An Extended BDI Case Study Toward Campus Mail Delivery

Bardia Parmoun[1][0009−0001−4952−7343] and Babak Esfandiari[1][0000−0002−4441−3435]

Carleton University, Ottawa, Ontario, Canada
bardiaparmoun@sce.carleton.ca, babak@sce.carleton.ca

Abstract. This paper presents a complex robotic case study for using BDI to program a mobile robot to navigate a tunnel environment. The objective is to travel through the tunnel network of Carleton University and deliver mail between buildings. The proposed solution uses an iRobot CREATE robot programmed in Jason, supported by LiDAR sensing and a network of Bluetooth beacons for localization. Building on prior work that limited the robot to line-following behaviour in a simplified environment, this approach extended to full navigation of the university tunnels. Additionally, a realistic simulation environment has been presented to support testing and iterative development. This work demonstrates the feasibility of applying Jason to a real-world robotic task and provides insights into its performance and practical challenges.

Keywords: Belief-Desire-Intention (BDI) · AgentSpeak · Jason · Robot Operating System (ROS) · Gazebo Simulator · iRobot CREATE

1 Introduction

Robotic systems can often be described as following a recurring cycle of observation and environmental response to achieve an overall objective. This process can be mapped to the notion of autonomous agents [13]. A popular paradigm when designing such agents is the belief-desire-intention (BDI) model, which groups the different aspects of the agent into the three concepts of beliefs, desires, and intentions [10]. AgentSpeak [4] is a theoretical language for describing BDI systems, with Jason [1], offering a Java-based implementation that supports internal actions and multi-agent environments.

Applications of BDI, particularly in the field of robotics, are often limited in scope and predominantly confined to simulation. To address this gap, we present a complex robotic case study for using Jason and BDI, named Hermes, a physical mobile robot to navigate Carleton University's tunnel network. The work is also accompanied by a realistic ROS Gazebo simulator to serve as a tool for development and evaluation [1]. We assessed the feasibility of applying Jason to

[1] Video demos of the robot and the simulator are available at: https://youtube.com/playlist?list=PL5eOOOo-XDN2NaJcktxF5PdxjNZqxwHEO&si=vJbpiiR-p-sUOnDL

a real-world robotic problem, focusing on key challenges such as performance, managing conflicting goals, coordinating synchronous and asynchronous actions, and maintaining time synchronization across all components. Our results indicate that using BDI, specifically Jason, is a viable and effective approach for complex robotic applications.

1.1 Problem Description

Hermes is a continuation of [17], which introduced a Jason-based autonomous line-follower robot (iRobot CREATE 3 [12]), navigating a simple network of QR-marked intersections. The system used infrared (IR) sensors for line following and a camera for QR codes; it was inspired by the underground tunnel network at Carleton University, Ottawa. The tunnels are used by the students and staff, including for manual mail delivery via carts. This paper aims to replace these carts with autonomous robots navigating between buildings.

The tunnels create an interesting environment for robotic use cases. Internet connection is unreliable, and Global Positioning System (GPS) coverage is limited due to the tunnel structure. Furthermore, the campus spans different elevation levels with slopes leading to irregular hallways. These tunnels are also shared with students, staff, manually driven carts, and potential obstacles, so the robot should be able to avoid and handle possible collisions along its path.

Despite the difficulties described above, the tunnel environment also has certain attractive features. Its design is fixed, so the robot can be assumed to have full knowledge of the entire map. The tunnels are a closed environment, easy to model as a network of intersections and hallways.

Given the above, this paper presents Hermes as an autonomous robot that runs in the tunnels, capable of completing a trip between any two buildings with basic collision handling capabilities. The current work intentionally focuses on a single agent completing a single trip scenario to isolate the challenges associated with applying BDI to a real-time robotic system. Extending Hermes to multiple cooperating robots introduces additional challenges, such as task allocation, communication, and coordination, which are outside the scope of this work. Unlike prior work, Hermes relies on LiDAR for perception and Bluetooth beacons at each intersection to compensate for the lack of WiFi and GPS. It also uses the new iRobot CREATE 3 platform for its built-in ROS support and improved native behaviours like docking. Table 1 provides a summary of the contributions detailed in this work compared to [17]. While the hardware upgrades enable a larger operational scope, the main contribution of this work is addressing the practical challenges that expose limitations of applying BDI agents to real-time robotic systems. Some of these challenges were briefly introduced in [17]. This work investigates how a Jason-based agent can handle perception uncertainty, concurrent goals, asynchronous actions, and timing constraints in a realistic robotic environment; however, unlike [17], some practical constraints, such as battery life, were not addressed in this iteration.

Table 1. A comparative summary of the contributions of this paper compared to [17].

Area	**[17]**	**This Paper**
Scope	Relies on lines, performs basic navigation, and self-charges if needed.	Can navigate the actual tunnels; handles obstacles.
Hardware	IR sensors, Camera, iRobot CREATE 2 (fewer sensors and controlled input data).	LiDAR, Bluetooth beacons, iRobot CREATE 3 (fast-paced and noisy input data).
Testing Tools	Debugger to verify each node	Realistic Gazebo simulator
Practical Challenges: plan priorities, action handling, & synchronizing inputs	• Uses event/goal-based plans for priorities (not easily extensible). • Drops incoming actions while executing current ones (unreliable). • No central way for sensor inputs (hard to synchronize).	• Adopts the subsumption pattern for plan priorities. • Creates a service handler for sync./async. actions. • Adds a centralized way for handling all perceptions.

2 Background

2.1 AgentSpeak and Jason

The belief-desire-intention (BDI) model represents the agent's behaviour in terms of the three concepts of beliefs, desires, and intentions. At a given time, a BDI agent has specific knowledge (beliefs) from its environment; using these beliefs, the agent follows a set of plans (intentions) that help it achieve a goal (desire) [10]. AgentSpeak is a theoretical Prolog-like language that is used for specifying these plans [4]. Various interpreters have been developed for AgentSpeak. The interpreter that has been chosen for this paper is Jason [1].

Jason represents beliefs as predicates. It represents desires in the form of achievement and test goals. Achievement goals, prefixed with `!`, represent the desire to reach a specific predicate. Similarly, test goals, prefixed with `?`, represent the desire to check if a specific predicate is true. Jason also adds a notion of triggers, which represent the addition (prefixed with `+`) or removal (prefixed with `-`) of specific beliefs, achievement goals, or test goals [1]. Using these concepts, Jason's plans are then defined as: `trigger : context <- body`.

2.2 Robot Operating System (ROS), Gazebo, & RViz

The Robot Operating System (ROS) is a set of open-source packages and libraries that aid in the development of robotic systems. These ROS packages act as a middleware between the lower-level hardware components and the higher-level implementation of the robot's logic. A typical ROS architecture consists of a network of nodes. These nodes then communicate with each other via the message queues maintained by ROS [19].

Gazebo is a collection of libraries geared towards developing robotic simulations. It includes simulations for many common robotic sensors, such as LiDAR, Infrared (IR) sensors, etc. [18]. RViz [20] is a 3D visualization tool for ROS that displays real-time data about a robot's environment, state, and sensor inputs.

2.3 Related Work

There have been various attempts to use Jason and BDI for robotic applications. A notable example is [17], which is an initial attempt to solve the campus delivery problem. The author emphasizes the need to prioritize plans for handling the different goals. This idea inspired us to choose the subsumption model for designing the agent behaviour. Additionally, [17] identifies key challenges for using Jason in a real-time use case such as managing perceptions, handling actions, or defining goal and event-driven plans, which we have explored further.

Another important work for the use of Jason in robotic applications is [23]. This paper investigates the various challenges that exist when it comes to programming mobile robots and demonstrates how BDI can be used to address them. Similar to the previous research, [23] also argues that the higher-level behaviour of the robot can be expressed via BDI, which helps improve the system's overall readability and explainability. It proposes a framework named `JaCaROS` for connecting ROS and Jason and handling some of the common robotic behaviours such as odometry. The main objective of this framework is to abstract and refactor the granular details of the robot's operations, such as its odometry, into a separate framework.

A more recent application of Jason and BDI is [5]. The paper empirically evaluates the BDI paradigm in three different robotic applications that vary in difficulty. It uses the `ROS-A` framework to connect Jason to ROS. [5] also considers three different simulated scenarios. Similar to [23], it also compares the implementation of these agents against their purely Python counterparts and measures their execution and failure handling times. The author concludes that although Python offers marginally faster execution times, it is much slower on the failure handling front. Additionally, [5] analyzes the BDI paradigm qualitatively and highlights its strengths in terms of task switching, fault handling, and maintainability compared to imperative programming.

3 Design

The design of Hermes comprises two main components: hardware and software.

3.1 Hardware Design

iRobot CREATE 3 This is a ROS-compliant mobile robot. This device includes a variety of IR sensors and comes with a set of predefined behaviours for common use cases, such as turning, docking, and wall following. [12].

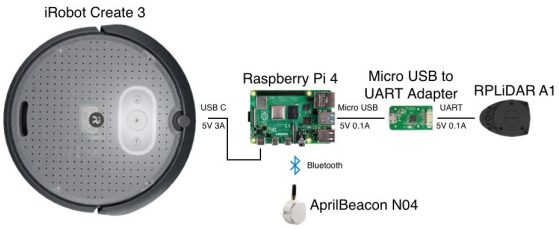
Raspberry Pi Hermes uses a Raspberry Pi 4 as its main processing unit. This is due to the Pi's many capabilities, including support for Bluetooth and Wi-Fi. It is also low-powered, allowing it to be powered directly by the robot [7].

LiDAR Sensor The robot required specialized sensors to measure its distance from its surroundings. This would allow it to achieve various tasks such as wall following and navigation. For this project, LiDAR sensors were used; specifically, RPLiDAR A1, due to its range in both distance and direction [21].

Bluetooth Beacons The Carleton University tunnels have very limited internet and GPS access. As such, we placed Bluetooth beacons at key locations to notify the robot of its approximate location. The project used April Beacon N04. These beacons are low profile, battery-powered, and have a range of $\sim 30m$ [6].

Hardware Schematic Figure 1 provides a summary of the hardware design:

Fig. 1. The hardware schematic of Hermes.



3.2 Software Design

Designing the Agent Behaviour As already outlined, the environment of this project was complex, and the various stimuli in it could create conflicting priorities for the agent. Different paradigms were considered when designing Hermes. One of the candidates was simple imperative programming involving a series of conditions. Another approach was the use of finite state machines (FSMs), a systematic approach for handling all input in a way that would take the relevant context into account. The subsumption architecture was a third plausible approach, which involves decomposing the robot's complex behaviour into a series of simple modules. These modules all behave independently of each other, produce separate behaviours based on the stimuli they receive, and can subsume one another based on their priorities [2].

By using the BDI paradigm, we came up with an approach that encompassed the three paradigms described above and properly handled conflicting priorities. The BDI design detailed in [17] broke down the problem into a set of small goals, which included: path following, navigation, mail delivery, and docking. It also introduced a notion of priority for the agent when it came to achieving these goals. [17] handled priority by moving the priority plans higher and binding them to events so they could be triggered at any time. We expanded on this

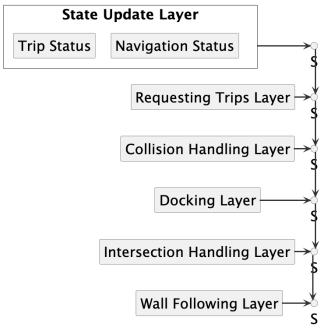
idea and mapped it to the subsumption architecture, with each module being in charge of achieving a specific goal. For the present implementation, the following sub-behaviours (goals) were considered, arranged in decreasing order of priority:

- **Requesting Trips:** requesting new trips for the robot.
- **Collision Handling:** handling possible collisions with obstacles.
- **Docking:** docking at the dock station when reaching a destination.
- **Intersection Handling:** detecting and performing the correct action at a given intersection (passing the intersection or doing a left, right, or U-turn).
- **Wall Following:** locating and maintaining a specific distance from the wall.

In this model, wall following was considered the lowest-priority goal since, with no other objective at hand, the robot had to focus on latching onto a wall and moving until it had obtained better information about its environment. Intersection handling subsumed wall following since it involved actively lifting the focus from the wall and performing a custom action at the intersection. Docking subsumed both intersection handling and wall following, since the robot had to prioritize its built-in docking behaviour when it reached the destination. Collision handling was the second important goal since the safety of the robot and the people in the tunnels was important. Finally, requesting trips was considered the highest priority goal since the robot should not move without an objective.

In addition, Hermes also needed to remember certain key information. In this case, when approaching a given intersection, the robot would receive a navigation instruction telling it how to react. This instruction would be sent to it only once; as such, the robot needed to remember it when it actually reached the intersection. Furthermore, it needed to remember that it had a trip to avoid moving aimlessly and wasting its battery. Since the general subsumption pattern does not contain a notion of central states, the process of updating and maintaining these central states was considered as a separate layer that subsumes every other layer. Figure 2 provides a summary of the agent's behaviour:

Fig. 2. An overview of the agent's overall behaviour following the subsumption model.



Connecting Jason to ROS Different methods exist for connecting Jason to ROS. We considered `JROS` [22], `Jason-ROS` [14], `Rason` [15], `Agent-in-a-Box` [9], `JaCaROS` [23], and `ROS-A` [3]. Although these were all great approaches for connecting Jason to ROS, none of them could be directly applied to the problem at hand. Due to the requirements set by the iRobot CREATE interface, the newer version of ROS (`ROS2`) was required. ROS2 has also become the standard for most robotic projects. In addition, having full control over the agent architecture class (`AgArch`) was desired for many reasons, like concurrently translating new perceptions to beliefs, controlling when the agent's actions were completed, and limiting any overhead involved in using a bridge to communicate with the system. Unfortunately, all of the existing approaches for connecting Jason to ROS only worked with ROS1 and did not offer full customization of the AgArch class. Porting these tools to ROS2 would have involved either relying on a ROS bridge, which was very inefficient, or directly compiling them for ROS2, which was quite difficult due to their reliance on ROS1-only libraries. Additionally, since most of these tools defined their own custom interface for AgArch, using them would add new constraints like expected message types or limited customization in action handling. We used `ros2-java` [8] to create direct Java applications in ROS2.

Simulator Design For the simulation aspect, Gazebo [18] was used due to its native support for ROS and its large library of realistic sensors, which also included the RPLiDAR sensor used in Hermes. iRobot has developed a detailed, realistic model of their iRobot CREATE 3 series for Gazebo [11]. We also created a realistic model of the AprilBeacon devices, using Gazebo's built-in radio frequency (RF) plugins. With Gazebo's built-in physical models, we simulated the tunnel environment. We also developed an RViz [20] model for the robot to monitor its behaviour and sensor inputs, both crucial for debugging purposes.

4 Implementation

A summary of Hermes hardware and software implementation is as follows [2]

4.1 Hardware Implementation

The USB-C port located in the cargo bay of the iRobot CREATE 3 robot was used to power the Raspberry Pi; it was secured using a custom base. This port was capable of outputting $15W$ of power, which was ideal for powering the Pi. The LiDAR sensor was mounted onto the robot and connected to the Pi through a Micro USB to UART connector. Figure 3 shows the final hardware setup.

4.2 Software Implementation

Implementing the Agent Architecture As described above, a ROS program is a network of nodes sending messages together via different topics. Figure 4

Fig. 3. An overview of the hardware setup.

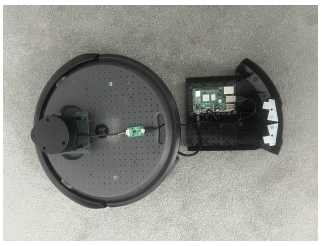
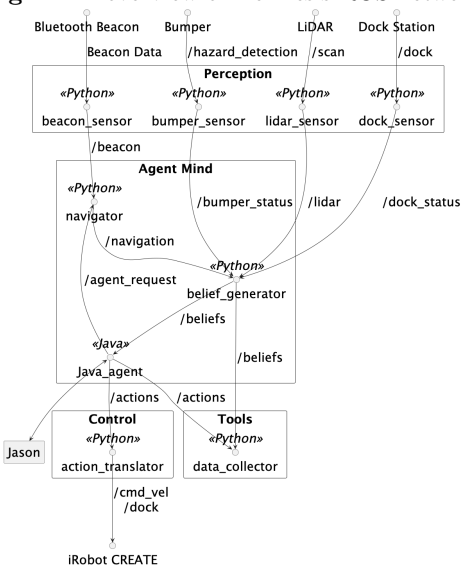


Fig. 4. An overview of Hermes's ROS network.



gives an overview of the ROS network in Hermes. As is evident, the ROS architecture for Hermes followed that of a typical agent, involving perception, reasoning, and action. At the perception layer, the nodes were in charge of observing the environment and capturing new information. Then, in the body of the agent, these perceptions were sent together to the Java node as beliefs. That node would send these beliefs to Jason and wait for actions. These actions were then eventually passed to the robot actuators to be executed. A data collection node was also used to probe the robot and collect necessary logs.

Implementing the Agent Behaviour The Jason portion of Hermes contains the majority of the agent's behaviour. The agent's perceptions are fed to Jason as beliefs. These perceptions are simplified at the ROS level to only contain the necessary information. These simplifications include analyzing the LiDAR

[2] The full implementation is available at: `https://github.com/NMAI-lab/hermes`

Fig. 5. An overview of the plans for the different layers in Hermes.

```
+bumperPressed: not(.intend(handleCollision)) <-
    .drop_all_intentions;
    !handleCollision.
+dockVisible: navigation(dock) & not(.intend(handleDocking)) &
↪  not(.intend(handleCollision)) <-
    .drop_all_intentions;
    !handleDocking.
+intersection(FDistance, LDistance, UDistance): navigation(...) &
↪  not(.intend(handleIntersection) & not(.intend(handleDocking)) &
↪  not(.intend(handleCollision)) <-
    .drop_all_intentions;
    !handleIntersection(...).
+facingWall(WallDistance, WallAngle): not(.intend(wallFollow) &
↪  not(.intend(handleIntersection) & not(.intend(handleDocking)) &
↪  not(.intend(handleCollision)) <-
    !wallFollow(WallDistance, WallAngle).
```

data to locate walls and intersections, identifying the closest Bluetooth beacon from its signal strength, and obtaining the correct navigation instruction using the robot's static map. The robot's configuration, i.e., its speed, appropriate wall following distance, etc., is also passed to Jason as beliefs. The agent then uses its plans to output the appropriate action. The highest-level Jason plans are a set of event-triggered plans that activate specific layers by adding them to the agent's desires queue. These plans must be trigger-based as opposed to being bound to specific desires since they can be activated at any time. Figure 5 provides an overview of the plans in charge of activating the different layers. As shown, these plans follow the design that was laid out in Figure 2, with each layer being activated by its appropriate trigger. The layers follow a pattern that aims to demonstrate subsumption. Each layer checks to see if the agent is executing any of the higher-priority plans. This is achieved by checking the agent's intentions. If not, the agent picks that layer and cancels the lower priority layers via `.drop_all_intentions`. It then proceeds to add the current layer to the agent's intention. These layers are defined in Jason as desire-based plans. Here is an overview of each layer:

- **State Update:** States are represented via beliefs in Jason. As a result, state updates refer to a group of plans that generate specific beliefs for the agent. This is the only layer that is not bound to any desire since it has the highest priority. This is because state updates should always be recorded. As shown in figure 6, the agent may receive a navigation instruction in the form of the belief `navigationInstruction`. It then uses that to maintain its own belief of `navigation`, which then gets used to handle intersections and docking. Similarly, the agent keeps track of its trip status via the `hasTrip` belief.

Fig. 6. The Jason plans for the state update layer.

```
-hasTrip: true <-
    !requestTrip.
+navigationInstruction(none): true <-
    -requestingTrip;
    !requestTrip.
+navigationInstruction(start): true <-
    -requestingTrip;
    +hasTrip.
+navigationInstruction(NavInstruction): hasTrip <-
    -+navigation(NavInstruction).
```

Fig. 7. The Jason plans for the request trip layer.

```
+!requestTrip: not(hasTrip) & not(requestingTrip) <-
    +requestingTrip;
    request_trip.
```

- **Requesting Trips:** Hermes is designed to run continuously and complete multiple trips. The trip requesting layer, shown in Figure 7, is used by the robot whenever a trip is completed to reach out to the client code and request a new one. This layer is also given its own desire of `requestTrip` and uses asynchronous action, `request_trip`, to ask the client for new trips.
- **Collision Handling:** Since the robot is equipped with a bumper sensor, collisions occur when it hits an obstacle head-on. These collisions are handled by backing up and doing a small turn. This behaviour is controlled via the `handleCollision` desire. Figure 8 provides an overview of this layer.
- **Docking:** The docking layer uses iRobot's built-in docking command. This behaviour, as detailed in Figure 9, is controlled via the `handleDocking` desire. The robot executes the `dock` command and completes the trip.
- **Intersection Handling:** At a given intersection, Hermes can do one of the following actions: go forward through the intersection or perform a right turn, left turn, or a U-turn. To simplify the implementation, the agent will first perform a turn, followed by a series of forward operations to go through the intersection. The Jason implementation for this behaviour is shown in Figure 10. Intersection handling is controlled via the `handleIntersection` intention. This intention takes in the robot's distance from the wall in all four directions as the input. It then uses the `navigation` belief to decide which plan to select. Each navigation plan has a `!performRepeatedTurns` intention to orient the robot at the correct angle and a `!performRepeatedForwards` intention to move the robot. The number of turns and forward moves is calculated based on distance, the robot's speed, and Jason's update rate.

Fig. 8. The Jason plans for the collision handling layer.

```
+!requestTrip: not(hasTri
+!handleCollision: wallFollowAimAngle(WALL_FOLLOW_AIM_ANGLE) <-
    !performRepeatedBackwards(...);
    !performRepeatedTurns(WALL_FOLLOW_AIM_ANGLE, ...).
```

Fig. 9. The Jason plans for the docking layer.

```
+!handleDocking: true <-
    dock;
    -hasTrip;
    -navigation(dock).
```

– **Wall Following:** Wall following entails a reactive approach to maintain a certain distance threshold from the wall. As shown in Figure 11, the robot calculates its current distance and angle with the wall and checks that against its threshold. If the robot is not within the threshold, it will adjust itself towards or away from the wall. As shown above, wall following is given its own intention of `!wallFollow`, which takes the robot's current distance and angle with the wall. The agent then uses its rules to see if the robot is too far or too close to the wall. Although more proven approaches, such as a PID controller, can be used to further fine-tune wall following, they were not considered at this time since we are mainly interested in focusing on Jason's capabilities and seeing if it could be feasible for time-sensitive tasks such as wall following. It is worth noting that, due to the predictable shape of the tunnels and Jason's quick reasoning time, this simple approach has already been very effective.

Simulator Implementation The simulator needed to be realistic to be used as a reliable tool for development. Firstly, iRobot's realistic RViz and Gazebo models were used to simulate the robot and the dock station. This model was further customized to include the RPLiDAR sensor mounted on it. A custom sphere-shaped model was created for AprilBeacons using Gazebo's radio frequency plugin with their exact specifications. This model incorporated Gaussian noise to increase realism. Additionally, a custom world generator was implemented, capable of creating a network of walls from a map of intersections using Gazebo's default wall model to generate custom maps. We also added the option of simulating obstacles of different sizes using Gazebo's default objects. To further increase the realism, the agent's configuration and code were kept identical on both the simulator and the robot. The simulator can also run headless for repeated experimentation. Figure 12 provides an overview of the simulator.

Fig. 10. The Jason plans for the intersection handling layer.

```
+!handleIntersection(FDistance, LDistance, UDistance, WallAngle): navigation(wall_follow) <-
    -navigation(wall_follow).
+!handleIntersection(FDistance, LDistance, UDistance, WallAngle): navigation(forward) <-
    !performRepeatedTurns(-1 * WallAngle, ...);
    !performRepeatedForwards(...);
    -navigation(forward).
+!handleIntersection(FDistance, LDistance, UDistance, WallAngle): navigation(l_turn) & lTurnAimAngle(L_TURN_AIM_ANGLE) <-
    !performRepeatedTurns(L_TURN_AIM_ANGLE - WallAngle, ...);
    !performRepeatedForwards(...);
    -navigation(l_turn).
+!handleIntersection(FDistance, LDistance, UDistance, WallAngle): navigation(u_turn) & uTurnAimAngle(U_TURN_AIM_ANGLE) <-
    !performRepeatedTurns(U_TURN_AIM_ANGLE - WallAngle, ...);
    !performRepeatedForwards(...);
    -navigation(u_turn).
```

5 Practical Challenges

The use of BDI in implementing robot control can give rise to many practical challenges, such as managing conflicting goals, coordinating synchronous and asynchronous actions, and maintaining time synchronization. In this section, we present a generalized summary of our approach for overcoming these challenges.

5.1 Managing Conflicting Goals

A BDI agent is often presented with various goals, which may conflict with one another. To overcome this challenge with Hermes, we followed the subsumption model and broke down the agent's behaviour into a set of layers that behaved independently. Using this model, each goal for the agent was modeled as a distinct layer, and conflicts were handled through subsumption. This design required the corresponding Jason plans to remain independent, and also enabled straightforward extension.

In the subsumption model, each layer is only activated once it has received its stimuli. This means upon the arrival of new perceptions, the system should be able to activate the appropriate layer if needed. To achieve this, we used a custom pattern that involved breaking down the implementation of each layer into two parts: a reactive plan for the layer's activation and a set of goal-based plans for its implementation. The reactive plan for each layer was bound to a specific event, which represented the addition of the main perception needed to activate the layer. We placed the reactive plans for all layers at the top of the

Fig. 11. The Jason plans for the wall following layer.

```
tooFarFromWall(Distance) :-
    wallFollowDistanceSetpoint(SETPOINT) &
    calculateWallDistanceError(WallDistanceError) &
    Distance > SETPOINT + WallDistanceError.
tooCloseToWall(Distance) :-
    wallFollowDistanceSetpoint(SETPOINT) &
    calculateWallDistanceError(WallDistanceError) &
    Distance < SETPOINT - WallDistanceError.
+!wallFollow(Distance, Angle): tooFarFromWall(Distance) <-
    !turn(-1 * AIM_ANGLE + Angle).
+!wallFollow(Distance, Angle): tooCloseToWall(Distance) <-
    !turn(AIM_ANGLE + Angle).
+!wallFollow(Distance, Angle) <-
    !turn(Angle).
```

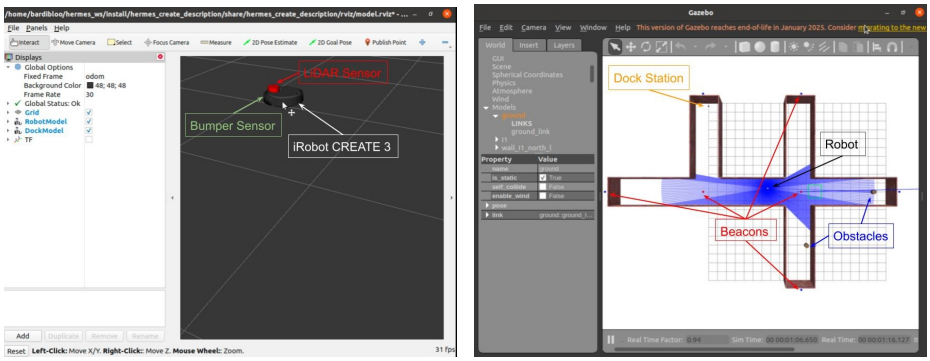


Fig. 12. An overview of the simulator for the project in Gazebo and RViz.

ASL file so they would be chosen first by Jason. Each layer was also given an overall goal. Therefore, each layer was accompanied by a set of plans bound to this overall goal. As such, the reactive plan, when chosen, added this goal to activate its implementation. This pattern has been summarized in Figure 13.

Figure 14 shows the general pattern for implementing each layer to allow for complete independence and subsumption. This pattern ensured that plans detailing the implementations of the layers were simple, to the extent possible, without the need to know anything about any of the other layers. In addition, having an overall desire to guard the layers could help Jason easily check the intention stack to see if a higher-priority layer was being executed. The use of `.drop_all_intentions` was important when selecting a specific layer in this model to allow each layer to start from a clean slate. Note that this still kept

Fig. 13. The proposed Jason pattern for the subsumption architecture.

```
// Layer Activation
+percept1: [conditions] <- !layer1.
+percept2: [conditions] <- !layer2.
// Layer Implementations
+!layer1: [conditions] <- ... .
+!layer2: [conditions] <- ... .
```

Fig. 14. Specifying each subsumption layer in Jason.

```
// Layer Activation
+percept: not(.intend(curr_layer)) & not(.intend(higher_layer))  <-
    .drop_all_intentions;
    !curr_layer.
// Layer Implementation
!curr_layer: [pre-conditions case 1] <- ... .
!curr_layer: [pre-conditions case 2] <- ... .
```

that layer in the agent's desire stack, allowing the agent to quickly go back to the low-level layers after executing the current layer.

5.2 Coordinating Synchronous and Asynchronous Actions

Hermes used synchronous and asynchronous actions, each with its own handling.

- **Synchronous Actions:** these were actions that could be done immediately such as `cmd_vel` which was the move/turn command. The implementation for these actions used Jason's built-in `setResult` mechanism. By default, Jason waits for the result of the action; thus, for our actions, the AgArch class sent the action via ROS, waited for it to complete by obtaining its status using ROS, then set the result of the action, allowing Jason to progress. It was important to ensure that the execution time for these actions was relatively quick to avoid missing reasoning cycles in Jason.
- **Asynchronous Actions:** these were slow actions that could span multiple cycles. An example of these actions was the robot asking for new trips from the server. Figure 15 shows our proposed solution for handling such actions. In this case, `perform_action` is an asynchronous action, which is triggered via a plan with the desire `!performAction`. To avoid repeatedly executing this plan, an intermediate belief of `performingAction` is created to serve as a simple mutex. The plan checks to see if this belief already exists; if not, it adds it to its beliefs and calls the asynchronous action. Jason then awaits the addition of the event that marks the result of this asynchronous action.

Fig. 15. Handling asynchronous actions in Jason.

```
+result(RESULT): true <-
    -performingAction;
    +saved_result(RESULT).
+!performAction: not(performingAction) <-
    +performingAction;
    perform_action.
```

Fig. 16. Handling velocity commands in Jason.

```
+!turn(Angle): speed(SPEED) & convertToRadian(Angle, RAngle) <-
    cmd_vel(SPEED2, 0, 0, 0, 0, RAngle).
+!performRepeatedTurns(Angle, N): N > 1 <-
    !turn(Angle);
    !performRepeatedTurns(Angle, N - 1).
+!performRepeatedTurns(Angle, N): N == 1 <- !turn(Angle).
```

5.3 Time Synchronization

Another concern when dealing with Jason in a real-time context is time synchronization. Hermit is constantly getting information from its environment at different rates. To overcome this, we relied on a central node to handle all perceptions. This node listened to all the sensors and periodically sent an update to Jason with a constant refresh rate of ($0.1s$), ensuring consistency. Additionally, we ensured that the Jason reasoning cycle slept only when an action was being executed, to keep it up to date. ROS's message queues also helped overcome synchronization issues by storing perceptions/actions in a queue (in our case, of size 10) in case the node was not able to handle them right away. Finally, when issuing velocity commands for the robot, the execution duration of them needed to be considered. `cmd_vel` is the standard angular velocity command supported by ROS for controlling most robots. It is a velocity value that the robot executes for a brief period (defined by the motors) or until another one is received, meaning repeated `cmd_vel` calls are needed. As shown in Figure 16, the plan guarded by the `!performRepeatedTurns` allowed the robot to perform repeated turns. This was used repeatedly throughout the project using the following pattern `!performRepeatedTurns(ANGLE, math.ceil(DISTANCE / ACTION_EXECUTION_DURATION));` action execution duration is how long a velocity command is executed for. It is the refresh rate for all nodes ($0.1s$).

6 Performance & Results

There are many performance considerations when working with Jason. Firstly, since working with Jason involves performing reasoning with the help of an

external interpreter, it was important to see if that reasoning time could create a potential bottleneck for the program and overwhelm the system. Additionally, most robotic applications often prefer simple hardware such as a Raspberry Pi, and the potential impact of the hardware on the reasoning performance was important. Finally, we were also curious to assess the agent's task-level performance and the simulator's accuracy as a development tool.

6.1 Jason's Time Performance

The key factor to measure when analyzing Jason's performance is its reasoning cycle, made up of different stages. We measured this reasoning cycle using logs from the robot and simulator on full trips. Figure 17 summarizes these results.

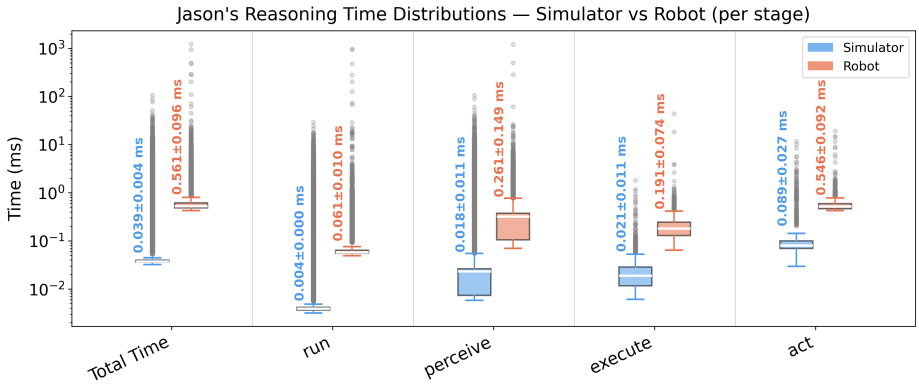


Fig. 17. Analyzing Jason's reasoning time in total and per stage.

As previously stated, the beliefs are fed to Jason every 0.1 seconds, which is a requirement imposed by the sensors in the system. This update frequency is the maximum time that can be allotted for reasoning. If the system surpasses that, it risks getting outdated information. As shown in Figure 17, the average reasoning time is much less than this limit on both the physical robot and simulator. Some outliers go above that limit, but they are very infrequent. These outliers belong to the cycles where Jason is unable to find an applicable plan for its current beliefs and is waiting for new perceptions. These cycles are long, since Jason has to go through every single plan and resolve different logical expressions before deciding there are no applicable ones. The reasoning stages all have consistent run-times with very low standard deviations, which indicates reliability in Jason's performance. These results also match that of [16], where the author demonstrates that Jason's reasoning time is quite fast and it remains under $0.1s$ for up to 10000 beliefs and plans. Although the occasional outliers exist, they do not pose the risk of potentially overwhelming the system; the project relies on ROS's message queue system to record up to 10 messages, meaning that, for a

slow cycle, new beliefs can be saved in the queue to ensure no data loss. Finally, we see that Jason runs slower on the robot than in the simulator. This is due to the difference in the processing power between the two. This difference is small, and Jason's performance is still acceptable.

6.2 Agent's Performance

We assessed Hermes's performance by analyzing its main behaviours: wall following, intersection handling, docking, and collision handling. An experiment setup was created by giving the robot a wall distance threshold of $0.4m$ and placing it at key locations, i.e., near intersections, dock stations, and obstacles, to observe its behaviour. This experiment was conducted in both the simulator and on the robot. Each scenario was repeated to ensure accuracy (30 times in simulation and 5 times on the robot). Figure 18 summarizes the results of these experiments. As shown, the agent maintains a high success rate for these behaviours in both environments. Additionally, it can be seen that the wall following behaviour remains very consistent and close to the expected threshold of $(0.4m)$. This attests to both the accuracy of the agent and the reliability of the simulator. Using this tool, we fast-tracked testing iterations and reduced the need for on-robot testing.

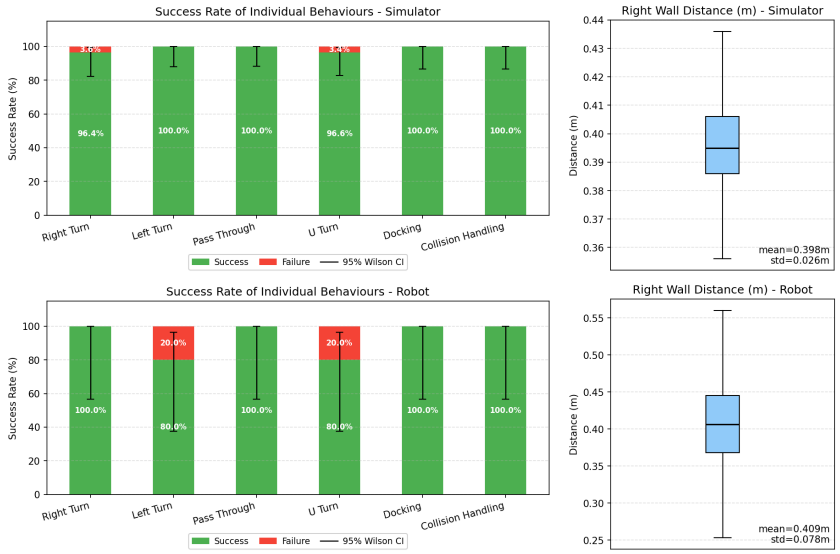


Fig. 18. Overview of the agent's task-level performance in simulation and on the robot.

7 Conclusion

This project described a reasonably complex use case for Jason in a robotic application. The problem to be tackled was the continuation of an earlier attempt at delivering mail in the tunnels of Carleton University, this time using a mobile robot (iRobot CREATE robot), a LiDAR sensor, and a set of Bluetooth beacons. The work also included a realistic Gazebo simulation with the same setup. The agent behaviour followed a subsumption-like model by grouping the agent behaviour into small atomic layers with different priorities. The project also presented the Jason patterns that were used to handle common practical challenges, i.e., goal priorities, action handling, and time synchronization. Jason's performance was also analyzed, showing promising results for robotic applications.

7.1 Future Work

There are many other areas to explore with this work. Most importantly, the scope of the problem could be further increased to include support for multiple robots working together and handling multiple trips at once, requiring the need to account for additional factors such as battery life. Additionally, different programming paradigms, such as imperative programming or a hybrid large language model (LLM) based agent, need further study. By having those implementations in parallel, one could properly analyze their tradeoffs in terms of design and performance on a real-time use case.

References

1. Bordini, R.H., Hübner, J.F.: BDI Agent Programming in AgentSpeak Using Jason. Computational Logic in Multi-Agent Systems **3900**(3), 143–164 (2006). https://doi.org/10.1007/11750734_9
2. Brooks, R.A.: A Robust Layered Control System For A Mobile Robot. IEEE Journal on Robotics and Automation **2**(1), 14–23 (2020). https://doi.org/10.1109/JRA.1986.1087032
3. Cardoso, R.C., Ferrando, A., Dennis, L.A., Fisher, M.: An Interface for Programming Verifiable Autonomous Agents in ROS. In: Multi-Agent Systems and Agreement Technologies. pp. 191–205 (2020). https://doi.org/10.1007/978-3-030-66412-1_13
4. D'Inverno, M., Luck, M.: Engineering AgentSpeak(L): A Formal Computational Model. Journal of Logic and Computation **8**(3), 233–260 (1998). https://doi.org/10.1093/logcom/8.3.233
5. Du, Q., Cardoso, R.C.: Evaluating BDI Agents in ROS: From Basic Integration to Fault Tolerant Multi-Robot Systems. In: European Conference on Multi-Agent Systems (EUMAS 2025) (07 2025)
6. FCC: April Beacon N04 (2025), https://fcc.report/FCC-ID/2ACAL-ABCONN04/4530161.pdf
7. FCC: Raspberry Pi 4 (2025), https://www.raspberrypi.com/products/raspberry-pi-4-model-b/
8. Fernandez, E.: ros2-java (2025), https://github.com/ros2-java/ros2_java

9. Gavigan, P., Esfandiari, B.: Agent in a Box: A Framework for Autonomous Mobile Robots with Beliefs, Desires, and Intentions. Electronics **10**(17) (2021). https://doi.org/10.3390/electronics10172136
10. Georgeff, M., Pell, B., Pollack, M., Tambe, M., Wooldridge, M.: The Belief-Desire-Intention Model of Agency. In: Müller, J.P., Rao, A.S., Singh, M.P. (eds.) Intelligent Agents V: Agents Theories, Architectures, and Languages, Lecture Notes in Computer Science, vol. 1555, pp. 1–10. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-49057-4_1
11. iRobot Education: create3-sim (2025), https://github.com/iRobotEducation/create3_sim
12. iRobot Education: Introducing the CREATE 3 educational robot (2025), https://edu.irobot.com/what-we-offer/create3
13. Jennings, N.R., Sycara, K., Wooldridge, M.: A Roadmap of Agent Research and Development. Autonomous Agents and Multi-Agent Systems **1**, 7–38 (1998). https://doi.org/10.1023/A:1010090405266
14. Laboratório de Sistemas Autônomos: Jason-ROS (2025), https://github.com/lsa-pucrs-old/jason-ros-releases
15. Laboratório de Sistemas Autônomos: Rason (2025), https://github.com/lsa-pucrs-old/rason
16. Miller, J., Esfandiari, B.: Analysis of the execution time of the jason bdi reasoning cycle. In: Alechina, N., Baldoni, M., Logan, B. (eds.) Engineering Multi-Agent Systems. pp. 218–236. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-97457-2_13
17. Onyedinma, C., Gavigan, P., Esfandiari, B.: Toward Campus Mail Delivery Using BDI. Journal of Sensors and Actuators **9**(56), 127–143 (2020). https://doi.org/10.3390/jsan9040056
18. ROS: Gazebo simulation (2025), https://gazebosim.org/about
19. ROS: Robot operating system (2025), https://www.ros.org/blog/why-ros/
20. ROS: RViz (2025), https://wiki.ros.org/rviz
21. SLAMTec: RPLiDAR A1 (2025), https://www.slamtec.com/en/lidar/a1
22. SMART Research Group (PUCRS): JROS (2025), https://github.com/smart-pucrs/JROS
23. Wesz, R.B.: Integrating Robot Control into the AgentSpeak(L) Programming Language. In: Workshop-Escola de Sistemas de Agentes, seus Ambientes e Aplicações. pp. 197–203 (05 2014). https://doi.org/10.5753/wesaac.2014.33302