The Journal of Systems and Software 244 (2027) 113113

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Modelling and model-checking a ROS2 multi-robot system using Timed Rebeca☆

Hiep Hong Trinh [a],*, Marjan Sirjani [a,b], Federico Ciccozzi [a], Abu Naser Masud [a], Mikael Sjödin [a]

[a] Mälardalen University, Universitetsplan 1, 721 23 Västerås, Sweden
[b] KTH Royal Institute of Technology, Brinellvägen 8, 114 28 Stockholm, Sweden

ARTICLE INFO

Keywords:
Multi-robot systems
Modelling
Model-checking
Formal verification
ROS2
Timed rebeca

ABSTRACT

Model-based development accelerates prototyping, enables earlier experimentation, and ensures rigorous validation of system design intents. In multi-agent systems with complex asynchronous interactions and concurrency, formal verification — particularly model-checking — offers an automated means of confirming that desired properties hold. Timed Rebeca, an actor-based modelling language supporting reactive, concurrent, and timed behaviours, together with its model-checking tool, provides a powerful framework for this purpose. By leveraging these capabilities, Timed Rebeca can intuitively capture ROS2 node graphs, recurring physical signals, motion primitives, and other time-convertible behaviours.

Nevertheless, modelling and verifying multi-robot systems entail significant challenges: abstracting intricate information, bridging the gap between discrete models and continuous system dynamics, and managing large state spaces while preserving fidelity. To address these challenges, we propose discretization strategies tailored to various data types and identify thresholds of abstraction that balance accuracy and tractability. We further introduce optimization techniques to accelerate verification.

Our work demonstrates how to systematically design and verify multi-robot systems through Timed Rebeca, efficiently transform continuous dynamics into discrete models for model-checking, and maintain a practical, bidirectional flow between the abstract model and the ROS2 implementation. The accompanying Rebeca and ROS2 codebases, made openly available, serve as a foundational reference for researchers and developers aiming to model and verify advanced autonomous robotic systems.

1. Introduction

System modelling offers a purposeful abstraction of a physical system that enables engineers to concentrate on its salient features, thereby facilitating the prediction of system qualities, the verification of specific properties, and communication with diverse stakeholders (Brown, 2004). Within this framework, model-checking serves as an effective method for automatically verifying properties of finite-state concurrent systems (Clarke et al., 2018), providing a deterministic outcome for each given scenario. Yet, the validity of model-based verification relies critically on the accuracy of the underlying model—a step commonly overlooked in practice.

Rebeca (Sirjani et al., 2004; Sirjani, 2006) is an actor-based (Agha, 1986) language specifically designed for modelling concurrent and reactive systems. Its extension, Timed Rebeca, adds temporal constructs that allow designers to handle timing semantics (Reynisson et al., 2014; Sirjani et al., 2020). Offering a user-friendly syntax and an integrated model-checker with an IDE, Rebeca provides a comprehensive platform for building and verifying complex system designs (Sirjani, 2018). Meanwhile, Robot Operating System (ROS), currently in its second generation (ROS2), is a well-established middleware for constructing robotic software (Open Robotics, 2023b). Ensuring functional correctness, safety, and reliability in multi-robot systems poses significant demands on both design and implementation. In such cases, modelling and model-checking can uncover concurrency issues early, substantially reducing overall development time and cost.

In this work, we leverage Timed Rebeca to create a holistic model of a multi-robot system and operational environment, complemented by equivalent ROS2 simulation code that preserves semantic alignment with the model. Using the Timed Rebeca model-checking tool, Afra (Khamespanah et al., 2023), we systematically test behavioural

☆ Editor: Dr Antonio Filieri.

* Corresponding author.
E-mail address: hiep.hong.trinh@mdu.se (H.H. Trinh).

https://doi.org/10.1016/j.jss.2026.113113
Received 18 March 2025; Received in revised form 10 September 2026; Accepted 16 September 2026
Available online 9 October 2026
0164-1212/© 2026 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

algorithms, determine operational parameters and their respective safety thresholds, and verify properties such as goal reachability, deadlock freedom, and collision avoidance. Repeated simulations under analogous conditions in ROS2 confirmed the predictions of the model-checker, including rare safety issues that are not always evident through conventional simulation alone.

Using models for developing and verifying a multi-robot system introduces inherent challenges: selecting an appropriate level of abstraction for a complex setup, reconciling discrete model representations with continuous physical behaviours, and containing the computational explosion that can arise during verification. To meet these challenges, we propose different discretization strategies for different kinds of information and behaviours (e.g., map representation, robot movements, laser-based range detection), optimization techniques on discrete data, and confidence-based verification of semantic alignment between the model and its implementation counterpart. The novelty of this work is a pragmatic modelling approach that crafts a discrete model with good-enough abstraction level[1] for analysing continuous robotic behaviours through model-checking rather than mere simulation, enabling verification over all possible executions. This is especially challenging in multi-robot systems, where concurrent autonomous behaviours cause state-space explosion.

Compared to existing literature (see Section 10), our principal contributions and distinctions are:

1. Modelling the architecture of a representative multi-robot system employing Timed Rebeca, a timed actor-based language; and showcasing a round-trip engineering process that maintains semantic consistency between the abstract model and the deployed code (Sections 4, 5, and 7).
2. Developing strategies for mapping continuous physical behaviours to discrete formal models through diverse discretization techniques that effectively reduce the state space without compromising model fidelity. Combined with techniques for optimizing computations on discrete data, like the use of precomputed data and lookup tables, the model can be model-checked efficiently (Section 6).
3. Demonstrating a practical approach to model validation by comparing the structures and behaviours of the Timed Rebeca model and the ROS2 code, and ensuring that the model-based verification results align with the actual behaviours (Section 8).

The remainder of this paper is organized as follows: Section 2 lays out foundational concepts. Section 3 describes the design and architecture of a multi-robot system using node graph in the context of ROS2. Section 4 explains our workflow, starting from node graphs and moving to ROS code and Rebeca model in an iterative and incremental way. Section 5 details our modelling approach with Timed Rebeca, while Section 6 addresses challenges encountered during modelling and the solutions adopted. Section 7 outlines the ROS2 code structure that mirrors the model. Section 8 presents our experimental setup and results, and Section 9 offers a broader discussion of the findings, limitations and technical soundness of our solution. After presenting the work, we compare it with others in Section 10. Finally, Section 11 concludes the paper.

2. Background

2.1. Robotics, formal verification, and multi-robot systems

This work originates from a master thesis project in Trinh (2023). Model-checking (Luckcuck et al., 2019) is the most used technique because of its automation and similarity to exhaustive testing. Barriers of adoption include: (1) formal methods are perceived as a complicated extra step without adding final values, (2) lack of appropriate tools, (3) lack of interoperability between formalisms, (4) lack of clear guidance, and (5) difficulty in formalizing the step from a formal model to program code. The implementation code can be automatically generated or manually written to reflect the model; how to ensure that the code correctly represents the model is an open question. We consider these barriers in our work, in particular verifying the model's validity.

Obstacle detection and avoidance algorithms are the core components of multi-robot systems. These algorithms based on occupancy grid and A* path planning are studied in the context of Automated Guided Vehicle (AGV), which helps us to understand the navigational problem of real-world bulky mobile robots (Berlin and Granath, 2021). Multi-robot systems should not be simply represented by dimensionless dots. The study on AGV motivates the need for verifying safety properties in multi-robot systems, especially in environments with human presence. Formal models are used in multi-robot path planning problem, reinforcement learning techniques are used to synthesize robot behaviour, and model-checking and simulation techniques are used to evaluate task-level metrics of multi-robot performance. However, the success of these techniques is limited due to the model's accuracy, in particular the ability to capture and predict run-time behaviour. It has been suggested that future research should focus on developing smaller multi-robot models by imposing realistic assumptions over the sources of uncertainty and robot interactions so that they can be solved efficiently without sacrificing model accuracy (Street et al., 2023).

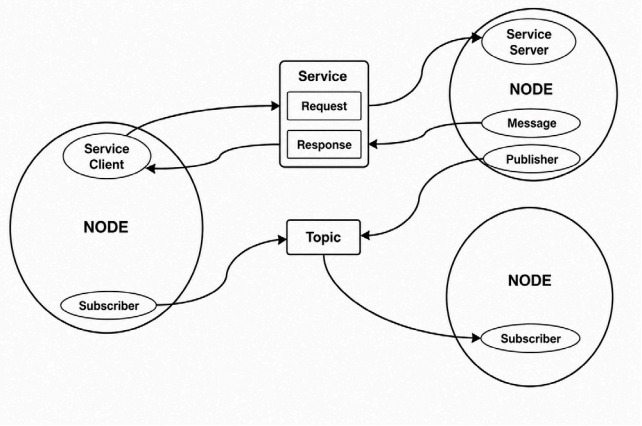

Fig. 1. A ROS2 node graph. *Source:* Adapted from docs.ros.org.

2.2. 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 to achieve some robotic goals (Open Robotics, 2023c,a; OSRF, 2023). ROS2 nodes do not share states, they only interact through messages. For each event, a callback function is defined to specify the behaviour of the node when the event happens. Major communication mechanisms of ROS2 nodes are publish–subscribe topics, client–server services, and actions (see Fig. 1).

In this article, the term node is used in the broader distributed-system sense, referring to any computational entity in a distributed or multi-agent system. When such an entity is implemented in ROS2, it is specifically referred to as a ROS2 node.

[1] Also called model resolution, granularity level, fine-grained level.

Model	::=	$(Class)^+$ *Main*
Main	::=	`main` {*InstanceDecl**}
InstanceDecl	::=	*C r* (⟨*r*⟩*) : (⟨*c*⟩*)
Class	::=	`reactiveclass` *C* {*KnownRebecs Vars MsgSrv**}
KnowRebecs	::=	`knownrebecs` {*VarDcl*}
Vars	::=	`statevars` {*VarDcl*}
VarDcl	::=	⟨*T v*⟩;
MsgSrv	::=	`msgsrv` *m*(*VarDcl*) {*Stmt**}
Stmt	::=	*v* = *e*; | Call | `if`(*e*) MSt [`else` MSt] | `delay`(*t*);
Call	::=	*r.m*(⟨*e*⟩*) [`deadline` *e*] [`after` *e*]
MSt	::=	{*Stmt**} | Stmt

Fig. 2. Abstract syntax of Timed Rebeca. Angle brackets ⟨⟩ denotes meta parenthesis, superscripts + and * respectively are used for repetition of one or more and repetition of zero or more times. Combination of with repetition is used for comma separated list. Brackets [] are used for optional syntax. Identifiers C, T, m, v, c, e, and r respectively denote class, type, method name, variable, constant, expressions, and rebec name, respectively.

2.3. Actor model and Timed Rebeca

The actor model (Agha, 1986) is a simple yet robust architecture for concurrent distributed systems. Each actor is a reactive unit with its own state, a queue of incoming messages to process, and a set of handlers that define how it should respond to each kind of message. Actors do not share states and only interact through asynchronous message passing. An actor can send messages to itself and other actors. Messages sent to an actor are put to its local queue and processed one by one in a run-to-completion manner. This way removes the complication in mutual exclusion of concurrent activities and basically eliminates deadlocks. The actor model is very suitable for modelling and implementing multiple autonomous agents systems.

Rebeca is an actor language for modelling and model-checking reactive, concurrent systems. The formal verification mechanisms are hidden underneath, modellers just need to focus on defining the structure and behaviour of the system through the high-level actor language. A model in Rebeca consists of environmental constants, blueprints of actor types (called *reactiveclass*), and a *main* function where instances of actors (called *rebec*) are declared and activated. A reactive class contains declarations of state variables, known rebecs that it may interact with, message server methods (*msgsrv*) whose name, arguments and body map to type, parameters and the way a message is processed. The class can also contain ordinary methods for internal use only.

Extract 2 shows the syntax of Timed Rebeca with three language primitives to augment message processing with time attributes (Reynisson et al., 2014; Sirjani and Khamespanah, 2023):

- **delay(t)**: to model the advancement of time during message execution in a *msgsvr*,
- **deadline(t)**: to specify expiry time of a message (the timeout to handle the event), e.g. `r1.doLaserScan() deadline(100)`,
- **after(t)**: to model time of a future event or message delivery time. Periodic events are modelled by a self-looping message server with a fixed repeat period, while discrete events of variable scheduled times are modelled with a dynamically computed period, e.g., `r1.updateMovingStatus() after(time_taken)`.

Each actor has a logical clock to order messages by their logical timestamps and all logical clocks are assumed to be fully synchronized. This assumption is independent of physical clocks, which can be unsynchronized and need to map to a shared timeline as in the model. A Timed Rebeca model can be equivalently converted to a set of timed automata (Khamespanah, 2018). Properties can be defined as assertions over state variables. The model defined in Timed Rebeca is compiled and checked by the built-in model checker to verify desired properties. The support of actor model, time semantics and language friendliness (Sirjani, 2018) reasons our selection of Timed Rebeca[2] for modelling autonomous robot systems.

2.4. Discretization

Model-checking relies on a discrete finite-state model for exhaustive exploration of the state space (that is why state variables have to be bounded enumerable types like *integer*, *boolean* or fixed-size arrays of such), while dynamics like movements in a robot system are often continuous, expressed by *real* – *number* variables. When mapping a continuous system to a discrete model, we perform a process called discretization (Fishwick, 2007), which involves sampling the continuous behaviour at selected intervals and approximating it with representative values. The resulting discrete model preserves the relevant system dynamics within an acceptable level of accuracy. Discrete abstraction allows modelling a continuous system as a discrete-time or discrete-state system, which can be more easily simulated and analysed using discrete mathematics and computational techniques.

3. Designing a multi-robot system using a node graph

To demonstrate our approach, we need a multi-robot system to work with. However, since available systems often deal with only one robot or contain many unnecessary features, we have to create an exemplary one to gain full control. Finally, we designed a hypothetical multi-robot system, modelled it in Timed Rebeca, and created corresponding ROS2 simulation code[3] (*the code* in the remainder of the paper).

A multiple autonomous mobile robots (AMR) system was chosen for the following reasons: (1) we can project the robots and obstacles onto a 2D map to track their movements and interactions, and handling 2D geometry is less complicated than 3D (like with robotic arms), provided limited mathematical facilities of modelling languages; (2) a mobile robot is equal to a mobile obstacle, thus the multi-robot problem subsumes the single-robot one which requires modelling other non-static obstacles (like a wandering human); (3) obstacle detection, path finding, collision avoidance are typical robotic problems to deal with and successful modelling practices with these can be generalized and inherited for several other cases; and (4) with multiple robots moving in the same area, the need of guaranteeing collision and deadlock freedom is more demanding, therefore model-checking is more valuable.

For effective modelling and simulation, it is essential to develop a thorough understanding of the problem domain, specifically in the context of robotics and multi-robot systems. Without it, there is a risk of either oversimplifying or overcomplicating the scenarios being addressed. This section provides a detailed explanation of the structure and operation of a system involving multiple mobile robots.

3.1. Basic structure

A multi-AMR system can be designed with variations in architectural topography and communication flows, but consists of the following core elements:

- A floor map: which is a 2D projection of static obstacles on the floor plane. The map is treated as a known environmental knowledge to the robots.

[2] Herefrom we use Rebeca and Timed Rebeca interchangeably.

[3] Full modelling code: https://github.com/thhiep/ros2rebeca_model.

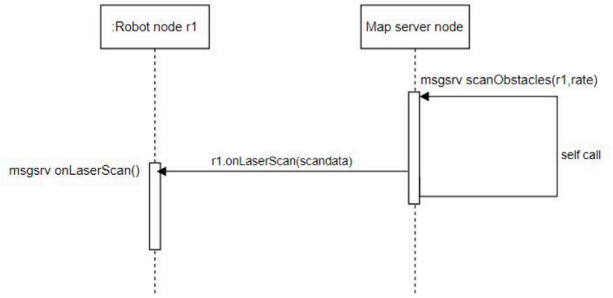

Fig. 3. Modelling laser scan events.

- Robot nodes: each robot has an ID, a footprint shape, a current pose (x, y, θ), linear velocity $\vec{v}$ and angular velocity $\vec{\omega}$, a queue of targets (intermediate and final) to reach, a planned path trajectory to walk, and a set of responses that defines how it should react to different events. Each robot reports its status to a map server node for system-wide observation and supervision.
- Map server node: a central coordinating node that keeps track of the current positions of the robots on the map, provides common services, and checks system-wide properties during execution.

The physical dimensions of a robot must be considered as they affect algorithms for collision detection and path planning. As a rectangular shape is more representative for vehicles, robot shapes are generalized to rectangular profiles in the model.

3.2. Obstacle detection

LIDAR sensors are extensively used for range detection (Yalcin et al., 2013). In ROS2, laser scan data is provided by the physical sensor and encapsulated in a standard message format.[4] In modelling and simulation, this data is generated programmatically. In the discrete model, the generation is done with discrete intervals (refer to 6.1) and the periodic recurrence is simulated through a timed loop (see Fig. 3). The robot maintains a matrix of the nearest obstacles, which it uses to identify obstructions in a given direction. This matrix is updated with each newly scanned signal; however, as the robot moves, the stored obstacle data becomes outdated since the robot is no longer in the same position as when the last scan was received. Consequently, there is a critical safety relationship between the robot's movement speed and the scanning rate, ensuring the robot does not move faster than the rate at which obstacle data is updated. This relationship is analogous to the Nyquist–Shannon theorem (Shannon, 1949), which states: *The sampling frequency of a signal must be greater than twice the highest frequency of the input signal*. In this context, the time a robot takes to perform an action should be at least twice the period of the signal it relies upon, i.e.,

$$action_time > 2 * signal_period \tag{1}$$

3.3. Collision detection

In the real world, a physical collision occurs without defining how it happens logically. However, in modelling and simulation, it is necessary to define the conditions under which a software-detected collision occurs. For round robots, collision detection can be simplified by measuring the distance between the centres of two robots. For rectangular robots, this method is insufficient, as they may be positioned parallel and close to one another without any actual collision. In general, a collision is detected by checking for an intersection between the bounding areas (shadows) of two robots (see Fig. 4). Based on each robot's pose and dimensions, we can calculate its current boundary, extended by a safety margin, and determine whether it intersects with the extended boundary of another robot.

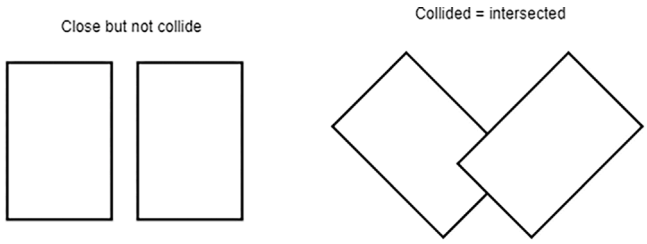

Fig. 4. Collision check based on intersection of shadows.

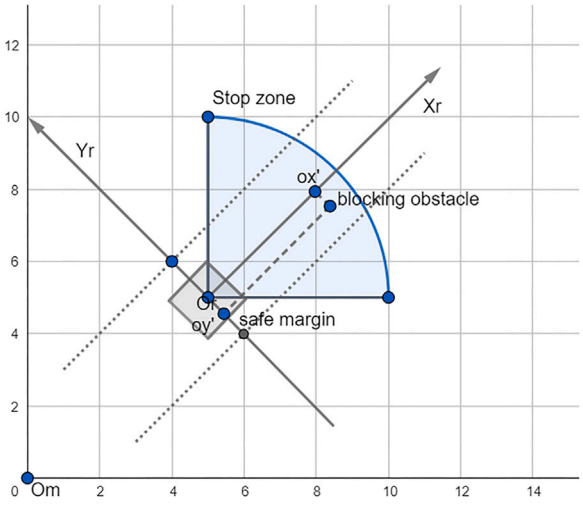

Fig. 5. Blocking obstacles.

3.4. Navigational problems

A mobile robot typically utilizes a floor map to navigate. In ROS2, the map data is saved to a portable grey map (PGM) image file, where the grey level of each pixel depicts the occupancy status of a point. The map file is then parsed into an occupancy grid (with resolution defined in a YAML file) to facilitate graph-based pathfinding algorithms, of which A* (Wikipedia, 2023; Hart et al., 1968) and its variants are widely used. Pathfinding is an optimization problem with no guaranteed shortcut to a predictable solution. Therefore, in modelling, we must implement a pathfinding algorithm to generate walkable paths and ensure semantic alignment between the model and the code (that is why we opted not to use an existing library like Nav2). However, the path is collision-free only at the time of generation and is not future-proof, as mobile obstacles may appear or move unexpectedly.

When the robot encounters obstacles along its route, it must decide whether to stop and wait, continue moving, or reroute to avoid obstacles or clear congestion. We developed a behavioural algorithm in Alg. 1 inspired by how humans walk or drive in real life. The algorithm follows a *look-think-act* sequence. Obstacles are categorized as either blocking or non-blocking. A blocking obstacle lies directly in the robot's path, while a non-blocking obstacle is outside of the robot's immediate frontal area (see Fig. 5). Even if an obstacle is close to the robot, it is considered non-blocking if it does not impede forward movement. The available frontal space is determined by estimating the distance to the nearest object in the robot's current direction. The robot then uses a heuristic value to assess whether there is sufficient space to proceed;

[4] https://docs.ros2.org/foxy/api/sensor_msgs/msg/LaserScan.html.

otherwise, it stops and waits. This heuristic is influenced by factors such as inertia, speed, and braking distance. However, for simplicity in the modelling process, this value is set as a constant (STOP_ZONE).

Algorithm 1: Human-like movement algorithm

```
procedure moveNext(P, micro_step, wait_inc) begin
    Let w_0 be the current position of the robot
    Let P = {w_1, ..., w_n} be the generated path where w_1 is the next
      waypoint and w_n is the goal
    if (P is empty) then
        // final waypoint reached;
        Stop(); return;
    end
    First, rotate the robot to the next waypoint w_1 so that it looks
      towards that way (skip if already)
    Estimate d = distance to the nearest blocking obstacle in that
      direction
    Let h = some heuristic value of required frontal space to be able
      to move forward safely
    if (d > h) then
        //move one microstep forward, reset waiting status;
        wait_count ← 0;
        w_0 ← w_0 + micro_step;
        if (Robot has reached w_1, meaning w_0 ≡ w_1) then
            Remove w_1 from stack P
        end
        moveNext(P, micro_step, wait_inc)
    else
        Stop(); Wait(wait_inc);;
        wait_count ← wait_count + wait_inc;
        if (wait_count >max_waiting_time) then
            Resolve congestion
        else
            // continue waiting or moving
            moveNext(P, micro_step, wait_inc)
        end
    end
end
```

Algorithm 2: Congestion resolving algorithm

1 Let $getObstacleAtDir(dir)$ be the function for estimating distance to the nearest blocking obstacle in a direction dir
2 Let $backdirs$ be the set of directions on the opposite side of the current direction
3 Let od be a list of $\{dir, freespace\}$ records
4 Estimate free space in all back directions:
$foreach(dir\ in\ backdirs)$
$od.push(\{dir, getObstacleAtDir(dir)\})$;
5 Sort od by descending order of $freespace$ field
6 Pick a random direction among the top (e.g. 3) group
Let $selected = random(1, max(count(od), 3)) - 1$
Let $backdir = od[selected].dir$;
7 Pick a random number of possible backoff steps in the selected direction (e.g. 3 to 10 steps):
Let $backsteps = random(3, max(od[selected].freespace, 10))$;
8 Walk the robot back to the "$backdir$" direction by "$backsteps$" steps, then request a new path from the new location to the destination

4. Modelling and development workflow

The modelling and development workflow of our multi-robot system is depicted in Fig. 6. In the first phase, we select a robotic problem to model, and design the architecture as a node graph. Next, we develop Timed Rebeca model based on the node architecture. We use a few scenarios which we expected to pass the model checking, for example, a simple scenario including only a single robot or scenarios with multiple robots with easy navigation paths from source to destination with low possibility of collisions. In the third phase, we develop the ROS2 code based on the node graph and the Rebeca model. Since the Rebeca model lacks visualization, some issues are identified during ROS2 execution with visualizers. Examples of these issues include using not optimized paths by the robots, or avoiding obstacles and rerouting too early. Modifications are made to the ROS2 code to remove such problems. In the fourth phase, the changes are propagated back to the model to ensure semantic consistency between the model and the ROS2 code. We also calibrated the model in terms of granularity level until it behaves the same as the code under selected scenarios. If we observe the Timed Rebeca model behaving differently compared to the ROS2 code, we need to revise the model again. For example, when we check the paths or when we observe a collision or deadlock in the simulation but model checking did not detect it, we conclude that the precision of the model is not enough and we increase the granularity.

The model is validated when it passes adequate synchronization checks, in which we compare the model and the implementation code in terms of structural and behavioural equivalence. Structural equivalence is based on the node graph of both the model and the code, and behavioural equivalence includes same information flow, same behavioural algorithms (path finding, movement, obstacle avoidance and congestion resolution), same generated paths and final outcome like lack of deadlock and collisions, under same settings of robot and map configuration. The ROS2 visualization and the traces in the state space generated by Afra IDE are used for comparing paths.

In the final phase, we perform the verification of the ROS2 code by model checking the Timed Rebeca model and verifying safety properties under different scenarios. If problems are detected, the loop can be restarted to modify the model, the code, or the design of the system.

5. Modelling the system in Timed Rebeca

5.1. General mapping rules

ROS2 nodes are equivalent to reactive objects (*rebecs*) in Rebeca. Service requests and responses in ROS2 are mapped to Rebeca messages and message servers with discretized and bounded arguments. Timers and periodic signals are modelled using timed loops in Rebeca via the $after()$ primitive. Real-number variables are subject to different discretization mechanisms in the model. General mapping rules are summarized in Table 1, items marked with (*) are discretized and explained in the next sections. We can see that all robotic information types are captured in Rebeca with equivalent or abstracted representation.

5.2. Modelling state evolution over time

For each robot, the progression of states over time is modelled as a sequence of timed events as outlined in Alg. 3. Based on the current state and external events (e.g., the arrival of a new laser scan or a new path), the next state and its occurrence time are computed. State transitions and time advancement are simulated by using a time specifier in Rebeca, typically the $after()$ primitive.

Algorithm 3: Time-wise discretization of state

1 Let loc_1 be the current pose (state) of the robot
2 Let loc_2 be the next pose of the robot (if the robot is waiting, loc_2 is same as loc_1)
3 Let Δt be the time taken to switch to loc_2
4 Then: `setNextState(`loc_2`) after(`Δt`)`

The transition time, denoted as Δt, is calculated based on the duration a robot takes to either:

- change its direction: $\Delta t = |\theta_2 - \theta_1|/\omega$, where ω is rotation speed, θ_1, θ_2 are current and next angles,

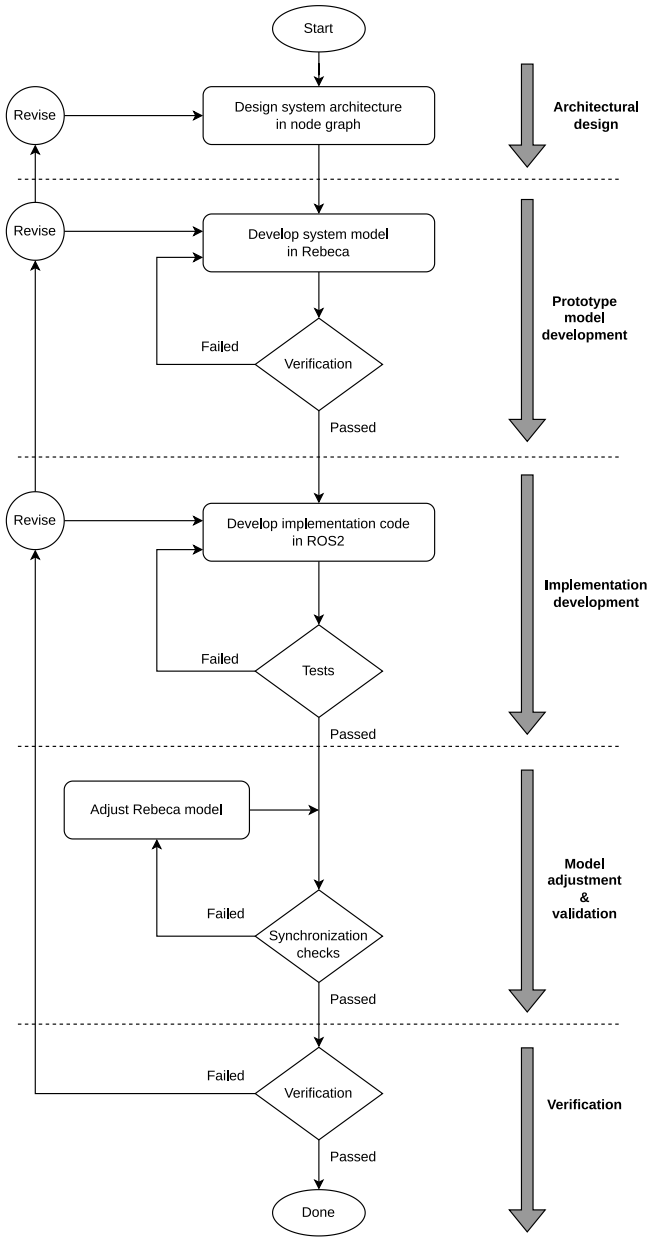

Fig. 6. The workflow for designing, modelling, implementation and verification.

- move to an adjacent cell: $\Delta t = s/v$ where v is linear speed, $s =$ `CELL_WIDTH` if moving along the edge, or $s =$ `CELL_WIDTH` $* \sqrt{2}$ if diagonally, or
- continue waiting: $\Delta t =$ `waiting_increment`.

In simulation, the robot's movement is updated by small steps (every 5–10 ms) to simulate the smooth continuous movement in reality, while in the model, it is done by discrete macro-steps since it only "hops" from cell to cell or rotates by a multiple of 45°. This approach allows the conversion of velocities into time, enabling the use of Rebeca's time semantics to model the timing sequence of the movement updates accurately. The time-wise discretization of states enables capturing only meaningful moments, skipping unnecessary iterations in the middle — which are typically found in a tick-based time model, causing state explosion and inefficiency.

Table 1
Robotic information in ROS2 application and in Rebeca model.

Information type	In ROS2	In Rebeca
Parameters	Constants in launch file	*env* constants
Floor map	PGM and YAML files	Equivalent occupation grid (*)
Node definition	ROS2 node class	Reactive class
Node instance	Instance of ROS2 node class	Instance of reactive class (*rebec*)
Event type	ROS2 message structure	*msgsvr* name and formal arguments (*)
Event instance	ROS2 message	*msgsvr* actual arguments (*)
Event response	ROS2 callback function	*msgsvr* body
Pub-sub channel	ROS2 pub-sub topic	Data publisher is a *msgsvr* of the service provider *rebec*. Data subscriber is a callback *msgsvr* in the service consumer *rebec*.
Robot physical shape	URDF file	Shape of shadow (rectangular)
Robot pose	ROS2 odometry message	Grid cell and direction $(i, j, rdir)$ (*)
Range detection	ROS2 laser scan messages	Simulation of scanning result (*)
Movement primitives	Periodic micro-step updates (5–10 ms)	Next pose scheduling with varying Δt (*)
Speed	Travelled distance in a micro-step	Time taken to transit to next state, Δt
Path finding	A* algorithm	Same
Obstacle avoidance	Look-think-act algorithm	Same
Congestion resolution	Random backoff	Same

5.3. Overall interaction flows and the model's code

Fig. 7 shows the interactions between a robot node and the map server node. The map server node exposes the following services: `scanObstacles` (generating LIDAR sensory data), *generatePath* (applying A* based algorithm for finding paths), `updateRobotLocation` (updating robots' locations on the map). Each robot node implements the following callbacks: `onNewPath` (start moving on the generated path), `onLaserScan` (updating the matrix representing the nearest obstacles and decide to continue moving or stop) and `updateMovingStatus` (deciding in a loop whether a robot can make a move, stop, wait, or reroute).

The full modelling code is provided on GitHub, here we only present its basic structure in Listing 1. In the Timed Rebeca model, first the configuration parameters are defined as *env* constants, and then two reactive classes are defined to model the mobile robots (class *Node*) and the coordination centre (class *MapServer*). The interactions of these two classes is visualized in Fig. 7. In the *main*{} part in the code, the map server and the robots are instantiated. The parameters of the robots denote the ID, starting point, target point, tolerance distance to target, linear speed, waiting time, and a *scanFirst* flag telling whether it should always wait for a new obstacle detection scan before moving. At start, each robot will send `theMap` a `generatePath()` request to create a path to its target. After receiving the path, it will start planning the moves along the computed waypoints. Depending on the situation it has to cope with the obstacles on its path, each robot will decide what to do next — continue moving, stop and wait, or reroute. For rerouting the robot asks the map server for a new path. The model execution stops when no more state evolution is possible, which happens when the destination is reached or a deadlock/livelock is detected.

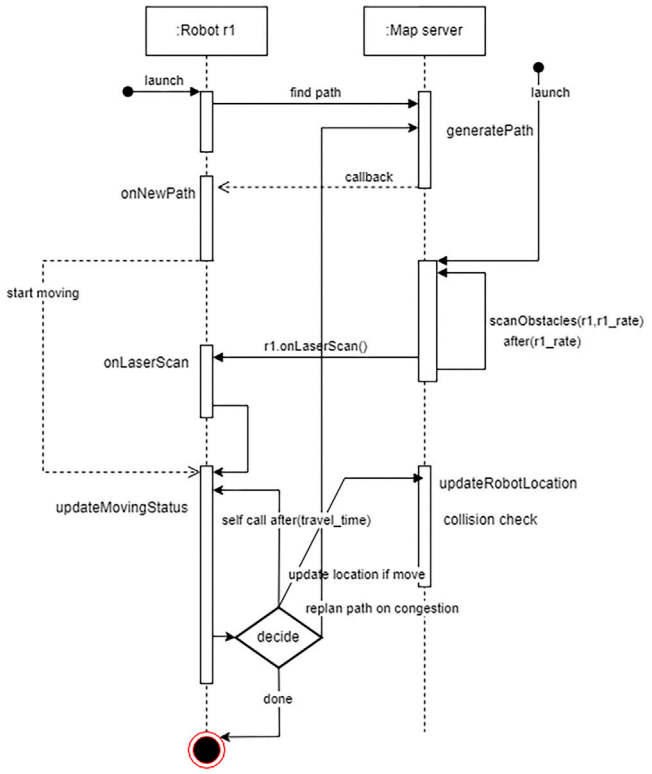

Fig. 7. Sequence diagram of map server–robot interactions.

Listing 1: Modelling code in Timed Rebeca.

```
env int MAP_WIDTH = 50;
env int MAP_HEIGHT = 50;
env double MAP_RESOL = 0.25500;
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 = 0.4;
env double ROBOT_LENGTH = 0.4;
env double SAFE_MARGIN = 0.02;
env double STOP_ZONE = 0.3;
env double COLLISION_RADIUS = 0.30;
env double COLLISION_MARGIN = 0.02;

env double ROBOT_ANGULAR_VELOCITY = 360;
env int ANGLE_INC = 2;
env int FIELD_OF_VIEW = 180;
env int LASERSCAN_RATE = 100;
env double LASERSCAN_MAX_DISTANCE = 10.0;

env boolean COMBINE_ZICZAC = true;
env int BUFFER_SIZE = 101;

reactiveclass Node(50) {

    knownrebecs { MapServer theMap; }

    statevars {
        byte rindex;
        int targetX, targetY;
        int rx, ry, rdir, velocity;
        int distance2target;
        int target_tolerance;

        int[101] obstacles;
        int front_obstacle;

        int[11] targets;
        int[101] moves;

        int movedIdx;
        boolean isWaiting;
        int waits;
        int max_waiting_rounds;
        int failures;
        boolean scan_before_move;

        int default_velocity;
        int default_time_cross;
        int default_time_cross_diag;

        int safe_front;
        int safe_side;
    }

    Node(byte idx, int x, int y, int tx, int ty, int tolerance, double real_velocity, int maxwait, boolean scanfirst) {
        // initialization routine
    }

    @priority(3)
    msgsrv onLaserScan(int[101] scandata, int[101] angles) {

        if (isWaiting ||
           (!scan_before_move && distance2target > target_tolerance))
            updateMovingStatus(!scan_before_move);
    }

    @priority(1)
    msgsrv updateMovingStatus(boolean moveNext) {
        // move / stop / wait decision
    }

    @priority(10)
    msgsrv onNewPath(int[101] path) {
        // handle new path
    }
}

reactiveclass MapServer(50) {

    knownrebecs {
        Node r1; Node r2; Node r3; Node r4; Node r5;
    }

    statevars {
        int[21] rloc;
        int collisions;
    }

    MapServer(int rcount) {
        collisions = 0;

        //initialize robot information array rloc

        //start laser scan simulation events...
    }

    @priority(1)
    msgsrv scanObstacles(Node robot, int rindex, int rate, int fov, int max_distance) {

        int[101] scandata;

        robot.onLaserScan(scandata, angles) deadline(rate + 5);

        scanObstacles(robot, rindex, rate, fov, max_distance) after(rate);
    }

    @priority(5)
```

```
110       msgsrv updateRobotLocation(int idx, int rx,
           ↪ int ry, int rdir, int velocity) {
111           // update location + collision check
112       }
113
114       @priority(3)
115       msgsrv generatePath(Node robot, int
           ↪ robot_index, int startX, int startY, int
           ↪ targetX, int targetY) {
116           int[101] path;
117
118           // A* path generation
119           // zigzag optimization
120
121           robot.onNewPath(path);
122       }
123   }
124
125   main {
126       @priority(3)
127       Node r1(theMap) :
           ↪ (1,5,5,45,45,2,0.5,1500,false);
128
129       @priority(5)
130       Node r2(theMap) :
           ↪ (2,5,15,45,35,2,0.7,2000,false);
131
132       @priority(6)
133       Node r3(theMap) :
           ↪ (3,5,25,45,25,2,0.8,2500,false);
134
135       @priority(4)
136       Node r4(theMap) :
           ↪ (4,5,35,45,15,2,0.5,3000,false);
137
138       @priority(2)
139       Node r5(theMap) :
           ↪ (5,5,45,45,5,2,0.3,1500,false);
140
141       @priority(1)
142       MapServer theMap(r1,r2,r3,r4,r5) : (5);
143   }
```

5.4. Correctness properties for model checking

For a multi-AMR system, we aim to ensure the following properties:

1. Collision freedom: At all times, no two robots collide.
2. Reachability (and deadlock freedom): Eventually, all robots reach their destinations.
3. Congestion freedom: The robots do not get stuck or indefinitely blocked.

These properties are formally defined as follows:

Collision freedom: Collisions between robots are detected when the map server node receives and processes robot location updates (*updateRobotLocation*). The abstracted poses of the robots in the discrete world are converted back to the continuous world to detect collision using the mechanism defined in Section 3.3, if a collision is detected, the *collisions* counter is incremented. Collisions with static obstacles are already avoided via the path generation algorithm by keeping the robot at a safe distance from the obstacles. Formally, collision freedom is expressed by the following CTL formula:

$$collisionFree := \mathrm{AG}(theMap.collisions = 0)$$

Reachability: Each robot is assigned a tolerance for reaching its target. A state variable ($distance2target$) tracks the remaining octile distance to the target cell. A robot r is considered to have reached its destination, $done_r = true$, when it is within the specified tolerance of the target (as implemented in ROS2) and has no further planned moves that drive it beyond the target. So, this property is defined as:

$$finallyDone := \mathrm{EF}(\forall r : done_r = true)$$

Deadlock freedom: Deadlocks are automatically detected by the model checker (Khamespanah et al., 2012) when the system reaches a state in which the robots are unable to proceed, preventing further progress towards a final state. Livelock is identified by the *failures* counter that tracks the number of unsuccessful rerouting attempts made by a robot. If the value of this counter exceeds a predefined threshold, the robot is considered to be potentially stuck. This property is expressed as

$$liveLockFree := \mathrm{AG}(\forall r : r.failures \leq tolerance_{failures})$$

In Timed Rebeca, temporal logic operators are equivalently converted to assertions as follows:

- AG(a), meaning a must hold true any time in any execution path, is equivalent to a native assertion expression $p := a$.
- EF(a), meaning finally a becomes true in at least one execution path. To verify that all robots finally arrive at their destinations, we add a counter assertion $notDone := \neg(\forall r : done_r = true)$, which will stop the modelchecker whenever all robots arrive. Violating this assertion means model-checking has passed successfully. Note that the robots are bounded by the collision free and livelock free conditions, which will prevent them from moving around endlessly without reaching targets.

Listing 2 shows the property specification file, where predicates are defined in the *define* block and properties are specified in the *assertion* block.

Listing 2: Property definition in Rebeca.

```
1   property{
2     define {
3       done1 = (r1.distance2target <=r1.target_tolerance)
4         && (r1.moves[0]==0 || (r1.moves[0]>0 &&
           ↪ r1.moves[0]-r1.moveidx<=r1.target_tolerance));
5
6       done2 = (r2.distance2target <=r2.target_tolerance)
7         && (r2.moves[0]==0 || (r2.moves[0]>0 &&
           ↪ r2.moves[0]-r2.moveidx<=r2.target_tolerance));
8
9       done3 = (r3.distance2target <=r3.target_tolerance)
10        && (r3.moves[0]==0 || (r3.moves[0]>0 &&
           ↪ r3.moves[0]-r3.moveidx<=r3.target_tolerance));
11
12      done4 = (r4.distance2target <=r4.target_tolerance)
13        && (r4.moves[0]==0 || (r4.moves[0]>0 &&
           ↪ r4.moves[0]-r4.moveidx<=r4.target_tolerance));
14
15      done5 = (r5.distance2target <=r5.target_tolerance)
16        && (r5.moves[0]==0 || (r5.moves[0]>0 &&
           ↪ r5.moves[0]-r5.moveidx<=r5.target_tolerance));
17
18      noCollisions = theMap.collisions==0;
19
20      noLivelock = r1.failures<=2 && r2.failures<=2 &&
         ↪ r3.failures<=2 && r4.failures<=2 &&
         ↪ r5.failures<=2;
21    }
22
23    Assertion{
24      notDone: !(done1 && done2 && done3 && done4 && done5);
25      collisionFree: noCollisions;
26      liveLockFree: noLivelock;
27    }
28  }
```

6. Modelling challenges and strategies

Rigorous verification of robotic systems is known as a complex, interdisciplinary task while formal methods are often hindered by the state-space explosion problem (Foughali and Zuepke, 2022). Modelling robotic behaviours for the purpose of model-checking presents significant challenges. What are the factors – 'the Bad' and 'the Ugly' – that hinder or complicate the path toward successful modelling?

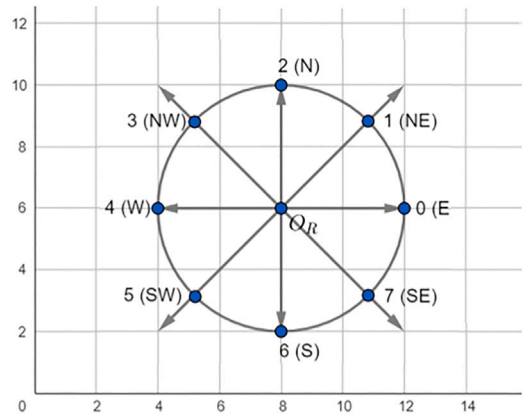

Fig. 8. Discretization of robot directions.

6.1. The bad: Discretization strategies and pitfalls

The primary issue, 'the Bad', lies in the inherent gap between a discrete model and a continuous system. This gap persists, regardless of the specific approximation or sampling technique used. A key challenge in this context is the trade-off between increasing the granularity of the model and managing the resulting state space explosion, a common and critical problem in model-checking (Clarke et al., 2018).

Discretizing robot pose: As the obstacle map is discretized to an occupancy grid, in the model a robot can only be at one cell and move to an adjacent cell in 8 possible directions. These 8 angles are multipliers of 45°, which is convenient for trigonometric computations (see Fig. 8). Using this strategy, we mapped the continuous pose (x, y, θ) from the real system to a discrete pose $(rx, ry, rdir)$ in the model, preserving essential information while ensuring the pose remains sufficiently enumerable for model-checking purposes.

So, from map coordinate to grid:

$$\begin{aligned} rx &= floor(\tfrac{x}{\text{CELL_WIDTH}}); \\ ry &= floor(\tfrac{y}{\text{CELL_WIDTH}}); \\ rdir &= floor(\tfrac{\theta}{45}); \end{aligned}$$

From grid to map coordinate, the robot is conventionally placed at the centre of the cell:

$$\begin{aligned} x &= (rx + 0.5) * \text{CELL_WIDTH}; \\ y &= (ry + 0.5) * \text{CELL_WIDTH}; \\ \theta &= rdir * 45; \end{aligned}$$

Discretizing laser beam angles: In reality, a LIDAR sensor's angular resolution (beam step) can be a small fraction (e.g. 0.02°). In the model, it is discretized to 2° (1 is too narrow, resulting in overlapped scans, while 3 is too wide, resulting in skipping adjacent objects); field of view is set to a multiplier of beam step, typically 180°, 270° or 360°.

Bridging the gap: When discretizing a continuous measurement, there is a risk of information loss between two consecutive steps, which increases as the step size becomes larger. This issue was observed in relation to the grid resolution and the robot size (see Fig. 9). In the model, each movement step corresponds to a full cell edge (CELL_WIDTH) or diagonal (CELL_WIDTH $* \sqrt{2}$). If the step length exceeds the robot's physical size, potential collisions occurring between steps may be overlooked. To mitigate this, the step length must be smaller than the robot's size, ensuring that any mid-step collisions are accurately detected due to the overlap between consecutive steps:

$$\texttt{cell_width} < \texttt{robot_length}/\sqrt{2} \tag{2}$$

If the step size is too small, the state space grows exponentially without further gain in accuracy. Determining the 'enough' threshold is therefore crucial in balancing between correctness and computational efficiency of the model.

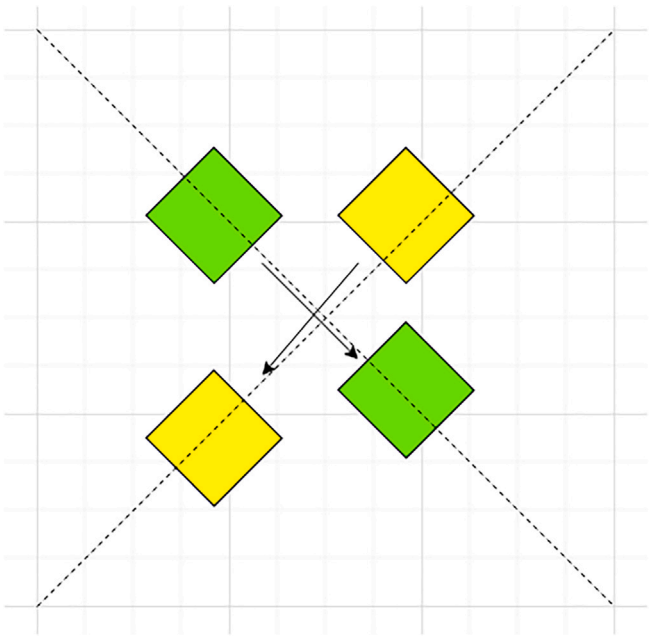

Fig. 9. Grid resolution vs. robot size.

6.2. The ugly: Workarounds and optimization techniques

Modelling robotic systems require both high-level and low-level expressiveness power of the modelling language and platform to capture system architecture and to do unavoidable computations and algorithms same as in implementation (e.g. coordinate translation, path finding). However, formal modelling languages are intended for specifying high-level logic rather than handling low-level computations. There are unmatched programming facilities in Rebeca that need to be coped with[5]:

- Inadequate type system: Rebeca does not support complex collection and structured types, which are necessary for describing robotic data. Arrays can only be of static size.
- Lack of common mathematical, IO and utility functions as well as mechanism for using externally defined ones. Modellers therefore have to recreate their own implementation of very basic functions and cannot reuse existing computation libraries.
- Limited debugging, visualizing capabilities.

To overcome these language-specific inconveniences and to boost computational efficiency in model-checking, we employ several techniques, described in the following.

Helper script: We developed a script in PHP to assist with the testing and troubleshooting of the modelling code: (1) prepare static data: convert between PNG and PGM image file formats, generate occupancy grid matrix and precompute trigonometric values; (2) write script code in Rebeca style to test and debug; (3) visualize map state to troubleshoot algorithms.

Precomputed data and lookup tables: The occupancy grid corresponding to a PGM map file is parsed by the script and saved to a text file. When this data is needed in the model, it is copied to a local variable. As angles (robot angles, laser beam angles) in the model were discretized to integers, we use the script also to precompute cos, sin, tan of all angles from 0° to 360°, saving to a key–value array to look up later. As a result, on-the-fly trigonometric calculations are eliminated,

[5] For the version in use at the time this work was done.

Table 2
File structure of the ROS2 demo codebase.

Root	Subfolders / files	Files and purposes
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

significantly reducing computation time. This optimization is particularly impactful in model-checking, where the same calculations are repeatedly executed during each transition.

7. Developing ROS2 code from the model

After having a working prototype model, we manually[6] develop the corresponding ROS2 code,[7] adhering to the established mapping rules and maintaining the same node structure, interaction flows, and behavioural algorithms as defined in the model. The code runs on ROS2 Foxy version, Ubuntu 20.04.5 LTS on a VirtualBox virtual machine. RViz2 was chosen as the visualizer, instead of Gazebo, because its 2D visualization is lightweight. Gazebo is often used in parallel to broadcast */laserscan* topic for RViz2 to capture; here we simulate it inside the code to have a self-contained, compact solution. Table 2 presents the structure of the ROS2 codebase. The code implements two kinds of nodes, as in the model: map server node and robot nodes. The ROS2 nodes need to generate and broadcast data in standard topics so that RViz2 can capture and visualize on the graphical interface (including occupancy grid, robot state, standard transformations, laser scan, path, and additional markers).

Multi-robot handling: Most open-source ROS2 codebases are designed for single-robot systems. To enable multi-robot functionality, each robot is assigned a unique identifier (e.g., *r*1, *r*2) which also serves as its namespace. This ID is appended — either automatically by ROS2 or manually — to each robot's interfaces and frames (e.g., */r2/laserscan*). A XACRO template is used to define each robot's parameters, such as name, dimensions, and colour, to generate a unique URDF model for each robot. Shared interfaces and frames (e.g., */map*), common to all robots, remain unprefixed.

Change synchronization: In the program, generated paths are optimized to be shorter and more efficient. This is done through *waypoint consolidation* (combining waypoints that lie in the same direction to reduce loop iterations and increase speed) and *zig-zag point elimination* (skipping zig-zag points along diagonals, allowing for straighter and faster diagonal movement). The model is then updated to align with these optimizations introduced in the program.

[6] As an initial step, we chose not to automate the model-to-code translation due to the need to uncover various underlying issues. Automation is planned for future iterations.

[7] Full ROS2 code: https://github.com/thhiep/ros2rebeca_code.

Table 3
Explanation of parameters and counters.

Name	Meaning
Scan rate	Laser scan rate in milliseconds
Speed limit	Threshold for linear speed, bounded by safety relationship with scan rate
Stop zone	A heuristic value the robot uses to decide to stop up to estimated distance to blocking obstacles
Speed	Linear movement speed (m/s)
Wait	Maximum waiting time in milliseconds before the robot decides to back off and find another path
Analysis result	Model-checking result
States	Number of states traversed by the modelchecker
Transitions	Number of state transitions tried by the modelchecker
Simulations	Number of ROS2 code executions under the same settings
Simulation results	Results of ROS2 code executions

8. Simulation and model checking results

We developed the Timed Rebeca model and the corresponding ROS2 code following the modelling workflow described in Section 4. To enable a meaningful comparison between model verification and execution, same configuration parameters were used for both the ROS2 experiments and the model-checking setup.

A randomly generated 50×50 grid map is used, with the resolution set to 0.255 to fit RViz2 viewport. The robot dimensions are set to 0.4×0.4, which satisfies the fine-grained threshold in Eq. (2) (see Section 6.1). An extreme scenario is designed with five robots that are required to navigate intersecting paths to reach their respective targets across the map. This scenario creates a high risk of collisions and congestion, allowing us to rigorously evaluate the correctness properties and assess the effectiveness of the behavioural algorithms. In this scenario, robots may start at arbitrary initial positions and are assigned arbitrary goals. Waypoints are computed by the A*-based algorithm. Due to the concurrent execution of the robots, the system behaviour is non-deterministic.

The experimental parameters and statistical counters are explained in Table 3.

We present two types of scenarios, ***working and non-working,*** each illustrated with three representative examples. For each type of scenario, we provide a table of the corresponding experimental parameters and statistics, along with screenshots from the simulation. These examples demonstrate that the issues predicted by model checking (such as collisions or congestion) indeed occur in the ROS2 executions.

Working Cases (Cases C1a, C1b, C1c). Table 4 provides the statistical and experimental parameters and results for cases C1a, C1b, C1c. All three cases share the same global safety envelope (scan rate, speed limit, stop zone). The speed limit is set to 1.275 m/s, satisfying the timing constraints from Eq. (1), which is formally derived from the sensor scan rate (100 ms) via an analogy to the Nyquist–Shannon sampling theorem. Just as a signal must be sampled at more than twice its highest frequency to be faithfully reconstructed, any robot action must take longer than twice the scan period to be reliably observable. Since the scan period is 100 ms, edge traversal must take more than 200 ms, which constrains the maximum safe speed to 0.255 m/0.2 s = 1.275 m/s. Exceeding this limit means the robot can cross an entire grid cell between two consecutive scans, making its position unobservable at cell-level granularity and invalidating the discrete model upon which the formal verification relies. The working cases (C1a, C1b, C1c) all respect this bound, which is why the model checking results carry over faithfully to the ROS2 simulations.

What varies is how individual robot speeds and wait times are assigned across the five robots. This variation in individual robot behaviour contributes to the success of these cases by naturally staggering

Table 4
The results for ROS execution and Timed Rebeca model checking for the cases which works properly and satisfied the correctness properties.

Parameters for different cases						
	Case C1a		Case C1b		Case C1c	
Scan rate	100		100		100	
Speed limit	1.275		1.275		1.275	
Stop zone	0.3		0.3		0.3	
Parameters for individual robots						
Robots	**Speed**	**Wait**	**Speed**	**Wait**	**Speed**	**Wait**
R1	0.5	1500	0.3	1500	0.4	1500
R2	0.7	2000	0.3	2000	0.5	2000
R3	0.8	2500	0.3	2500	0.7	2500
R4	0.5	3000	0.3	3000	0.8	3000
R5	0.3	1500	0.7	1500	0.9	1500
Model checking results						
Analysis result	Satisfied		Satisfied		Satisfied	
States	19264		20554		15747	
Transitions	43814		46868		35642	
ROS2 simulation results						
Simulations	5		5		5	
Simulation results	5/5 passed, 0/5 failed		5/5 passed, 0/5 failed		5/5 passed, 0/5 failed	

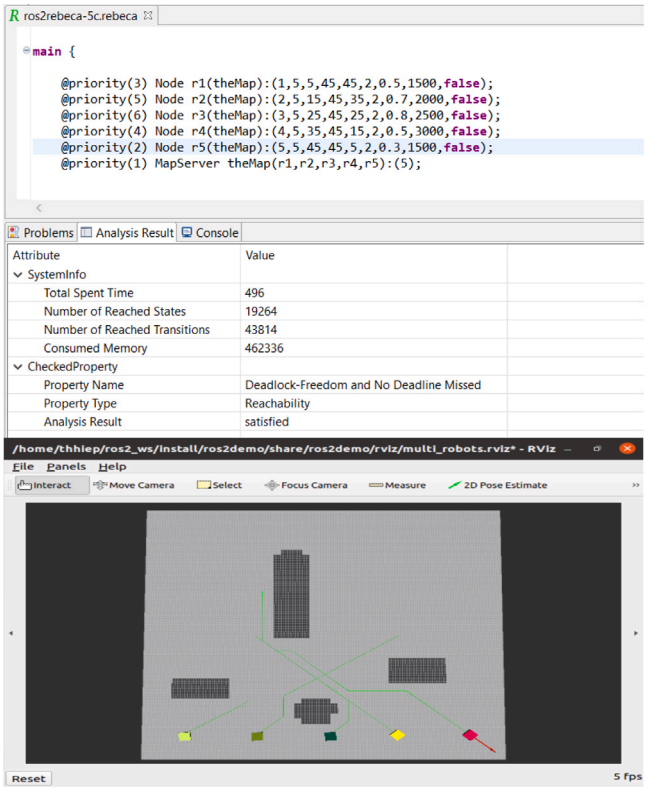

Fig. 10. Screenshots of Afra results (up) and ROS2 simulation results (down) for a working case. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

movements and reducing the likelihood of simultaneous intersection conflicts. Model checking returns **Satisfied** in all three cases after exhaustively exploring their respective state spaces. The differences in state/transition counts reflect varying degrees of non-determinism — more varied speed combinations produce more interleavings for the checker to explore. All 5/5 ROS2 simulations then confirm what the model proves: all robots reach their destinations without problems.

Table 5
Non-working cases.

Parameters for different cases						
	Case C2a		Case C2b		Case C2c	
Scan rate	140		100		140	
Speed limit	0.91		1.275		0.91	
Stop zone	0.3		0.3		0.3	
Parameters for individual robots						
Robots	**Speed**	**Wait**	**Speed**	**Wait**	**Speed**	**Wait**
R1	0.9	1500	0.5	1500	0.9	1500
R2	0.9	2000	0.5	1500	0.8	2000
R3	0.9	2500	0.5	1500	0.7	2500
R4	0.9	3000	0.5	1500	0.6	3000
R5	0.9	1500	0.5	1500	1.0	1500
Model checking results						
Analysis result	Assertion failed (collision)		Assertion failed (collision)		Assertion failed (collision)	
States	3074		6816		1740	
Transitions	6602		15293		3727	
ROS2 simulation results						
Simulations	5		5		5	
Simulation results	2/5 passed, 3/5 failed		3/5 passed, 2/5 failed		0/5 passed, 5/5 failed	

Fig. 10 illustrates Case C1a as a representative of working cases. The upper panel shows the Timed Rebeca model instantiation, where five robots are assigned distinct priorities, speeds, and wait times. The middle panel shows the model checker output: after exploring 19,264 states and 43,814 transitions in 496 ms, deadlock freedom, no missed deadline, and reachability properties are verified as satisfied. The lower panel shows the corresponding ROS2 visualization in RViz2, where the green lines trace the A*-computed paths of the five robots navigating around obstacles toward their respective goals. The crossing paths represent the deliberately congested scenario, yet all robots successfully reach their targets without collision or deadlock, consistent with the model checking result.

Non-working cases (Cases C2a, C2b, C2c). Table 5 provides the statistical and experimental parameters and results for cases C2a, C2b, C2c. Each of the three cases violates the safe operating constraints in a distinct way (speeds that violate the scan-rate constraint, short stop distance, and uniform speed parameters across all robots), which explains both why the model checker fails and why the ROS2 failure rates differ across cases.

C2a uses a scan rate of 140 ms with all five robots running at the same speed of 0.9 m/s. Applying the Nyquist-derived constraint: the maximum safe speed is $0.255/(2 \times 0.14) = 0.91$ m/s — so 0.9 m/s is right at the boundary. However, the uniform speed across all robots means there is no natural staggering: robots tend to arrive at intersection points simultaneously, overwhelming the conflict resolution. The model checker finds a collision after only 3074 states, and 3/5 ROS2 simulations fail.

C2b keeps the scan rate at 100 ms and speed limit at 1.275 m/s — identical to the working cases globally — but assigns all robots the same speed (0.5 m/s) and crucially the same wait time (1500 ms). This complete uniformity in both speed and wait time creates a worst-case scenario: robots move in lockstep, systematically converging on intersections together. The model checker needs slightly more exploration (6816 states) before finding the collision, reflecting that the failure is less immediate than in C2a. 2/5 simulations fail in ROS2.

C2c is the most aggressive case: scan rate 140 ms, speed limit 0.91 m/s, with robot speeds ranging from 0.6 to 1.0 m/s. R5 at 1.0 m/s exceeds the Nyquist-derived safe limit of 0.91 m/s, and even the lower-speed robots (0.6–0.9 m/s) are operating at or near the boundary given the slower scan rate of 140 ms. The combination of a degraded scan rate and a wide speed spread means that faster robots outrun their sensing capability while slower robots still create congestion, producing

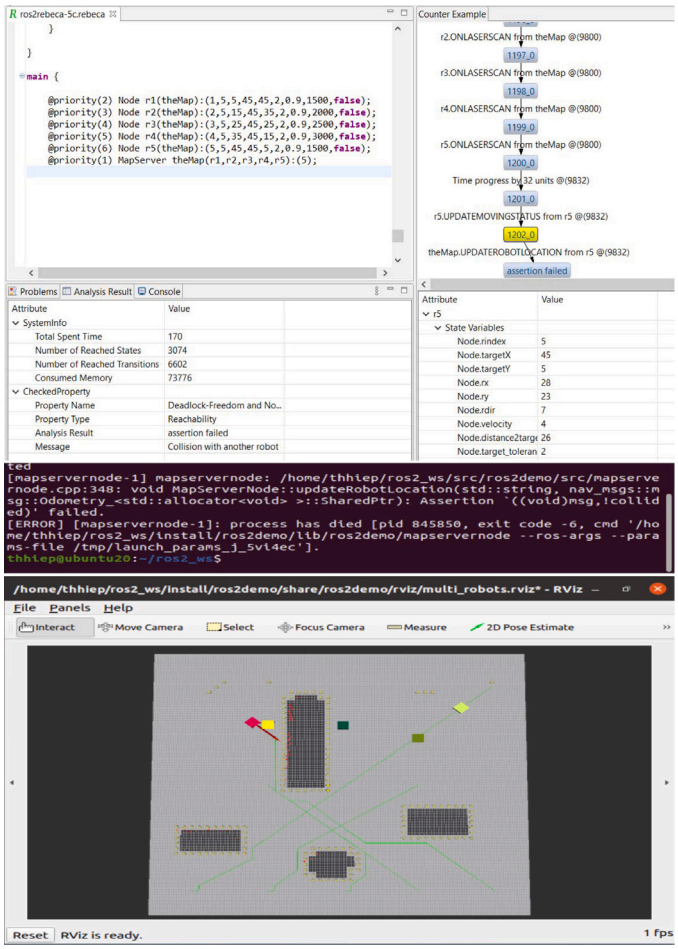

Fig. 11. Screenshots of Afra results (up) and ROS2 simulation results (down) for a non-working case. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

conflicts across the entire robot team rather than just at one robot. This explains why C2c yields the fewest states (1740) before a counterexample is found and the worst ROS2 outcome — 0/5 simulations pass — as the failure conditions are pervasive across all robots rather than dependent on a specific unlucky timing.

The state counts in the non-working cases (1740–6816) are dramatically lower than in the working cases (15,747–20,554). This is because the model checker terminates as soon as it finds a counterexample. A bad configuration leads to a violation quickly, so only a small portion of the state space is explored before the search stops. In contrast, a safe configuration requires exhaustive exploration of the entire reachable state space before correctness can be declared. The low state counts in C2a–C2c are therefore a direct indicator of how early and how easily the system fails. The ROS2 simulations reproduce the same failures, confirming that the model's predictions are faithfully transferred to the real system.

Even in the non-working cases, some ROS2 simulations pass (2/5 in C2a, 3/5 in C2b). It reflects the non-deterministic nature of concurrent execution. The model checker proves that a collision is reachable, not that it is inevitable in every run. Whether a collision actually manifests depends on the precise timing of scheduler decisions at runtime. C2c's 0/5 pass rate shows that when the Nyquist analogous constraint (Eq. (1)) is directly violated, the failure becomes effectively unavoidable.

Fig. 11 illustrates a representative non-working case. The counterexample trace (upper-right panel) shows the exact sequence of events leading to the collision: laser scan messages arriving at timestamps 9800–9832 ms, followed by R5 updating its moving status and then reporting its location to the map server, at which point the assertion fails with the message "Collision with another robot". The state variables of R5 at the moment of failure are also shown — position ($rx = 28$, $ry = 23$), direction 7, velocity 4, and distance to target 26 — pinpointing precisely where and when the collision occurs in the model. The terminal output (middle-bottom panel) confirms the same failure in ROS2: the map server node crashes with an assertion error — "Collision with another robot" — and the process exits with code -6, indicating an abnormal termination triggered by the same condition the model predicted.

The RViz2 visualization (bottom panel) shows the physical consequence: two robots (red and yellow) have converged on the same location near an obstacle, with their overlapping positions visible in the top-left area of the map. The frame rate has dropped to 1 fps, reflecting the system crash, compared to 5 fps in the working case.

9. Discussion, limitations, and future work

Our experimental results confirm that model checking accurately predicts the actual behaviour of the system. A multi-robot system implemented as a concurrent multi-agent system cannot be fully verified through traditional simulation or real deployment alone — concurrency and nondeterminism mean that not all scenarios can be verified through case-by-case testing. Model-checking addresses this by systematically exploring the entire reachable state space.

However, model checking introduces its own challenges: the system must be expressed as a discrete, abstracted model to remain tractable. For an autonomous multi-robot system with continuous physical behaviours, this raises two fundamental questions. First, how to model the system and the interactions discretely and compactly while still correctly capturing the relevant behaviours? Second, how to ensure that the model faithfully represents the real system so that conclusions drawn on the model are also valid on the real system in practice? These are the questions that motivated our work. Some prior work did not adequately address this discrete–continuous gap, resorting to integer-only variables rather than properly handling real-valued quantities.

Robotic systems are complex by nature, even basic multi-robot systems already exhibit significant system-level complexity. Oversimplified models risk losing essential domain information. In particular, many existing works neglect geometric and kinematic features such as physical dimensions, speed, and motion constraints. The central challenge is therefore to capture the necessary physical complexity while retaining the simplicity and tractability required for model checking.

Question 1: How do you model a robotic system discretely and compactly while still correctly capturing the relevant behaviours?

The key is to properly account for the physical constraints of the system rather than relying on naive or pattern-based randomization. Robotic systems are constrained by kinematic features, geometry, and physics laws — if these are not incorporated, the model generates infeasible scenarios that cause unbounded state space explosion. In this work, the discretization strategy explicitly handles the discrete–continuous gap through the Nyquist-derived speed constraints and grid-based abstractions, ensuring that the discrete model only explores physically realizable states. The model is intentionally kept as a baseline that captures movement, pathfinding, obstacle management, and robot interaction at a level of abstraction that generalizes beyond the specific robot platform.

Question 2: How do you know the model faithfully represents the real system?

The work addresses this through two complementary models serving distinct purposes. An initial prototype model is created according to the workflow diagram in Fig. 6 prior to the code, to test architectural design and processing logic, discovering parameters that influence the behaviours of the system, anticipating potential issues to cope with, and

experimenting behavioural algorithms. A separate verification model is then derived after the code is written, reflecting what has been implemented. The alignment between the verification model and the code is confirmed by the experimental results themselves: correctness properties verified in the verification model are consistently matched by the ROS2 simulation outcomes — violations predicted by model checking reproduce as actual failures, and configurations verified as safe pass all simulation runs. While simulation produces fluctuating results due to concurrency and nondeterminism, model checking gives a consistent yes or no answer, making the correspondence between model and system unambiguous and repeatable.

Note that verifying distinct properties may require deriving separate verification models at varying levels of abstraction. A model A is an abstraction of model B if both represent the same system but A has less detail. A is a sound abstraction if every property of interest that holds in A also holds in B — equivalently, B is a refinement of A that adds detail without violating properties established by A. A is a complete abstraction if every property of interest that holds in B also holds in A. Both soundness and completeness are always defined with respect to the specific properties of interest, not the model as a whole. In this work, we use Timed Rebeca for developing both the initial prototype model and the verification model for the purpose of verifying reachability, deadlock freedom, and collision freedom with the abstractions described in Sections 5 and 6, reducing the discrete–continuous gap. This approach integrates prototyping and verification within a unified framework.

The faithfulness of the Timed Rebeca model to the ROS2 implementation rests on three explicit geometric and temporal conditions. First, the robot's position is represented by its frame origin (at its centre), which is mapped to the centre of the nearest grid cell as the robot moves between cells. The robot's physical dimensions are accounted for by projecting its footprint into the continuous world when checking for collisions. Eq. (2) ensures that this footprint projection is fine-grained enough that any collision occurring during travel between two adjacent cells in the continuous world is guaranteed to be detected in the discrete model — no collision can be missed by the cell-level abstraction. Second, the stop zone must exceed the edge length (0.3 m > 0.255 m), ensuring a robot is never simultaneously in two cells. Third, the speed must satisfy the Nyquist-derived constraint (Eq. (1)), ensuring the robot's position is observable at cell-level granularity at every scan instant. When these three conditions hold, every concrete transition in the ROS2 system has a corresponding transition in the discrete model, and the safety properties — collision freedom and deadlock freedom — are preserved across the abstraction. The experimental results in Section 8 provide empirical confirmation: all property violations found in the model reproduced as failures in ROS2, and all verified-safe configurations passed all simulation runs.

Limitations. There are several limitations in this work:

- The codes of Timed Rebeca model and ROS2 implementation are manually constructed and synchronized. It is not straightforward and error-prone to map from a discretely abstracted model to real-world implementation code with many details, and vice versa. They all need to be derived from a common higher-level model for semantic conformance and synchronization.
- Semantic preservation was verified manually by selective spot tests with representative cases in safe or unsafe areas of the parameter space and manual comparison of traces and outcomes.
- ***Scalability.*** Our experiments can handle 5 robots in a 50 × 50 grid, which is already a larger system compared to others' work (e.g. by Gu et al., 2018 which can only handle one robot with one simple moving obstacle). The 50 × 50 grid size was chosen to set a limit on the worst-case path length (which can be 50 × 50) and to remain manageable to Rebeca model checker. In order to check the scalability, we obtained verification results by running Afra on a grid of 100 × 100 and 10 robots keeping their path contained in a 50 × 50 grid. The experiment was conducted on an HP Pavilion+ laptop (Core i7, Windows 11) equipped with a 477 GB hard disk and 16 GB of RAM. Afra generated 434,615 states and 2,151,259 transitions. If we change the configuration by increasing the length of the path or the number of robots, the model-checking process continues until all available memory is exhausted.

  The bottleneck for state explosion is the storage of the full generated path in the state space, whereas the path length on an $m \times m$ grid in worst case can be $m \times m$. The routing algorithm and data structures can be adjusted to tackle the state space explosion problem.
- ***Resolution vs. accuracy.*** The relation between the resolution and accuracy of the discrete model is captured by Eq. (2) through analysis and experimentation, not fully empirically evaluated. Through experimentation, we detected that the model starts to behave correctly when Eq. (2) is satisfied. If the resolution is reduced beyond that limit, that means the model is too coarse/sparse, it will result in uncaptured important moments, as stated. Conversely, if the resolution is increased, meaning the step size is thinner than necessary, by analysis, Eq. (2) is still respected, safety properties are still preserved as before, resulting in no gain in accuracy — that is why we did not go further than that limit. Doubling resolution means halving the step size, thus doubling the grid dimensions, which is equal to the problem of scalability assessment.

Future work. Currently, both the Timed Rebeca model and the ROS2 implementation are manually constructed and semantically synchronized, with abstraction soundness verified through manual spot checks. A key future direction is to introduce a continuous model as a higher-level representation from which both the discrete model-checking model and the implementation code are generated automatically, with semantic preservation formally proved as part of the transformation process. This would simultaneously eliminate manual synchronization effort, shield developers from discretization complexity, and replace selective manual validation with systematic, provable soundness guarantees across the entire parameter space.

A complementary direction is the design of a unified language that encompasses both discrete and continuous aspects within a single framework, enabling automated generation of verification models and ROS2 code from a common specification. This requires formalizing the mapping rules between modelling and implementation constructs, enhancing the discretization strategies developed in this work, and establishing quantitative metrics for evaluating any remaining model–system gaps.

10. Related work

This section reviews existing works on modelling and formal verification of robotic systems, with a particular focus on ROS-based and multi-robot systems, to contextualize and substantiate the contributions of this work.

Halder et al. (2017) used Timed Automata and UPPAAL to model and detect communication issues in a ROS pub-sub channel, focusing on queue overflow and message interruptions from higher priority publishers. The analysis was demonstrated on the Kobuki robot, verifying only concurrency and timing issues in a communication thread, without addressing robotic behaviours.

Gu et al. (2018) used Timed Automata to model the control system of an autonomous wheel loader, including algorithms for path planning (A*) and collision avoidance (Dipole Flow Field), and used UPPAAL to encode and model-check against several requirements via TCTL logic. The verification properties include the following: compute collision-free paths given a dummy mobile obstacle (moving along a preset path), switch to stopping mode upon malfunction, and complete a

cruise within a time limit. The solution works with a single vehicle, moving objects are treated as dimensionless dots, LIDAR data is skipped, and dynamic path planning has issues with arbitrarily moving obstacles. Difficulty in using integer variables in UPPAAL to model robotic behaviours was mentioned but not addressed, and no comparison with actual system behaviours was provided.

Drona (Desai et al., 2017) is a framework for reliable planning of collision-free 3D flying paths for a swarm of drones with consideration of local clock deviations. Drona uses the P language for modelling, then map the code in P to a deployable ROS application in C language. The P code of Drona can be verified using the Zing model checker for P programs. The multi-robot motion planner in Drona can compute timed trajectories for all drones so that they fly coordinatedly in time without collision. Drona specifically works with connected drones, assuming each fits one grid cell. Our solution works for dimensional ground robots and arbitrary moving obstacles dynamically detected by a LIDAR sensor, and each robot behaves independently and adaptively up to its own situation.

In Miyazawa et al. (2019) and Li et al. (2024), the authors present RoboChart, a domain-specific modelling language based on UML state machine diagram supported by the RoboTool environment built on the Eclipse Modelling Framework/Ecore to model robot control software. The toolchain generates both deployable code and mathematical models, enabling automated verification of correctness properties expressed in a natural language like assertion syntax. The formal foundation is discrete-timed CSP, with FDR as the accompanied model checker. The consistency between the program and the model is verified by recording the execution trace in the program and feed it back to the model checker to verify if the model passes it as well. The generated robot code is a custom C++ framework rather than ROS. Physical aspects concerning the robot and its interactions with the environment are not modelled (Li et al., 2024). The approach focuses on software modelling only. It is not clear how to map ROS2 system building blocks like the map data, the nodes and their interactions, simulated sensory data, planning algorithms and movement/action trajectories to the RoboChart language and tooling. Modelling of continuous behaviours is left for a future work.

Mohammed et al. (2010) used hybrid automata and hybrid state machine to specify and model-check a rescue scenario in a multi-robot system called RoboCup. However, as experienced with Hybrid Rebeca and hybrid automata (Jahandideh et al., 2019), it is not straightforward to express system architecture using hybrid automata. The analysis also suffers from usability and scalability.

In Nigam and Talcott (2023), Nigam and Talcott present automated methods to prove safety properties for cyber physical systems, which involves steps of reading sensors and sampling time between observations. The adequacy of sampling time step is addressed so that nothing important is missed between steps. It is a fixed rate sampling approach which is known to suffer from rounding-off and performance problems (the rate at some point cannot get smaller because pseudo-real number representation in computers has limited precision, and the number of iterations is huge). The way we model laser scans is analogous to their approach (scanning step is fixed to 2°), while the way we model the movements can be considered as discrete-event simulation, which captures representative points and models the time advancement to the next point. In any case, the step length in terms of time or distance has to be small enough to avoid missing adversarial events that may happen between two points.

Balabonski et al. (2019) study formal models of mobile robot movement under different assumptions about time and synchronization: (i) robots operating in a continuous 2D Euclidean space, moving in continuous time with asynchronous progress (continuous models), and (ii) robots abstracted to move on discrete structures (e.g., graphs) with discrete asynchronous steps (discrete models). The continuous model captures physical reality (real positions and motions) while the discrete model captures the digital nature of many algorithmic robot protocols — where moves are abstracted as atomic asynchronous steps. The work uses the Pactole framework and Coq proof assistant to specify the two theoretical models and prove that any execution in one model can be simulated in the other, thus certifying that a discrete reasoning framework is safely applicable to continuous real robots. Their work uses abstract mathematical models of asynchronous robot movements and provides formal reasoning about equivalence between continuous and discrete models. Dal Zilio et al. (2023) proposed a framework that generates executable models from formal specifications, enabling runtime monitoring and verification while preserving semantic consistency between specification and execution. Their framework allows properties such as schedulability, timing constraints, and mutual exclusion to be verified offline using model-checking techniques while also supporting runtime safety mechanisms. Praveen et al. (2022) proposed a design and formal verification framework to model multi-agent protocols, focusing on a distributed drone scenario. Their approach represents the system as networks of timed automata in UPPAAL and translates the verified models into ROS-based simulations to enable early validation before deployment. Model-driven approaches have also been explored for analysing human–robot interaction scenarios. In particular, prior work (Lestingi et al., 2022) extends a model-driven development framework with statistical model checking to estimate mission success probabilities in multi-robot service scenarios, incorporating richer models of human behaviour and task handover among robots in complex healthcare environments. In contrast, our work is a more engineering-oriented, pragmatic approach, addressing the bridging between a realistic ROS2-based multi-robot system and a discrete system model which is based on the actor model for asynchronous interactions and uses timing constructs in Timed Rebeca to approximate continuous physical behaviour.

11. Conclusion

Robotic systems are inherently complex and span multiple disciplines. As these systems grow in size and sophistication, modelling them prior to implementation becomes both intuitive and highly beneficial. Model-based development and verification enable early-stage prototyping and more efficient problem resolution. In this work, we have demonstrated a concrete methodology for co-designing and verifying multi-robot systems using Timed Rebeca, whose asynchronous, message-based execution model and explicit time semantics make it naturally suited for representing ROS2 node topologies and performing model-checking to validate essential system properties.

Realizing this advantage entails systematically handling the intricate aspects of robotics: from bridging the gap between the continuous nature of physical-world dynamics and the discrete abstractions in formal models, to resolving discrepancies between real-world programming environments and modelling languages. This work has successfully tackled these challenges by constructing a precise, efficient model of a multi-robot system and using that model to guide ROS2 software development. Three critical system properties were verified — target reachability, collision freedom, and deadlock freedom — with experimental results confirming that property violations detected in the model consistently reproduce as failures in the ROS2 implementation, and that configurations verified as safe pass all simulation runs.

Thus, the central technical contribution is a principled strategy for bridging the discrete–continuous gap — translating continuous robot dynamics into discrete models that are compact enough for model checking yet faithful enough that verified properties carry over to the real implementation.

CRediT authorship contribution statement

Hiep Hong Trinh: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Conceptualization. **Marjan Sirjani:** Writing – review & editing, Supervision, Conceptualization. **Federico Ciccozzi:** Writing – review & editing, Supervision, Funding acquisition. **Abu Naser Masud:** Writing – review & editing, Supervision. **Mikael Sjödin:** Supervision.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Hiep Hong Trinh reports financial support was provided by Vetenskapsrådet. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work was partially funded by the Swedish Research Council through the ORPHEUS project (rn. 2022-03408).

Data availability

No data was used for the research described in the article.

References

Agha, G., 1986. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA.

Balabonski, T., Courtieu, P., Pelle, R., Rieg, L., Tixeuil, S., Urbain, X., 2019. Continuous vs. discrete asynchronous moves: A certified approach for mobile robots. In: Atig, M.F., Schwarzmann, A.A. (Eds.), Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers. In: Lecture Notes in Computer Science, vol. 11704, Springer, pp. 93–109. http://dx.doi.org/10.1007/978-3-030-31277-0_7.

Berlin, F., Granath, S., 2021. Obstacle detection and avoidance for an automated guided vehicle. URL https://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-177709.

Brown, A., 2004. Model driven architecture: Principles and practice. Softw. Syst. Model. 3, 314–327. http://dx.doi.org/10.1007/s10270-004-0061-2.

Clarke, E.M., Grumberg, O., Kroening, D., Peled, D.A., Veith, H., 2018. Model Checking, second ed. MIT Press, URL https://mitpress.mit.edu/books/model-checking-second-edition.

Dal Zilio, S., Hladik, P.-E., Ingrand, F., Mallet, A., 2023. A formal toolchain for offline and run-time verification of robotic systems. Robot. Auton. Syst. 159, 104301. http://dx.doi.org/10.1016/j.robot.2022.104301, URL https://www.sciencedirect.com/science/article/pii/S0921889022001907.

Desai, A., Saha, I., Yang, J., Qadeer, S., Seshia, S.A., 2017. DRONA: A framework for safe distributed mobile robotics. In: 2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems. ICCPS, pp. 239–248.

Fishwick, P.A. (Ed.), 2007. Handbook of Dynamic System Modeling - Chapter 11. Chapman and Hall/CRC, http://dx.doi.org/10.1201/9781420010855.

Foughali, M., Zuepke, A., 2022. Formal verification of real-time autonomous robots: An interdisciplinary approach. Front. Robot. AI 9, 791757. http://dx.doi.org/10.3389/FROBT.2022.791757.

Gu, R., Marinescu, R., Seceleanu, C., Lundqvist, K., 2018. Formal verification of an autonomous wheel loader by model checking. FormaliSE '18, Association for Computing Machinery, New York, NY, USA, pp. 74–83. http://dx.doi.org/10.1145/3193992.3193999.

Halder, R., Proença, J., Macedo, N., Santos, A., 2017. Formal verification of ROS-based robotic applications using timed-automata. In: Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering. FormaliSE '17, IEEE Press, pp. 44–50.

Hart, P.E., Nilsson, N.J., Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. 4 (2), 100–107. http://dx.doi.org/10.1109/TSSC.1968.300136.

Jahandideh, I., Ghassemi, F., Sirjani, M., 2019. Hybrid Rebeca: Modeling and analyzing of cyber-physical systems. pp. 3–27. http://dx.doi.org/10.1007/978-3-030-23703-5_1.

Khamespanah, E., 2018. Modeling, Verification, and Analysis of Timed Actor-Based Models (Ph.D. thesis). Reykjavik University, Reykjavik, Iceland, https://hdl.handle.net/20.500.11815/1185.

Khamespanah, E., Sabahi Kaviani, Z., Khosravi, R., Sirjani, M., Izadi, M.-J., 2012. Timed-rebeca schedulability and deadlock-freedom analysis using floating-time transition system. In: Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions. In: AGERE! 2012, Association for Computing Machinery, New York, NY, USA, pp. 23–34. http://dx.doi.org/10.1145/2414639.2414645.

Khamespanah, E., Sirjani, M., Khosravi, R., 2023. Afra: An eclipse-based tool with extensible architecture for modeling and model checking of Rebeca family models. In: Hojjat, H., Ábrahám, E. (Eds.), Fundamentals of Software Engineering - 10th International Conference, FSEN 2023, Tehran, Iran, May 4-5, 2023, Revised Selected Papers. In: Lecture Notes in Computer Science, vol. 14155, Springer, pp. 72–87. http://dx.doi.org/10.1007/978-3-031-42441-0_6.

Lestingi, L., Sbrolli, C., Scarmozzino, P., Romeo, G., Bersani, M.M., Rossi, M., 2022. Formal modeling and verification of multi-robot interactive scenarios in service settings. In: Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering. FormaliSE '22, Association for Computing Machinery, New York, NY, USA, pp. 80–90. http://dx.doi.org/10.1145/3524482.3527653.

Li, W., Ribeiro, P., Miyazawa, A., Redpath, R., Cavalcanti, A., Alden, K., Woodcock, J., Timmis, J., 2024. Formal design, verification and implementation of robotic controller software via RoboChart and RoboTool. Auton. Robots 48 (6), http://dx.doi.org/10.1007/s10514-024-10163-7.

Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M., 2019. Formal specification and verification of autonomous robotic systems: A survey. ACM Comput. Surv. 52 (5), http://dx.doi.org/10.1145/3342355.

Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J., 2019. RoboChart: modelling and verification of the functional behaviour of robotic applications. Softw. Syst. Model. 18 (5), 3097–3149. http://dx.doi.org/10.1007/s10270-018-00710-z.

Mohammed, A., Furbach, U., Stolzenburg, F., 2010. Multi-robot systems: Modeling, specification, and model checking. http://dx.doi.org/10.5772/7349.

Nigam, V., Talcott, C., 2023. Automating recoverability proofs for cyber-physical systems with runtime assurance architectures. In: David, C., Sun, M. (Eds.), Theoretical Aspects of Software Engineering. Springer Nature Switzerland, Cham, pp. 1–19.

Open Robotics, 2023a. ROS2 concepts. URL https://docs.ros.org/en/foxy/Concepts.html.

Open Robotics, 2023b. ROS2 online documentation. URL https://docs.ros.org/en/foxy/index.html.

Open Robotics, 2023c. Understanding ROS2 nodes. URL https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html.

OSRF, 2023. ROS concepts and design patterns. URL https://osrf.github.io/ros2multirobotbook/ros2_design_patterns.html.

Praveen, A.T., Gupta, A., Bhattacharyya, S., Muthalagu, R., 2022. Assuring behavior of multirobot autonomous systems with translation from formal verification to ROS simulation. IEEE Syst. J. 16 (3), 5092–5100. http://dx.doi.org/10.1109/JSYST.2022.3149677.

Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingólfsdóttir, A., Sigurdarson, S.H., 2014. Modelling and simulation of asynchronous real-time systems using Timed Rebeca. Sci. Comput. Program. 89, 41–68, URL https://doi.org/10.1016/j.scico.2014.01.008.

Shannon, C., 1949. Communication in the presence of noise. Proc. IRE 37 (1), 10–21. http://dx.doi.org/10.1109/JRPROC.1949.232969.

Sirjani, M., 2006. Rebeca: Theory, applications, and tools. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (Eds.), Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, the Netherlands, November 7-10, 2006, Revised Lectures. In: Lecture Notes in Computer Science, vol. 4709, Springer, pp. 102–126. http://dx.doi.org/10.1007/978-3-540-74792-5_5.

Sirjani, M., 2018. Power is overrated, go for friendliness! expressiveness, faithfulness, and usability in modeling: The actor experience. In: Principles of Modeling - Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday. In: Lecture Notes in Computer Science, vol. 10760, Springer, pp. 423–448, URL https://doi.org/10.1007/978-3-319-95246-8_25.

Sirjani, M., Khamespanah, E., 2023. Timed actors and their formal verification. In: Mezzina, C.A., Caltais, G. (Eds.), Procs of EXPRESS/SOS 2023. In: EPTCS, vol. 387, pp. 1–7. http://dx.doi.org/10.4204/EPTCS.387.1.

Sirjani, M., Lee, E.A., Khamespanah, E., 2020. Model checking software in cyberphysical systems. In: 44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020. IEEE, pp. 1017–1026, URL https://doi.org/10.1109/COMPSAC48688.2020.0-138.

Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S., 2004. Modeling and verification of reactive systems using Rebeca. Fund. Inform. 63 (4), 385–410, URL http://content.iospress.com/articles/fundamenta-informaticae/fi63-4-05.

Street, C., Mansouri, M., Lacerda, B., 2023. Formal modelling for multi-robot systems under uncertainty. Curr. Robot. Rep. 4, 1–10. http://dx.doi.org/10.1007/s43154-023-00104-0.

Trinh, H.H., 2023. Model Based Development and Verification of ROS2 Robotic Applications using Timed Rebeca (Master Thesis). Malardalen University.

Wikipedia, 2023. A* search algorithm. URL https://en.wikipedia.org/wiki/A*_search_algorithm.

Yalcin, O., Sayar, A., Arar, O., Akpinar, S., Kosunalp, S., 2013. Approaches of road boundary and obstacle detection using LIDAR. In: IFAC Workshop on Advances in Control and Automation Theory for Transportation Applications.