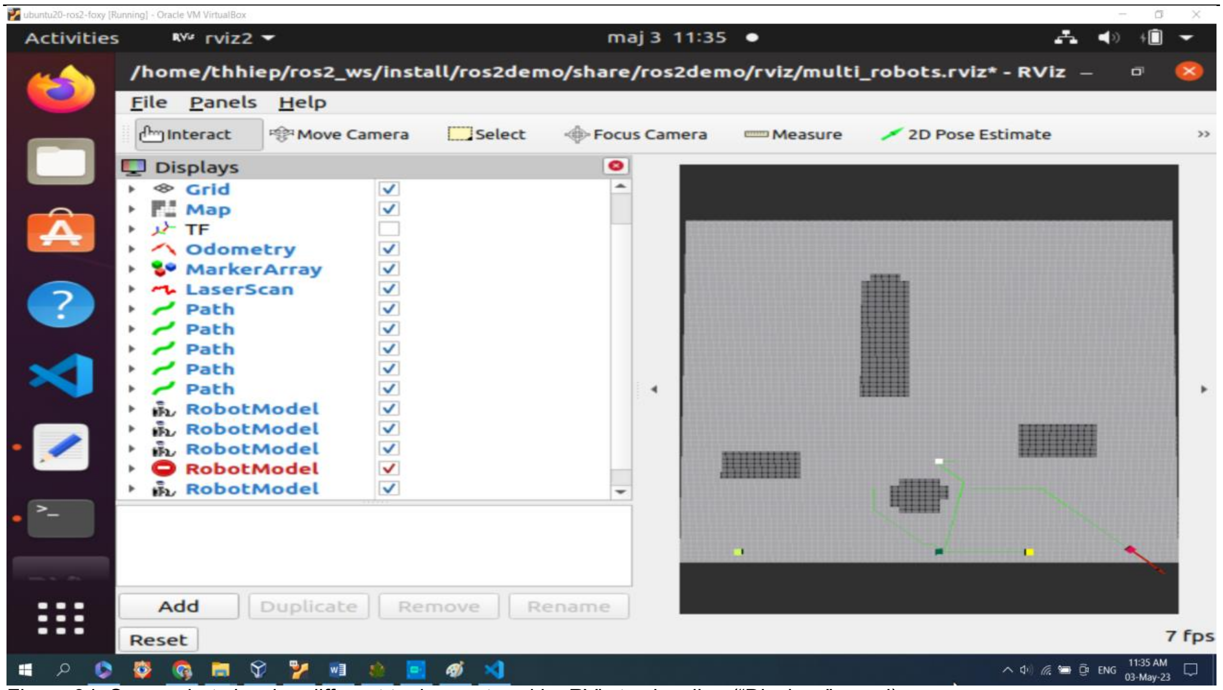


Figure 34: Screenshot showing different topics captured by RViz to visualize ("Displays" panel)

Folder and file structure:

TABLE 2: FOLDER AND FILE STRUCTURE OF THE ROS2 DEMO CODEBASE

Root folder	Subfolders	Files and purpose
ros2demo	/maps	.pgm and .yaml files for floor maps
	/launch	Launch profiles (in Python) to define startup configurations for the nodes
	/urdf	.xacro and .urdf files for defining robot's physical structure
	/rviz	Launch configuration for RViz2
	/src	Source code files for nodes (in C++): - mapservernode.cpp - robotnode.cpp
	/include/ros2demo	Common library files: - common.hpp
	CMakeLists.txt	A text file that contains build directives
	package.xml	An XML file that provides metadata and configuration information (dependencies) about the package

9.3. Multi robot handling

Most available ROS2 codebases only deal with one robot. To create a system of multiple robots, all names (topic names, frame names) allocated for each robot have to be made unique with a prefixing scheme:

- Each robot is assigned a unique ID ($r1, r2, r3$, etc.) – string name is used instead of pure number (Rebeca does not support key-value dictionary type, so only integer ID can be used in the model)
- Namespace: at spawning, a namespace is specified for each robot ($r1, r2, r3$, …), ROS2 then automatically adds the namespace ($/r1, /r2, /r3, \ldots$) as a prefix to the robot's topic names ($/r1/laserscan, /r1/odom, /r1/path$), service interfaces ($/r1/generate\_path$), and frames (frame names do not start with a dash, so "$r1/odom$", "$r1/base\_link$" instead). However, this automatic prefixing is not always ensured and consistent, it is better to double-check inside the code and make sure that the prefixing is done correctly and not doubled.
- URDF file: the XACRO template is filled with each robot's own parameters (name, dimensions, color, etc.) to generate an individual URDF model for each robot; so that the names of the links (and joints) come with the corresponding prefix ("$r1/base\_link$").
- Shared topics, services, frames (like $/map, /map\_updates, /register$) for all robots are not prefixed.

The map server node uses robot ID as key to store and look up information entries for each robot – which can be current pose, speed, communicators (topic publishers, subscribers, services) dedicated for each robot.

boxbot.xacro ●

D: > study > Master thesis > code > ros2demo > urdf > boxbot.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="box_bot">
    <xacro:property name="rname" value="$(arg rname)" />
    <xacro:property name="rlen" value="$(arg rlen)" />
    <xacro:property name="rwidth" value="$(arg rwidth)" />
    <material name="color_r1">
        <color rgba="0.9 0.04 0.34 1"/>
    </material>
    <material name="color_r2">
        <color rgba="1.0 1.0 0.0 1"/>
    </material>
    <material name="color_r3">
        <color rgba="0.02 0.39 0.28 1"/>
    </material>
    <material name="color_r4">
        <color rgba="0.45 0.55 0.14 1"/>
    </material>
    <material name="color_r5">
        <color rgba="0.81 0.98 0.4 1"/>
    </material>
  <link name="${rname}/base_link">
    <parent link="${rname}/odom"/>
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <box size="${rlen} ${rwidth} 0.1"/>
      </geometry>
      <material name="color_${rname}"/>
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <box size="${rlen} ${rwidth} 0.1"/>
      </geometry>
    </collision>
  </link>
</robot>
```

Figure 35: XACRO file as template for generating URDF model for each robot

9.4. Coordinate transformations

2D Cartesian coordinate systems are often positioned *left-handed*, that means the first corner (quadrant I, where x>0 and y>0) is on the left, the origin is at bottom-left corner of the window. However, RViz2 uses a *right-handed* system, that means the quadrant I is on the right, the origin is at the top-right corner of the window.

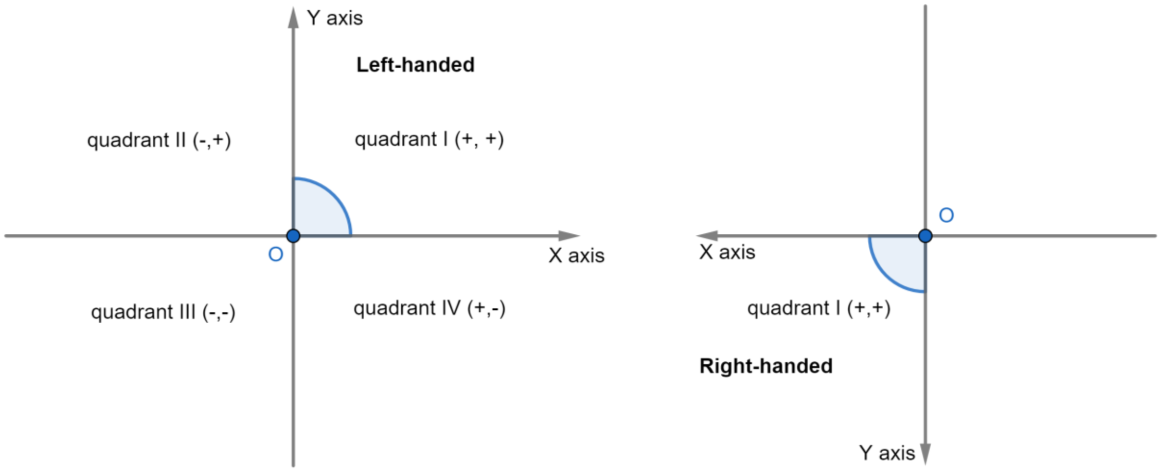


Figure 36: Left-handed vs. right-handed systems

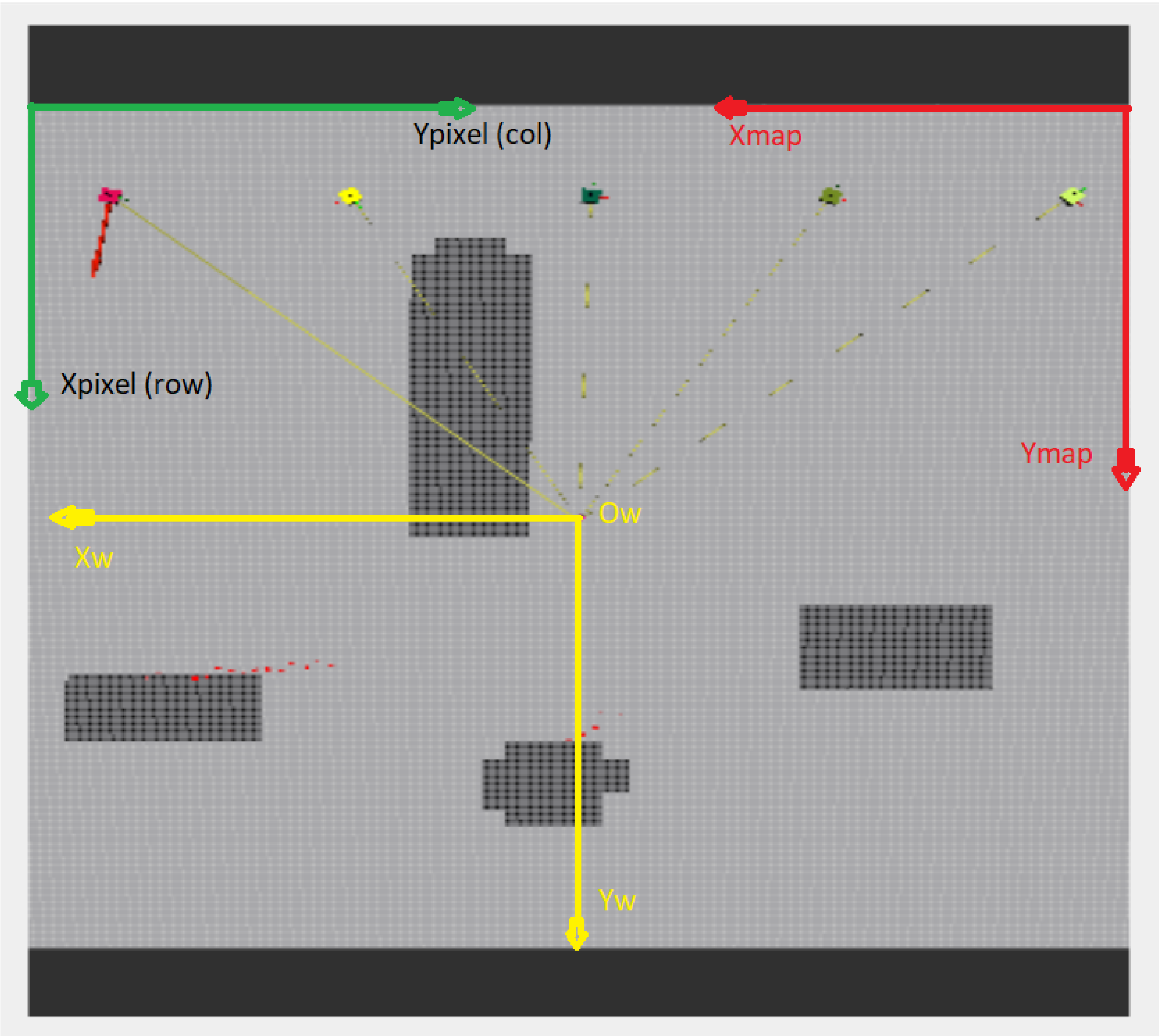


Figure 37: Relative poses of RViz2 coordinate frames (pixel, world, map/occupancy grid)

The positions of coordinate frames in ROS2 are often documented wrongly in many sources, which leads to wrong transformations and results in incorrect calculation and rendition of object positions.

There are four frames in use:

- World frame: fixed frame of real world position
- Map frame: relative position on the map (scaled down from the world frame by map resolution factor). The orientation of the map frame is the same as the world frame, but the map origin is at the top-right corner of the view port, while the world origin is at the center of the view port.
- Occupancy grid: same as map frame, but the coordinates are integer indexes of cells.
- Pixel frame: image coordinate (i, j) where i is row index, and j is column index. The grid frame is achieved by shifting the pixel frame to the right edge, and rotate -90 degrees.

Understand the relative positions of the frames and using the coordinate transformation, as explained in the background sector, will help achieving the right coordinate transformation among frames.

During the revision of the model-program synchronization, I have detected the bug of using wrong rotation matrix in the first version of the model and in the code and got it fixed. This happened due to wrong knowledge documented in many sources (including the work by Kostiantyn).

9.5. Node graph & interfaces

The table below lists communication interfaces (topics or services) among the nodes (map server, robots, RViz). R1 is a representative of any robot, other robots (r2, r3, …) will have the similar dedicated channels, prefixed by their name.

TABLE 3: TOPICS AND SERVICES IN THE ROS2 SIMULATION PROGRAM

Topic / Service	**Publisher / Server**	**Subscribers / Clients**	**Explain**
/map	Map server	All robots, RViz	Occupancy grid published periodically by the map server. RViz captures to display the map, robot nodes captures to get the occupancy grid.
/register	All robots	Map server	Each robot publishes its name to this channel periodically (e.g. every 10s) to register itself with the map server. This is handshaking process, so that the map server knows the ID of the robot and creates individual communication interfaces with it. This way, the map server can handle any number of robots.
/expanded_obstacles	Map server	RViz	The map server publishes markers of expanded occupancy cells, RViz captures to visualize (yellow dots).
/r1/laserscan	Map server	Robot r1, RViz	Simulated laser scan broadcasted by the map server. RViz captures to visualize (the red dots), robot r1 captures to detect nearby obstacles.
/r1/generate_path	Map server	Robot r1	Service interface provided by the map server so that robot r1 can request a path and trigger a callback when it is available.
/r1/path	Map server	RViz	When the map server finishes generating a path for a robot, it also publishes to this channel so that RViz can capture and visualize the generated path.
/r1/odom	Robot r1	Map server, RViz	R1 publishes its odometry (message and transformation frames) through this channel. RViz captures to visualize the movement vector. Map server captures to update the robot’s location and speeds.

/r1/obstacles	Robot r1	RViz	Robot r1 calculates blocking obstacles in its direction and publishes this data so that RViz can capture and visualize (light green dots).
/r1/robot_description	robot_state_publisher	RViz	ROS2 package reads the URDF model of the robot and publishes to this channel so that RViz can capture and visualize the shape of the robot.

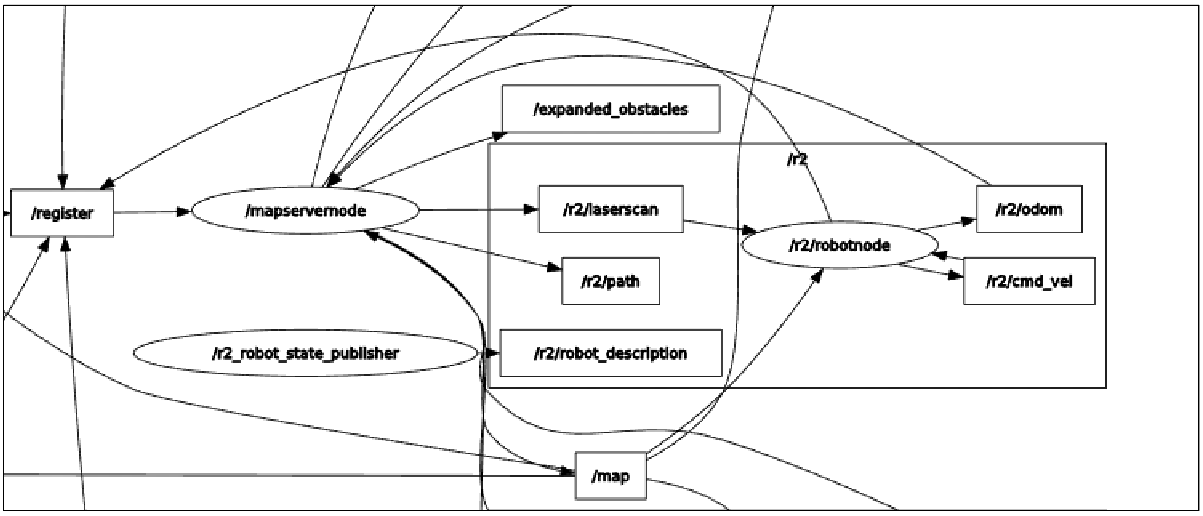


Figure 38: Extract from the node graph generated by the tool rqt_graph

9.6. Program-model synchronization

When going from a discrete model to continuous, incremental happenings in the program, adjustments must be made to smooth and optimize the robot's movements:

- Combine waypoints: in the generated path, waypoints in the same direction should be combined so that only starting and ending points are considered, middle points are skipped, which allows the robot to move faster and reduces the computation amount.
- Combine zig-zag points: zig-zag points in the middle of a diagonal should be skipped so that the robot can move more straightly, diagonally, and thus much faster than going through zig-zag points.

Other than that, basically all the design logics in the prototype model (node architecture, interaction flow, obstacle detection, collision avoidance, path planning & replanning, movement strategy) work well when ported to the program. This has proven the overall correctness of the prototype model.

Now when the program has evolved, the model needs to be revised to correct bugs and match with the evolutions in the program. To be usable for verification, the model must faithfully reflect what the program does. The prototype model is evolved to version 2 - the verification model - which is more correct and better in all aspects. The overall evolution cycles are illustrated in the flow chart below:

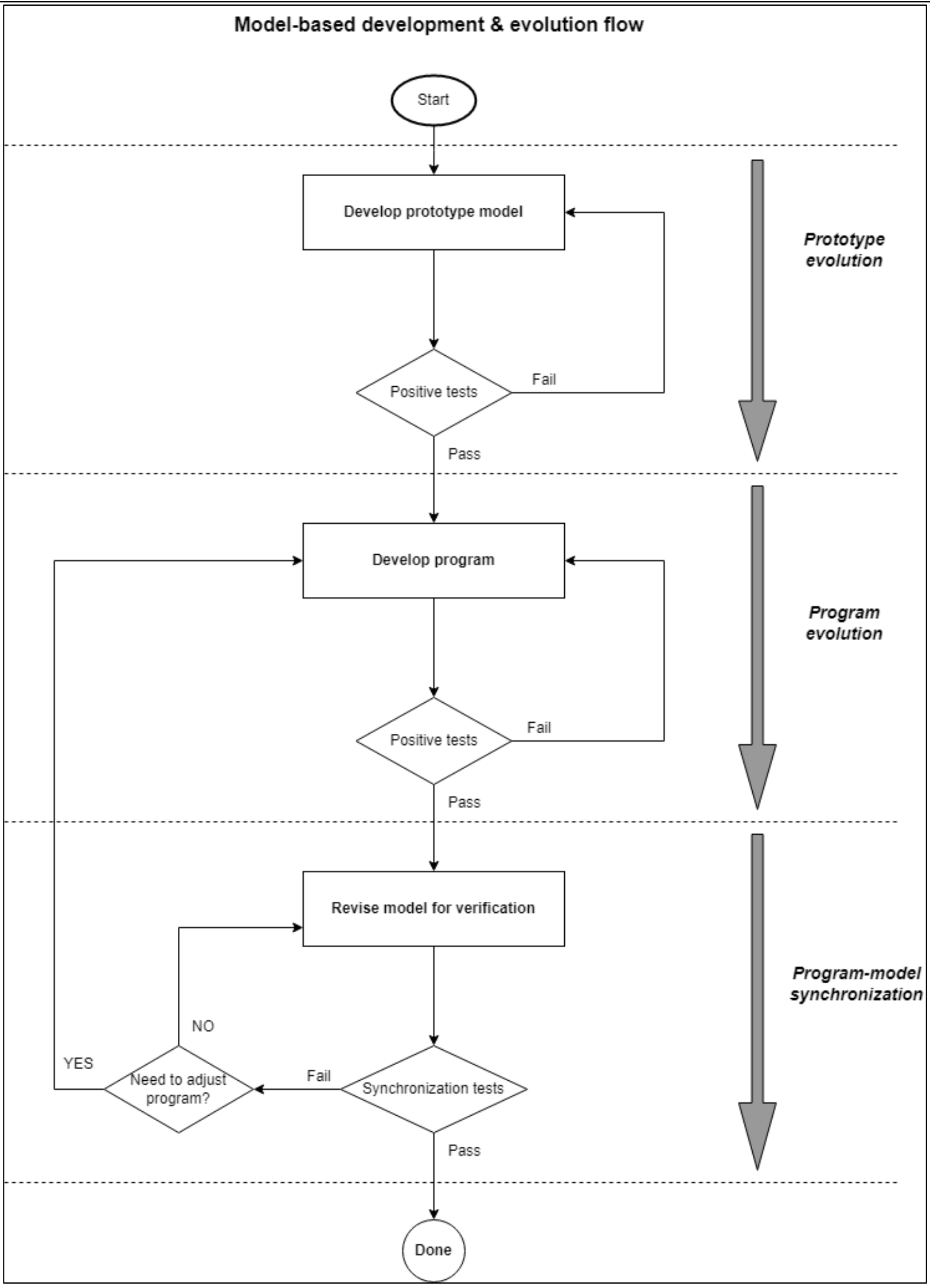


Figure 39: Model-based development & evolution flow

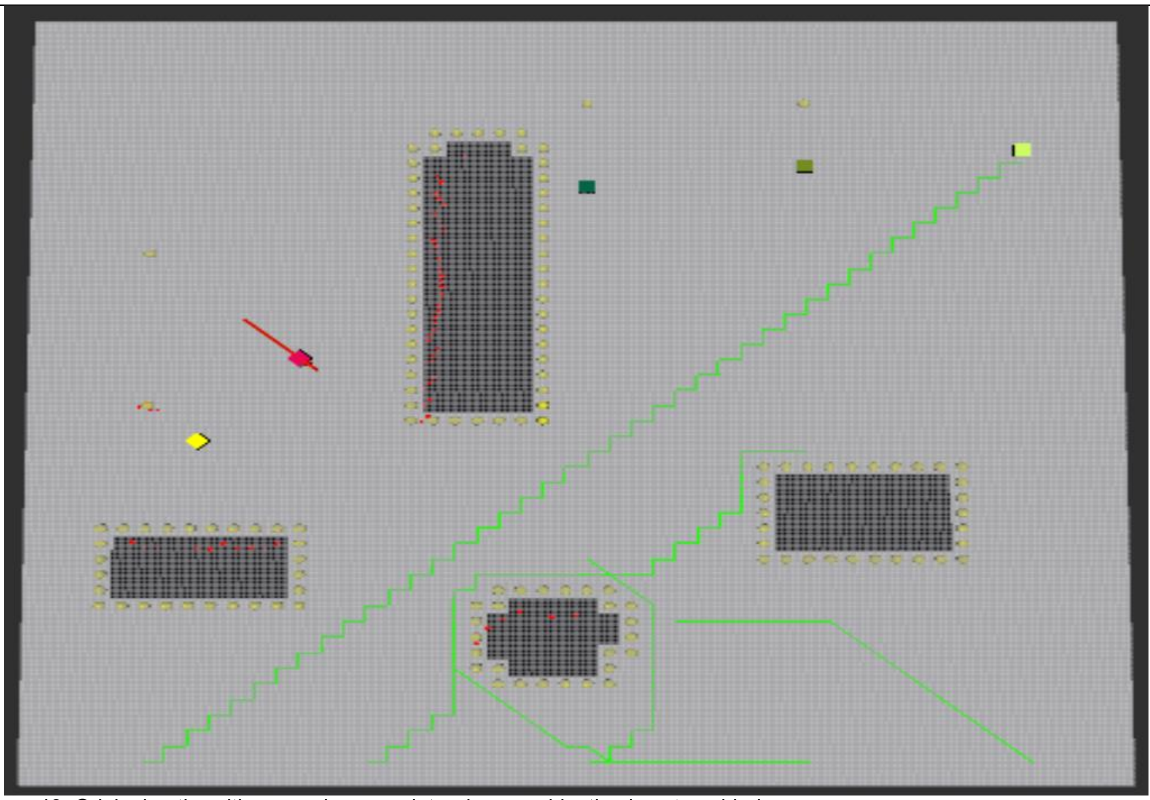


Figure 40: Original paths with many zig-zag points when combination is not enabled.

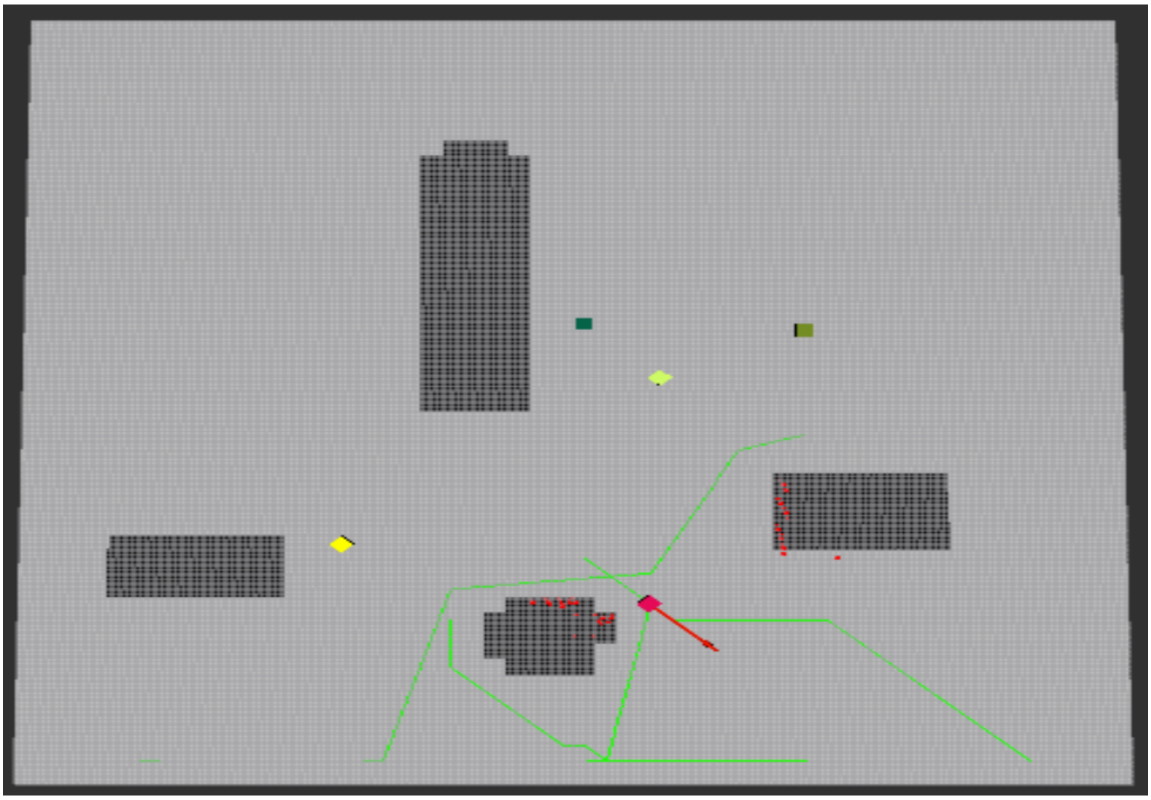


Figure 41: Straighter & shorter paths when zig-zag combination is enabled (not all paths visible due to RViz)

10. Results

To prove the success of the modelling, model-based development & verification, two cases are presented to show the synchronization between the model and the program, and how the model helps to reveal system errors:

- Working case: all robots reach their destinations without collision. There should be no problem in either the model or the program.
- Non-working case: one of the run-time properties is broken (a collision or deadlock happens). The problem detected in the model should also occur in the program.

The model and the program are set up to work on the same configuration:

- The map (PGM file, resolution)
- Laser scan rate, field of view, angle step, maximum range
- Robot dimensions (width, length), other safety distances (safe margin, stop zone)
- Robot working parameters (speed, maximum waiting time, movement mode – scan first or not)
- Path generation options (combine zig-zag)
- Starting point and target point for each robot (in pixel coordinate)

So under the same context, the model and the program should produce the same outcome. An extreme scenario is set up where 05 robots have to travel highly crossing paths to reach to their targets across the map. There is a high risk of collision and congestion to test the correctness and intelligence of the algorithm.

10.1. Working case

Environment parameters:

```
env int MAP_WIDTH=50;    //cols
env int MAP_HEIGHT=50;  //rows
env double MAP_RESOL=0.25500;   //cell size in reality in meters
env double CELL_WIDTH = MAP_RESOL;
env double CELL_DIAG = SQRT2*MAP_RESOL;
env int XMAX = MAP_HEIGHT-1;
env int YMAX = MAP_WIDTH-1;

env double ROBOT_WIDTH = 0.2; //in meters
env double ROBOT_LENGTH = 0.2;
env double SAFE_MARGIN = 0.02;
env double STOP_ZONE = 0.4;//bad value: 0.2

env int ANGLE_INC = 2; //angle increment in degrees, fixed value, do not change
env int FIELD_OF_VIEW = 180;         //field of view (LIDAR sensor)
env int LASERSCAN_RATE = 140;   //7Hz
env double LASERSCAN_MAX_DISTANCE = 10.0;   //= max range in meters

env boolean COMBINE_ZICZAC = true;
```

Figure 42: Screenshots of a working case in the model

Robot configuration:

```
main {
    //good config
    @priority(5) Node r1(theMap):(1,15,5,48,45,2,1.5,1500,false);
    @priority(4) Node r2(theMap):(2,25,5,48,35,2,0.7,1500,false);
    @priority(3) Node r3(theMap):(3,5,25,48,25,2,0.7,1500,false);
    @priority(2) Node r4(theMap):(4,5,35,48,15,2,0.8,1500,false);
    @priority(6) Node r5(theMap):(5,5,45,48,5,2,1.2,1500,false);
    @priority(1) MapServer theMap(r1,r2,r3,r4,r5):(5);
```

Figure 43: Screenshots of a working case in the model

Parameters are explained as below: e.g. `Node r1(theMap):(1,15,5,48,45,2,1.5,1500,false);`

- 1 = robot ID
- (15,5) = starting point (in pixel coordinate)

- (48,45) = target point
- 2 = tolerance to target
- 1.5 = speed (m/s)
- 1500 = maximum waiting time (in milliseconds)
- False = scan first mode, so it moves in parallel with scanning.

The robots are configured to be heterogeneous in some way (different speeds). The model checking shows that everything went well, all robots reached their targets without collision.

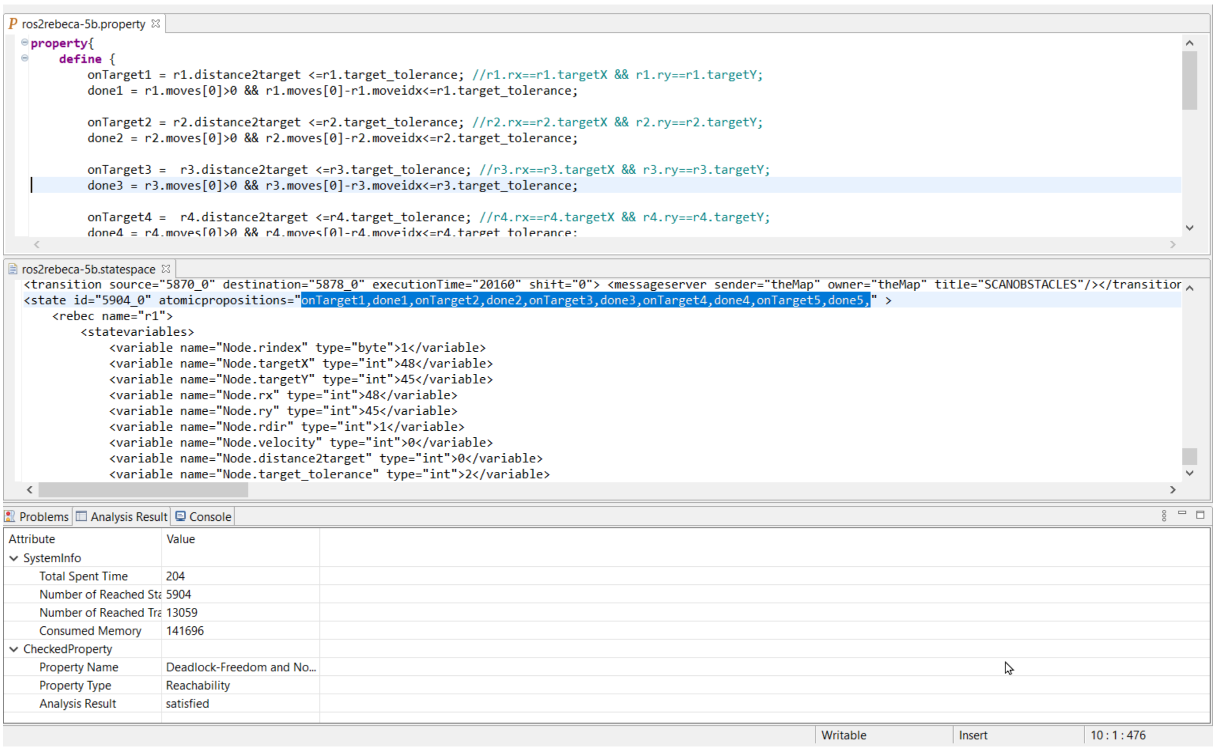


Figure 44: Screenshots of a working case in the model

Next, the program is set up to run with the same configuration:

```python
map_name = 'map50';

map_pgm = pkg_dir+'/maps/'+map_name+'.pgm'
assert os.path.isfile(map_pgm), 'Map file not found at '+map_pgm
map_yaml = pkg_dir+'/maps/'+map_name+'.yaml'
assert os.path.isfile(map_yaml), 'Map file not found at '+map_yaml
with open(map_yaml,'r') as stream:
  map_cfg = yaml.safe_load(stream);
#print(map_cfg);

def generate_launch_description():
    # Create the map server node
    map_server_node = Node(
        package='ros2demo',
        executable='mapservernode',
        output='screen',
        parameters=[{
            'map_pgm':map_pgm,
            'map_yaml':map_yaml,
            'map_resol':map_cfg['resolution'],
            'map_origin_x':map_cfg['origin'][0],
            'map_origin_y':map_cfg['origin'][1],
            'map_occupied_thresh':map_cfg['occupied_thresh'],
            'map_free_thresh':map_cfg['free_thresh'],
            'combine_ziczac':1,
            'laserscan_rate':140,
            'fov':180.0,
            'angle_inc':2.0,
            'max_range':10.0,
            'min_range':0.01,
            'robot_length': robot_length,
```

Figure 45: Screenshots of a working case in the program

```python
#measurements are in meters
robot_length = 0.2;
robot_width = 0.2;
safe_margin = 0.02;
stop_zone = 0.4;

#(x,y)=start point, (tx,ty)=target point, in pixel coordinates
robots = [
    {'x':15,'y':5,'tx':48,'ty':45,'velocity':1.5,'max_waiting_time':1500,'scan_first':0,},
    {'x':25,'y':5,'tx':48,'ty':35,'velocity':0.7,'max_waiting_time':1500,'scan_first':0,},
    {'x':5,'y':25,'tx':48,'ty':25,'velocity':0.7,'max_waiting_time':1500,'scan_first':0,},
    {'x':5,'y':35,'tx':48,'ty':15,'velocity':0.8,'max_waiting_time':1500,'scan_first':0,},
    {'x':5,'y':45,'tx':48,'ty':5,'velocity':1.2,'max_waiting_time':1500,'scan_first':0,},
];
```

Figure 46: Screenshots of a working case in the program

The program is built and executed with corresponding launch profiles and shows that all robots have travelled to their targets without collision, same as in the model.

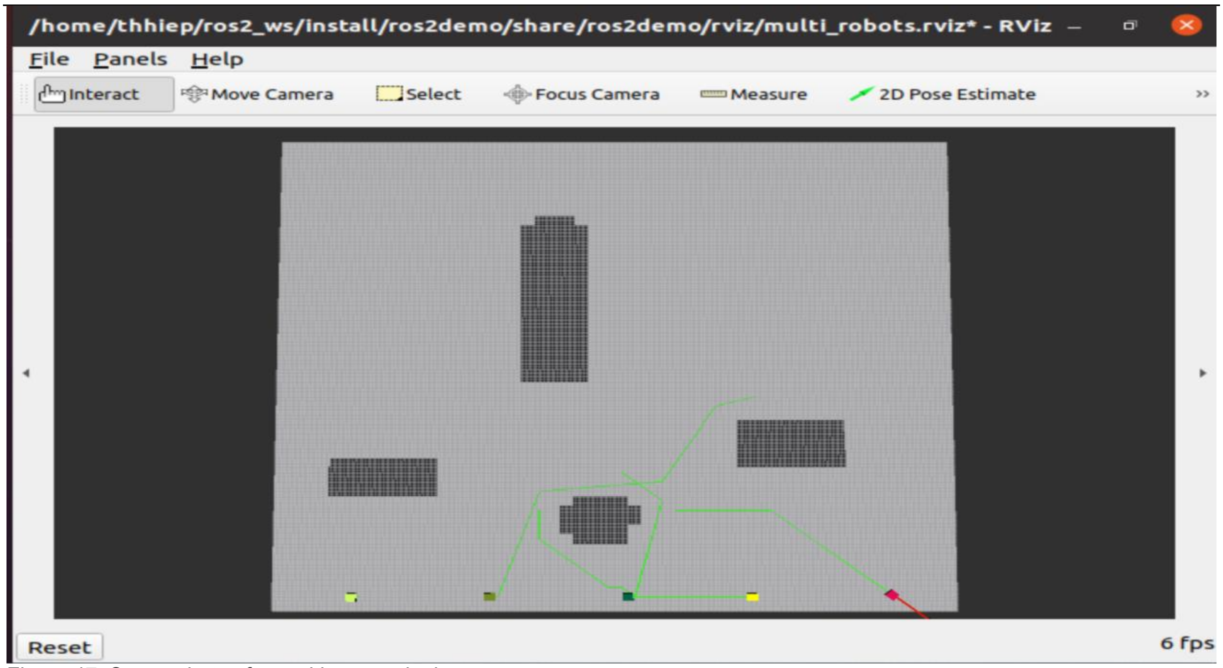


Figure 47: Screenshots of a working case in the program

10.2. Non-working case

A bad configuration is selected deliberately with unmatched parameters that lead to a high risk of collision:

- High speed compared to laser scan rate
- Short stop distance
- Same parameters for all robots

So the stop zone is set to $0.2\ meters$, while speeds are set at $1.6m/s$ (time to cross a cell is $1000 * 0.255/1.6 = 160ms$, laser scan rate should be $<= 160/2 = 80ms$ while it is set to $140ms$)

*ros2rebeca-5b.rebeca

```
env int MAP_WIDTH=50;    //cols
env int MAP_HEIGHT=50;  //rows
env double MAP_RESOL=0.25500;    //cell size in reality in meters
env double CELL_WIDTH = MAP_RESOL;
env double CELL_DIAG = SQRT2*MAP_RESOL;
env int XMAX = MAP_HEIGHT-1;
env int YMAX = MAP_WIDTH-1;

env double ROBOT_WIDTH = 0.2; //in meters
env double ROBOT_LENGTH = 0.2;
env double SAFE_MARGIN = 0.02;
env double STOP_ZONE = 0.2;//bad value: 0.2

env int ANGLE_INC = 2; //angle increment in degrees, fixed value, do not change
env int FIELD_OF_VIEW = 180;        //field of view (LIDAR sensor)
env int LASERSCAN_RATE = 140;    //7Hz
env double LASERSCAN_MAX_DISTANCE = 10.0;    //= max range in meters

env boolean COMBINE_ZICZAC = true;
```

Figure 48: Screenshots of a non-working case in the model

The model checking fails, a collision between two robots has been detected:

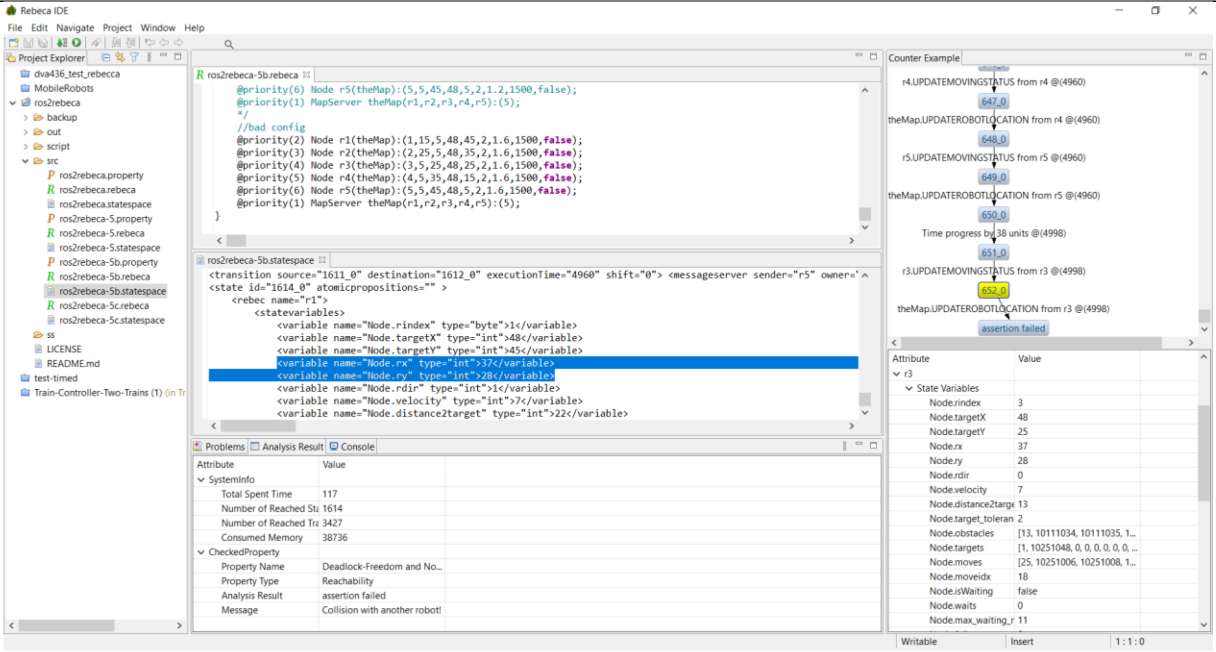


Figure 49: Screenshots of a non-working case in the model

The same settings are set in the program, and after a few runs with flaky test results, a collision happens as well.

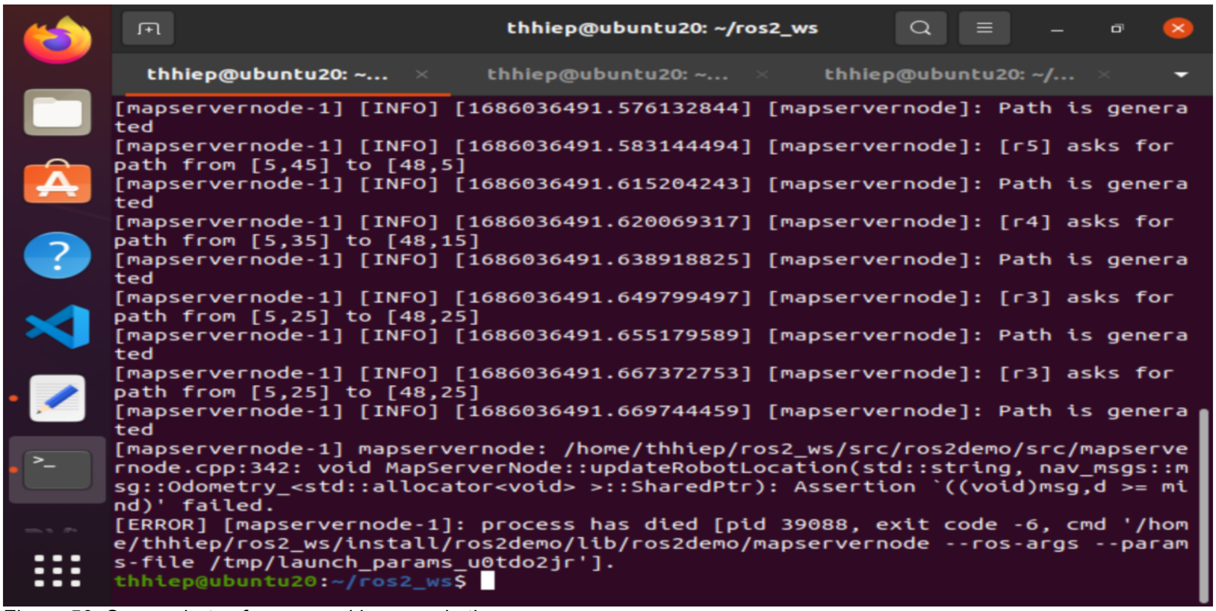


Figure 50: Screenshots of a non-working case in the program

Figure 51: Screenshots of a non-working case in the program

This non-working case is selected deliberately to be a marginal case, where the simulation behaves inconsistently – sometimes the robots passed without collision, sometimes they collided. But in the end, safety is not guaranteed as the risk is exposed in the model. The flaky behaviors of the simulation happen because it is suffered from lags due to various visualization tasks, while the model checker behaves consistently – if it fails, it will always fail.

11. Discussion

11.1. Modelling & development results

By following a systematic & new approach, the project has successfully delivered two Rebeca models (two versions) to model and verify a realistic and complicated ROS2 robotic problem – multiple autonomous mobile robots:

- A prototype model (version 1) which is used as a base for the development of corresponding ROS2 program code.
- A verification model (version 2) which is an upgrade of version 1, revised to synchronize with the program, tested to prove its validity and used to automate the verification process using model checking technique.

The two versions of the model reflect the evolution process. We can find that:

- The prototype model: this is the first attempt to model a complicated system, created on top of pure analysis & design ideas, it works somehow but still contains bugs and is not 100% correct yet. However, we do not need to wait for it to mature to start developing the program code, because the model cannot correct itself without realization.
- The verification model: once created, the program code evolves independently to smooth real robotic behaviours. The model then has to be revised to match with the evolution in the final program. Synchronization is the prerequisite for it to be used for verification process (What is the point in having a perfectly working model while it turns out to be a different thing from the real system?). Synchronization tests have proven the faithfulness of the revised model.

The achievements of this project are far more advanced than previous works by (Saeid, 2019) & (Kostiantyn, 2020) by:

- re-addressing the problem from a new & systematic perspective,
- looking closely at mathematical grounds of robotic problems,
- clearly identifying and strategically dealing with the continuous-discrete gap between the real world and the model (simplification, discretization),
- using optimization techniques to overcome the limitations of Rebeca (helper script, pre-calculated data, etc.).

It successfully deals with real industrial-level robotic problems, instead of toy problems, taking equivalent consideration of all components of a ROS2 multiple mobile robots application:

- Mapping
- Sensor-based obstacle detection
- Sophisticated path planning algorithm
- Robot physical dimensions
- Robot movement characteristics (rotation & linear movement, speeds, stopping distance)
- Human-like collision avoidance strategy (identifying blocking and non-blocking obstacles, waiting, backing off, resolving congestion, rerouting, …)

The modelling principles and practices in this project can be reused for many other cases, not limited to mobile robot problems.

11.2. Answers to research questions

- **RQ1: What robotic problems can be modelled in Timed Rebeca and how to model them?**

The similarity in asynchronous, concurrent lifecycle of ROS2 nodes and Rebeca reactive objects basically allow to model ROS2 node graph using Rebeca reactive objects and their message server methods. The project has demonstrated a very precise and smooth model of multiple autonomous mobile robots system, which can be used as a template for modelling other applications.

- **RQ2: Which correctness properties of ROS2 robotic applications are suitable to verify with Rebeca model checking?**

In the demonstrated model, following properties are verified by the model checker: goal reachability, freedom of deadlocks, freedom of collision. A correctness property is a logical expression of state variables, it can only be verified throughout all possible state transitions if their changes are tracked completely, without missing any moment or any part. Therefore, it is not true to state that "*there is no limitation on defining desirable properties as long as the property can be formulated in TTL / Assertion format used by Rebeca Model Checker (RMC)*" [Saeid, 2019]. If the model misses to capture the flows of involved state variables, then the verification of the property is incomplete. As stated, a model is a simplified representation of a real system for certain purposes, multiple models may need to be created for verifying different aspects of a robotic system. The other properties that can be further checked are (but not limited to):

- Other timing and sequence requirements: an action must be achieved within some period of time, some states/actions/responses must or must not happen in some order.
- Synchronization problems: as ROS2 nodes react in an asynchronous, parallel, concurrent manner; there are possible problems of synchronization that can be efficiently detected by model checking.

To stay on the safe side, we always need to create a working model and test it to have an evidence-based conclusion of which properties it can verify.

- **RQ3: Can the Rebeca models be generated from ROS2 code?**

Initially, the expectation was to generate Rebeca model from ROS2 code so that it can help with the automatic verification process. However, this expectation is based on an implicit hypothesis that the generation can be automatized. It does not work when there is no code at all, in prototyping phase. As mentioned, the term "code" misleads to programming language files only, while a ROS2 application contains multiple information resources in different formats. By using the working model as a template, I have used the helper script to generate application-specific data (e.g. the map, application configurations) and insert it into the template to create the model. That way, the generation is partially automatized.

Figure 52: Overall approach in generating Rebeca model from ROS2 application

Due to many programming inequivalences between an implementation language in ROS2 (C++, Python) and current supported features in Rebeca, there are many parts that have to be manually bridged. The same processing logic is implemented differently in ROS2 and Rebeca, for example, the same behaviour $\{move\ robot\ to\ (x, y)\}$ is done differently because the movement is continuous in the real application and abrupt in the model. And in ROS2 code, not everything is defined in the code, any reference to an external library function has to be reproduced in the model, for example, if a path is generated by an external module, then the model has to model that external function as well.

Therefore, the automatic generation of any model from an application code (if there is) is not done straightly. The recommended approach is to create template models for specific purposes and use a helper script to generate an instance for each application context, like depicted in the above flow.

The full automatic generation should be addressed in a different project, based on the result of this project. This question is partially and indirectly answered by the findings of this project. One question should be asked before rolling out any attempt to do such is "Is it worth the effort? Why do we have to do it?".

- **RQ4: What are the achievements and unresolved challenges?**

The achievements have been summed up in the above section. This work is a ground-breaking type, trying to discover many unknowns and to correct prior misunderstandings. The values it contributes are not just a final model and program code, but the many knowledge during the process. Since robotics is a wide and complex interdisciplinary field, it cannot address and solve everything within the scope of a thesis project. The project only handles one common and complex enough context: multiple mobile robots. There are many other robotic problems that need further studies:

- Robotic arms: the movement of a robotic arm is modelled differently from a mobile robot. They have joints, links, an end-effector; thus have many degrees of freedom.
- Modelling and verification of timing, sequence, concurrency properties: these matters have to be investigated separately, a different model may need to be created for each.

11.3. Open matters

There are open matters that have been experienced and observed during the project. I document it here so that other researchers can look at:

- **Pitfall in discretization**

As the continuous behaviour in the real system is discretized in the model, there is always some unfilled gap. For example, when a robot moves from point $(1,1)$ to point $(3,1)$, in reality it moves gradually from $1, 1.1, 1.2, ...$ until 3 and the events happening in this process can be captured by high frequency scanning. In the model, it happens abruptly with some timing representation only:

$$robot.location = (1,1);\ delay(travel\_time);\ robot.location = (3,1);$$

That way if some event happens during that $delay()$, it may be missed. If we chop down the process into smaller discrete steps, e.g. having an extra integer variable to track the fraction of the steps, we have more chance of not missing in-the-middle event but we can end up in a too detailed model with an unmanageable explosion of the state space. Do we need to do that? What are the loss and the gain? This matter has to be further investigated.

- **Instability of ROS2**

During the development of the ROS2 program, I experienced memory management issues that caused the map server node to terminate abruptly. The error could not be traced in the application code, despite different techniques used. It is highly due to hidden synchronization bugs in ROS2 which can only be exposed in a heavy traffic system (5 robots with multiple concurrent, high frequency writings and readings in the shared map server node). This matter also requires deeper investigation and community effort to verify in different contexts.

Figure 53: A screenshot showing possible ROS2 memory management bugs.

- **Validity of model checking result**

In theory, if the model checking process verifies that a correctness property is not broken in any possible state transition, the system is proven to be free from that fault. However, there is always a gap between the model and the real system, how to guarantee 100% that the model models exactly what is implemented? It is referred to as model validation, model-program synchronization. Currently synchronization is only verified by trying some scenarios and observed that the model and the program produce the same outcome. How to exhaustively, systematically verify this "last link" synchronization? It is an open matter.

Due to this gap, to stay on the safe side, when there is no problem detected in the model, we can say that we have more confidence in building the real system, there is a much smaller probability of such error in the real system. It is not safe to say that because the model works perfectly well, there will be no problem in the real thing.

11.4. Modelling lessons

Along the way of discovery and coming to a working model for such complex robotic problems, I learned a lot of valuable modelling knowledge and practices which are not just narrowly applicable to ROS2 and Rebeca.

11.4.1. Characteristics of a model

A model is a simplified representation of a real world object, system, phenomenon for certain purposes. We have to create a simplified, miniature version of a real world system which already exists or is to be constructed because the real system is too much complicated to study or test. A model is a slice, an extracted view of a real system which composes of many layers of information.

Modelling has been used in other fields long before software engineering, like in construction and geography. To build a bridge, construction engineers have to create different models of it so that they can visualize, test their design before building the actual bridge. They have to model because they cannot build a real bridge to test, it is a huge work. They use models to test and improve their design to gain more confidence before building the actual one. It is too much expensive or impossible to "debug", fix or scrap and rebuild the actual bridge. They may create different models for different purposes:

- A paper model to visualize the shape, the outside appearance of the bridge
- A wooden model to test the structure of the bridge against weight and vibrations
- A 3D software simulation of the bridge with more complicated information layers

In geography, for example to study the Earth, geographers created different models of the Earth:

- A globe to visualize the shape, the axis, the continents, oceans and lands on Earth
- Different kinds of maps with different levels of details for different purposes

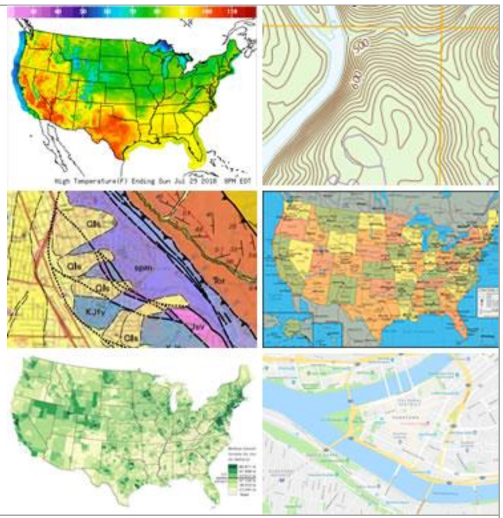
		
The real Earth	A globe	Different types of map

Figure 54: The Earth and different models for studying it (source: Google Images)

We can see the following necessary characteristics of a model:

- Simplification: modelling is to keep necessary information to serve the interested view only and drop other information that is out of concern. I term it as "*information retain*" and "*information drop*". If we do not see simplification but only 1-on-1 imitation of reality, it is not modelling.
- Purpose: modelling always comes with a clear, narrow purpose in mind. For different purposes we may have to create different models of the same system. As said, there is no for-all-purposes model, the only one that contains everything is the actual system.

11.4.2. Simulating vs. modelling

Robotics involves a lot geometry, kinematics (motion), dynamics; hence naturally there comes a great desire in visualizing the structure, movements and interactions of the robots and their operational environment. Seeing is believing, apparently. What we see more often in robotics is simulating, not the type of modelling for model checking. It is easy to mistake between the two. Simulating catches human eyes and seems to be more convincing, but it does not provide the theoretical verification as model checking can. In model checking, an operationalizable mathematical model of the system is created to facilitate exhaustive traversing of the state space to prove that the expected properties hold in all circumstances – what simulation cannot do automatically.

Since modelling robotic problems is still not established, it is easy to do simulating instead of modelling, which results in overly-detailed models. On the other hand, due to the complexity of robotic problems, it is equally easy to overly simplify the model, resulting in too much information drop and the created model starts to be an unfaithful, even incorrect reflection of the real system. These two extremes in modelling should be both avoided.

There is a subtle line between simulating and modelling that the modelling engineer has to decide on their own, depending on the modelling purpose and the capability of the model checker in use. It is also the line between "enough" and "too much" of information drop and information retain.

Here are some differences between simulating and modelling up to our observations:

- <u>Detailed level:</u> in terms of detailed level, reality > simulating > modelling. Simulation tries to retain more information to imitate reality, while modelling tries to drop more to keep only information of interest. In other words, for simulation, "more is better" while, for modelling, "less is better". We can make an analogy as follows: to reproduce a curve as smoothly as possible, simulation tries to capture more sampling points, while modelling captures only turning points where the changes happen.

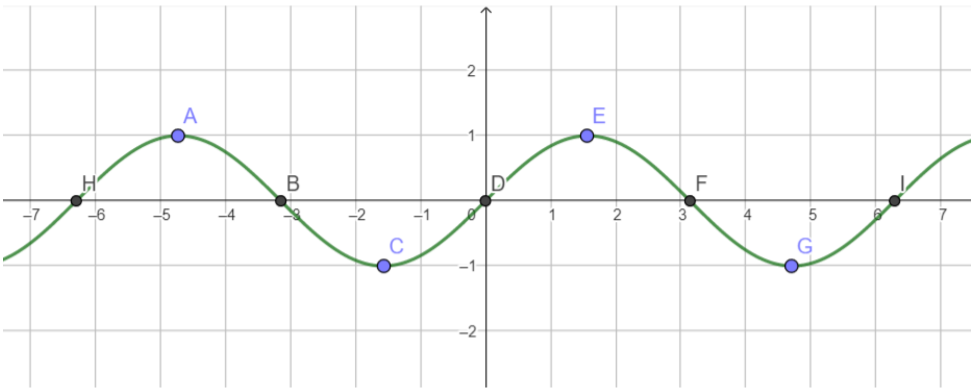


Figure 55: Turning points in a curve

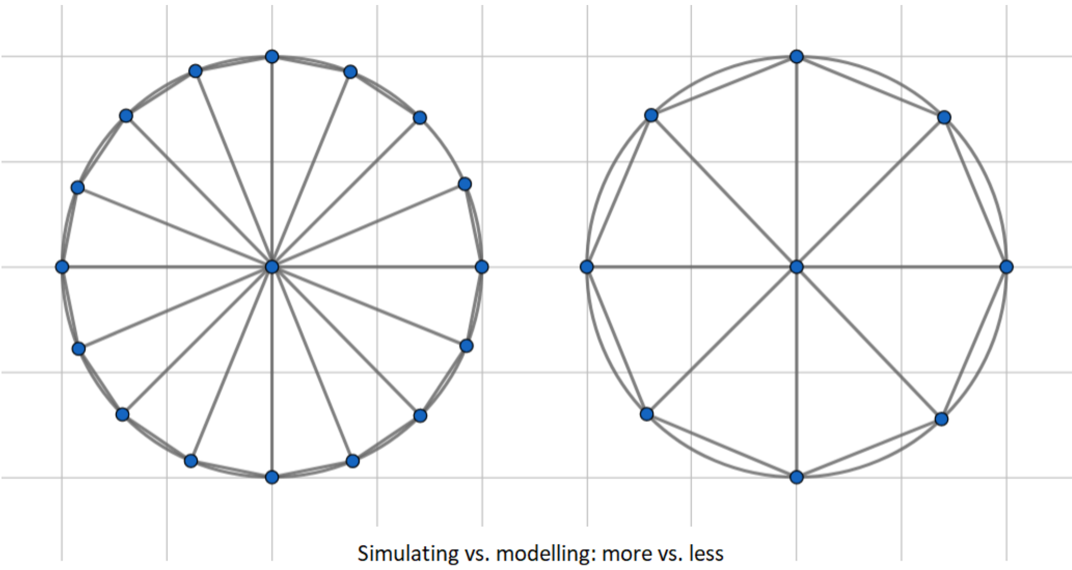


Figure 56: Simulation vs. modelling

- <u>Visualization:</u> simulation is more about visualizing. Since model checking works invisibly, troubleshooting its correctness can be difficult without visualization, especially with the limited debugging capability in Rebeca IDE. To aid modelling job, we have used a middle tool to bridge the gap, which helps to prepare, visualize and troubleshoot data, methods used in the model.

11.4.3. Physical model vs. software model

The hard part of software modelling is that it is not bounded physically, visibly. With software modelling, a value can be set arbitrarily just by an assignment command, so it can get out of range easily, while with physical modelling it cannot be done because of physical constraints. Let's make an example for easier understanding:

- In real world, a bird inside a cage cannot get out easily because it is bounded by the cage.
- In a software model, the "software bird" can be set free easily out of the "software cage" just by setting the bird's (x, y) coordinate outside the cage.

Figure 57: Bird in cage (source: Google Images)

Similarly, in modelling robotic behaviors, a software robot can easily be set to a location where it cannot be because it is occupied by an obstacle. In reality, the robot just has to stop because of the physical collision, while a software collision happens or not just depending on the way it is defined, tracked and asserted.

Therefore, a good practice in developing models is to impose the "physical cage" by assertions so that the assignment of values does not get out of range easily. For example:

- assertion(bird.x >=xmin && bird.x<=xmax && bird.y>=ymin && bird.y<=ymax, "Bird cannot get out of cage");
- assertion(map[robot.x][robot.y]==0, "Robot's location can only be set to an unoccupied cell");
- if (input == x1) { assertion(ouput == y1, "We knew that output for x1 is y1"); }

11.4.4. Reality gap

There is a coherent gap between continuous, smooth behaviors in the real world and discrete, abrupt behaviors in the model – due to the information drop for model checking purpose. Particularly in robotics, geometric/kinematic computations use real numbers, while in a Rebeca model state variables have to be discrete values (*integer*, *byte*, *boolean*) so that the model checker can exhaustively test the model. The existence of the reality-model gap requires strategies to bridge it in both ways:

- Discretizing (or integerizing) when going from implementation to model, that means doing modelling. Continuous real values must be discretized to representative values while retaining the same semantics. We have applied this strategy in modelling the robot's directions and laser scanning.
- Smoothing when going from the model to implementation, that means implementing what the model specifies. For example, instead of just 8 directions as in the model, the real robot can rotate to any angle; or instead of abruptly setting the robot's location to a point, it can move gradually and smoothly to that in reality.

Due to this coherent gap, it can be said that model checking is necessary but not sufficient to guarantee the absence of failures in the real robotic system. If there is a detected problem in the model, there should be the same problem in the implementation – provided that the implementation and the model are in sync (the "*last link*" synchronization). But if there is nothing wrong with the model, we can only say that we have more confidence in the design and there is much less probability of failure in the real deployment. It is very analogous to constructing a bridge: if there is no problem in the model of the bridge, we have more confidence in the real bridge but we cannot say that the bridge is 100% safe because there is nothing wrong with the model.

Currently there is no systematic, exhaustive way of verifying the correctness of the model. We can only make sure that the code of the model and the code of the program implement the same logic, and test the synchronization between the model and the program by trying several working and non-working cases to show that under the same inputs, they produce the same outputs.

11.4.5. Dimension reduction

Simplification is a key feature of modelling. Robotics uses complicated 3D geometric computations. In modelling for model checking, we have to reduce this complexity by projecting to the objects to 2D surface(s). Collision is detected by checking the intersection of occupied bits between objects' projected bit-map images ("*footprints*"). In this project, to reduce complexity, we do not use the footprints of the robots, collision is determined only by checking if two robots are too close to each other.

The 2D projection technique can be used to detect collision potential in other contexts – like with robotic arms – if we can compute the footprint of the robot in each pose.

12. Conclusions & Future work

Robotics is a complex, interdisciplinary field. The more complex, the larger a system is, the more natural & beneficial it comes to the need of modelling the system before building it. Model-based development and verification allows early prototyping and troubleshooting. ROS2 is a popular middleware stack to build robotic applications. The similarity in concurrent, asynchronous, message-based characteristics of ROS2 nodes and Rebeca reactive objects justifies the relevance of bringing Rebeca modelling language and model checker to aid the development and verification of ROS2 applications. To make it work, there must be a systematic approach in handling the complexity of robotics, bridging the gap between continuous behaviours in a physical world and discrete behaviours in a model for model checking, overcoming the unmatched programming facilities between robotic development languages in ROS2 and modelling language in Rebeca. This project has successfully identified and strategically dealt with those challenges to deliver a highly precise and performant model of a realistic multiple autonomous mobile robots, using it as a base for development of corresponding ROS2 application and proving the synchronization between the two. Run-time correctness properties like target reachability, deadlock freedom, collision freedom have been automatically verified by the model checker and proven to match with real behaviours in the actual system. The knowledge discovery process, modelling principles, development practices, final deliveries and all lessons learnt along the way are all valuable assets of this project that can be adopted for many other contexts.

For a future work to continue and expand the findings of the project, more attempts may be made to look at these matters:

- How to model robotic arms? Which properties can be verified and how to verify them?
- How to model and verify sequence & timing properties in robotics?
- Is there a chance to integrate the model checker into the simulation software so that we can have everything in one place?
- Is it possible to generate Rebeca model(s) out of an existing ROS2 code to fully automate the model-based verification process?

13. References

1. Saeid Dehnavi et al., 2019. “Towards an Actor-based Approach to Design Verified ROS-based Robotic Programs using Rebeca”. Procedia Computer Science, Volume 155, 2019, Pages 59-68, ISSN 1877-0509. https://doi.org/10.1016/j.procs.2019.08.012

2. Saeid Dehnavi, 2019. “ROBOREBECA: A NEW FRAMEWORK TO DESIGN VERIFIED ROS-BASED ROBOTIC PROGRAMS”. Master thesis, Mälardalen university.

3. Kostiantyn Sharovarskyi, 2020. “SOFTWARE FOR SAFE MOBILE ROBOTS WITH ROS2 AND REBECA”. Master thesis, Mälardalen university.

4. Arni H. R., Marjan S., Luca A., Matteo C., Ali J., Anna I., Steinar H. S., 2014. “Modelling and simulation of asynchronous real-time systems using Timed Rebeca”. Sci. Comput. Program. 89: 41-68 (2014)

5. https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html

6. https://docs.ros.org/en/foxy/Concepts.html

7. https://osrf.github.io/ros2multirobotbook/ros2_design_patterns.html

8. Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. 2019. “Formal Specification and Verification of Autonomous Robotic Systems: A Survey”. ACM Comput. Surv. 52, 5, Article 100 (September 2020), 41 pages. https://doi.org/10.1145/3342355

9. Filip Berlin, Sebastian Granath, 2021. “Obstacle Detection and Avoidance for an Automated Guided Vehicle”. Master thesis, Linköping University.

10. Qianqian Lin, Shuling Wang, Bohua Zhan, Bin Gu. “Modelling and Verification of the RTPS Protocol Using UPPAAL and Simulink/Stateflow”. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 33(1): 1–27 July 2020. DOI 10.1007/s11390-015-0000-0

11. Müller-Olm, Schmidt, D., & Steffen, B. (1999). “Model-Checking: A Tutorial Introduction”. Lecture Notes in Computer Science, 330–354. https://doi.org/10.1007/3-540-48294-6_22

12. https://www.questionpro.com/blog/exploratory-research/

13. https://en.wikipedia.org/wiki/Rotation_of_axes

14. http://motion.cs.illinois.edu/RoboticSystems/CoordinateTransformations.html

15. https://www.intel.in/content/www/in/en/robotics/autonomous-mobile-robots/overview.html#

16. https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem

17. https://en.wikipedia.org/wiki/A*_search_algorithm

18. https://github.com/qiao/PathFinding.js

19. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

20. https://index.ros.org/p/nav2_navfn_planner/

21. https://navigation.ros.org/configuration/packages/configuring-smac-planner.html

22. https://docs.ros.org/en/foxy/index.html

23. Per Bjesse, 2005. "What is formal verification?". ACM SIGDA newsletter volume 35, issue 24, 15 December 2005, pp 1–es, https://doi.org/10.1145/1113792.1113794

24. Edmund M. Clarke et. al, 2018. "Handbook of Model Checking". Springer International Publishing. https://doi.org/10.1007/978-3-319-10575-8

25. Alan Robinson, Andrei Voronkov, 2001. "Handbook of Automated Reasoning". Elsevier Science Publishers B.V., 2001.

26. Edmund M. Clarke et. al, 2018. "Model checking, second edition". ISBN: 9780262038836, The MIT Press, December 4, 2008.

27. Christel Baier and Joost-Pieter Katoen, 2008. "Principles of Model Checking". ISBN: 9780262026499, The MIT Press, April 25, 2008.

14. List of figures

15. List of tables