Toward Campus Mail Delivery Using BDI

Chidiebere Onyedinma
University of Ottawa
Ontario, Canada
conye066@uottawa.ca

Patrick Gavigan Babak Esfandiari
Carleton Univeristy
Ontario, Canada
patrickgavigan@sce.carleton.ca babak@sce.carleton.ca

Autonomous systems developed with the Belief-Desire-Intention (BDI) architecture are usually mostly implemented in simulated environments. In this project we sought to build a BDI agent for use in the real world for campus mail delivery in the tunnel system at Carleton University. Ideally, the robot should receive a delivery order via a mobile application, pick up the mail at a station, navigate the tunnels to the destination station, and notify the recipient.

We linked the Robot Operating System (ROS) with a BDI reasoning system to achieve a subset of the required use cases. ROS handles the low-level sensing and actuation, while the BDI reasoning system handles the high-level reasoning and decision making. Sensory data is orchestrated and sent from ROS to the reasoning system as perceptions. These perceptions are then deliberated upon, and an action string is sent back to ROS for interpretation and driving of the necessary actuator for the action to be performed.

In this paper we present our current implementation, which closes the loop on the hardware-software integration, and implements a subset of the use cases required for the full system.

1 Introduction

Autonomous systems are designed in such a manner that they can intelligently react to ever-changing environments and operational conditions. Given such flexibility, they can accept goals and set a path to achieve these goals in a self-responsible manner while displaying some form of intelligence. An autonomous agent can be defined as a system that senses the environment and acts on it, over time, in pursuit of its own agenda and to affect what it senses in the future [2].

The Belief-Desire-Intention (BDI) framework is meant for developing such autonomous agents, in that it defines how to select, execute and monitor the execution of user-defined plans (Intentions) in the context of current perceptions and internal knowledge of the agent (Beliefs) in order to satisfy the long-term goals of the agent (Desires). But so far, very few applications of BDI have been observed outside of simulated or virtual environments. In this paper, we describe how we built our autonomous robot that uses BDI (and specifically, the Jason implementation of the BDI AgentSpeak(L) Language (ASL)) and Robot Operating System (ROS) to eventually deliver interoffice mail in the Carleton University campus tunnels. The Carleton tunnel system allows people to go from one campus building to another without having to face Ottawa's harsh winters, and makes for a fairly controlled environment that our iRobot platform can navigate. However, being underground also means that access to GPS is not possible, and internet access is limited to certain areas. In this context, ultimately the robot will have to know where it is and where to deliver mail, but there are also some sub-goals, like obstacle avoidance and battery recharge, which it might have to achieve in other to get to its main goal.

In the remainder of the paper, we first provide some background on BDI, ROS, and related work on known implementations of agent-based robots; we then describe our overall hardware and software architecture; next, we describe in more detail the hardware and software implementation; finally, we describe the few test cases that we have implemented so far using the completed robot.

2 Background

We provide background on BDI and the ASL in section 2.1. We then introduce ROS in section 2.2 followed by a discussion of various related work in section 2.3.

2.1 Belief-Desire-Intention Architecture

The principles that underpin BDI originated in the 1980s cognitive science theory as a means of modelling agency in humans [3]. Since that time, this model has been applied in the development of software agents as well as the field of Multi Agent System (MAS). An example of a popular implementation of applying BDI to agent reasoning is Jason [2] [13]. In Jason, an agent's initial belief base, goals and plans are specified using ASL. This language is also used for communication between agents.

In BDI systems, a software agent performs reasoning based upon internally held *beliefs*, stored in a belief base, about itself and the task environment. The agent also has objectives, or *desires*, that are provided to it, as well as a plan base, which contains various means for achieving goals depending on the agent's context. The agent's reasoning cycle consists of first perceiving the task environment, and receiving any messages. From this information, the agent can then decide on a course of action suitable to the context provided by those perceptions, the agent's own beliefs, messages received, and desires. Once this course of action has been selected, we can say that the agent has set an *intention* for itself. These plans can include updating the belief base, sending messages to other agents, and taking some action in the task environment. As the agent continues to repeat its reasoning cycle, it can reassess the applicability of its intentions as it perceives the environment, and drop intentions that are no longer applicable [2] [13].

Agents developed for BDI systems using Jason are programmed using a language called ASL. This is a logic based programming language that bears similarities to Prolog. The syntax provides a means for specifying initial beliefs for the agent to have, rules that can be applied for reasoning as well as plans that can be executed.

Listing 1: Example AgentSpeak program.

```
destination(post1).
atDestination :- destination(DESTINATION) & postPoint(DESTINATION,_).
+!goToLocation : atDestination <- drive(stop).
```

The ASL syntax is illustrated in listing 1. First, we have a simple belief in the form of a predicate, in this case it is the knowledge that the destination (of our robot) is `post1`. Next, we have a rule following the format of *implication*, where the $:-$ operator holds the meaning of the implication arrow ($\Longrightarrow$) in reverse. In this case, we have a rule defining `atDestination`, which is implied when `DESTINATION` unifies to the current location specified by the `postPoint(CURENT,PREVIOUS)` perception. Finally, we have a plan, which follows the form:

```
triggeringEvent : context <- body.
```

A *triggering event* can be the addition or deletion of a belief, achievement goal, or a test goal. Achievement goals begin with an exclamation mark (!), test goals begin with a question mark (?) and beliefs do not begin with punctuation. Triggers that are based on the addition or deletion of a belief or goal begin with a positive (+) or negative (-) sign respectively. An achievement goal is used for providing the agent with an objective with respect to the state of the environment whereas a test goal is generally used for querying the state of the environment. The *context* is a set of conditions that must be satisfied in order for the plan to be applicable based on the state of the agent's belief base. This is a logical sentence that can use both beliefs as well as previously defined rules. The *body* includes the instructions for the agent to follow for executing the plan. The plan body can include the addition or deletion of beliefs and/or goals as well as actions for the agent to perform [2] . In our example, we have a plan for the achievement goal of `!goToLocation`. This plan sends the action drive(stop) when the `atDestination` context rule is satisfied.

2.2 Robot Operating System

ROS is a package for developing software for robotic applications [28]. This system operates using a tuple-space architecture. Various software nodes publish and subscribe to various *topics* using socket-based communications. This is managed using a central master node which has the role of brokering peer-to-peer connections between nodes that publish and subscribe to the same topics. This allows developers to focus on the implementation of individual nodes and enables flexibility to use one of many available nodes that are compatible with ROS. For example, various hardware component developers have made ROS nodes available, allowing systems developers to use those modules without concern as to how those nodes are implemented in detail. ROS also provides functionality for recording runtime data, which can be used for diagnostics.

2.3 Related Work

Although there are many examples of research on software agents and the use of BDI, this review of related work focuses on the application of BDI to robotics where the development was targeted toward real-world applications. We will also discuss work that sought to use BDI agents with ROS.

The Australian military conducted research into the use of BDI for controlling a fixed-wing Unmanned Aerial Vehicle (UAV) called a Codarra Avatar. As part of this project, they developed both the "Automated Wingman", a graphical programming environment where pilots could provide mission-specific programming for a UAV, as well as a BDI-based flight controller for the UAV itself. The intent of this research was to enable pilots, who may not have programming skills, to provide mission parameters in a way more natural to them using the military's Observe Orient Decide Act (OODA) loop. The authors proposed that the OODA loop could be approximated using BDI. Successful flight tests were performed using these systems in the mid 2000s, although it is unclear if any follow-on research was conducted [33] [15].

A more recent example of BDI being used for controlling a drone was provided by Menegol [18] [19]. Their implementation uses the JaCaMo framework [24], which includes Jason. A video of their UAV flying is available online [26]. This work is currently being extended to use the ROS as the core of the architecture [32] [31]. Their approach is to build a linkage between ROS and Jason, where Jason agents can run actions by passing messages to modules in ROS and receive perceptions by receiving messages from other modules. The perceptions and actions are defined using manifest files that specify the properties and parameters of the messages. This is similar to other efforts to link ROS to Jason, such as rason [21], JaCaROS [20], and JROS [5], although it is unclear if these efforts are related to this project.

Taking another approach using Python, the Python RObotic Framework for dEsigning sTrAtegies (PROFETA) library implements BDI and the AgentSpeak language designed for use with autonomous robots [10]. They are interested in determining if Agent Oriented Programming (AOP) can be implemented with Python for simpler robotic implementations. In their paper, the authors used the Eurobot challenge as well as a simulated warehouse logistics robot scenario as case studies. In the Eurobot challenge, the robot must sort objects in the environment while also working in the presence of other, uncooperative, robots [22].

The ARGO project [29] has interfaced Jason agents with Arduino using a library called Javino [16]. Javino is a Java library for controlling Arduino computers from Java programs that was specifically designed with the intention of using it to control a robot using Jason programs. That being said, the authors of the ARGO paper claim to not be tied to specific hardware or a specific AOP language, such as AgentSpeak [29] [16].

Alzetta and Giorgini contributed work toward a real-time BDI system connected to ROS 2 [1] [9]. Their implementation uses a custom built BDI engine, implemented in C++, which supports soft real-time constraints. The agent's *desires* are encoded with soft real-time deadlines for when they need to be achieved. The plans in the agent's plan library include the execution time for that plan. The agent reasoning system can then reason about the priority of desires, time constraints and execution time when performing plan selection.

Finally, the authors' own related work includes the Simulated Autonomous Vehicle Infrastructure (SAVI) project, which aimed to develop an architecture for simulating autonomous agents implemented using a Jason BDI [8] [7]. Among its key features is the decoupling of the agent reasoning cycle from the simulation time cycle, enabling the simulated agents to run in their own time. The agent's perceptions and actions passed between the simulated agent body running in a separate thread and decoupled from the agent's reasoning cycle. Although this system was targeted toward a simulated environment, the design was intended to be useful for application to robotic agents, not only simulated agents.

3 Motivation

Our use of BDI for this work is two fold. First, BDI provides a good goal oriented agent architecture that is resilient to plan failure and changes to context. It also supports the notion of shorter-term and longer term plans that can be organized so as not to conflict with each other. BDI is also inherently a social agent system, providing useful utilities for message passing and communication in MAS, which relates to future work for this project. Granted, BDI may not necessarily be the *perfect* ad hoc solution for agent based robotics, but there's just not enough literature to demonstrate the appropriateness of BDI (or lack thereof) in robotics.

There are alternative agent architectures to BDI that are available, for example the subsumption architecture [4] [34]. Although it is likely possible that the robotic behaviours implemented in this paper likely could have implemented the same robot using subsumption, our longer term goals for this project would likely make the use of other architectures more difficult. BDI is inherently social and goal directed whereas in the case of the subsumption architecture, the agent behaviour emerges from the various layers built into the agent [6].

Our goal is to use an established BDI system, namely Jason in an ecosystem for various robotic platforms (ROS) and enable to use of agent systems to solve real-world problems using robotics, taking advantage of ROS' ecosystem of publishers and subscribers. As mentioned in section 2, while there are some projects that have sought to control real-world robotics using BDI reasoning systems, there are a limited number of works in this area. Those that are available differ from our work in a number of ways. For example, in the case of the Codarra avatar agent, although it is very interesting, it does not seem to be openly available. Other work, such as PROFETA, uses a Python based BDI, as opposed to the more commonly used Jason. Our work is more similar in motivation to the efforts to link Jason and ROS mentioned in section 2, although our implementation of the connection between ROS and Jason is quite different. In our case, the BDI reasoning system is built as a stand alone program with rosjava using Jason as a library, without the use of an external middleware.

4 Architecture

This section outlines the architecture of the mail delivery robot. The robot is intended to function on an on-demand basis, where a mail-sending user would summon the robot to collect mail, similar to how users request rides using ride-sharing apps. The robot would then autonomously navigate to a nearby mail collection and delivery location to collect the item from the user. Once the mail has been collected, the robot would then autonomously navigate to the mail delivery location and alert the receiver that there is mail for them to receive. The receiver would then meet the robot at another mail collection and delivery location. For the purposes of this early stage prototype, the mail delivery locations, and any other points of interest are indicated using a Quick Response code (QR code), and the robot paths are marked using a line for the robot to follow. Removing the need for instrumenting the environment will be discussed in the future work, in section 7.2.

First, in section 4.1, we examine the task environment that the robot will operate in. We then discuss the hardware configuration in section 4.2. The software architecture is discussed in section 4.3.

4.1 Environment

The task environment for the robot is the tunnel system that connects the buildings of Carleton University. This provides our robot with an indoor space with no weather to deal with and smooth floors to drive on, and also connects to many buildings on campus. Although these are attractive features of the tunnel system, there are some drawbacks. First, the tunnels do not have consistent wireless internet coverage, although there are locations where there is reliable network access. The tunnels also have lower lighting levels than typical office environments, providing a potential challenge to the design. Finally, in the tunnel there is no access to Global Navigation Satellite System (GNSS) signals, such as Global Positioning System (GPS), meaning that the robot will need to determine its location another way.

4.2 Hardware

The hardware configuration of the mail delivery robot is shown in figure 1. The mail delivery robot is primarily implemented using an iRobot Create2, which is the development version of the Roomba vacuum cleaning robot, without the vacuum-cleaning components. This robot can be controlled using a command protocol over a serial interface [14], and can also be used to provide power to other connected devices. A Raspberry Pi 4 computer was attached to the robot and connected via a serial cable, and powered from the robot's battery using a power adapter. Also connected via a serial connection are a camera and a line sensor used for detecting a line on the floor of the tunnels.

4.3 Software

The control software is implemented using a set of modules connected via ROS, as shown in figure 2. The reasoning system for this robot, inspired by the SAVI project [8, 7], decouples the reasoning cycle from the interface to the sensors and actuators using a state synchronization module. The internal reasoning system for this project, called *SAVI ROS BDI*, and shown in figure 3, is inspired by the original SAVI configuration. Implemented in Java, using the rosJava package [25] and the Jason BDI engine [2]

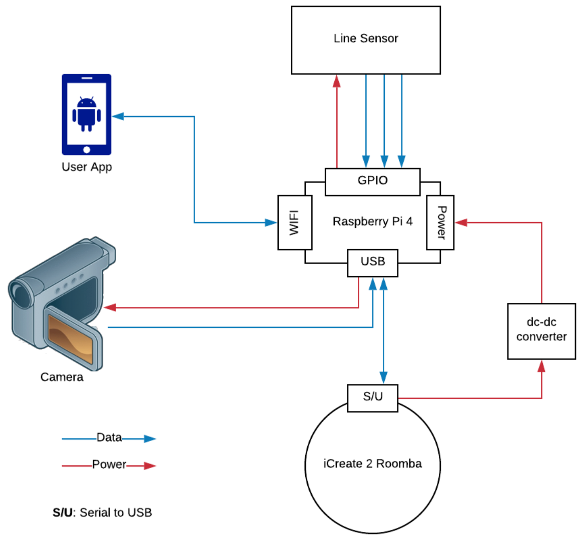


Figure 1: Mail delivery robot hardware.

[13], this module connects to ROS directly, subscribing to perceptions and inbox messages and publishing actions and outbox messages as required. Again, the state synchronization module is important as perceptions and messages can arrive at any time, decoupled from the reasoning cycle of the agent. This is set up in three main components: The ROS connectors, the state synchronization module, and the agent core. The ROS connectors are responsible for subscribing to either perceptions or inbox messages, or publishing actions or outbox messages, each in their own thread of execution. These are connected to the state synchronization module, which manages queues or messages in and out of the agent as well as perceptions and actions in and out of the agent. The agent core, which runs the agent reasoning cycle in a separate thread of execution, checks for perceptions and inbox messages at the beginning of the reasoning cycle. Then, the agent decides on an appropriate course of action and then updates the agent state with new outbox messages and actions which need to be executed. The agent behaviour is defined by an ASL file which is parsed by the reasoning system at start-up, making this module fully platform agnostic: there are no assumptions about the underlying hardware, capabilities, or mission of the agent in the implementation of this system. This agent reasoning system is available at [11].

The Create2 robot platform can use the `create_autonomy` package available in ROS, which connects to an underlying C++ library called `libcreate` to ROS, publishing the data from various sensors as ROS topics and subscribing to topics related to the various commands available to the robot [30]. Also connected in this way are drivers for the QR code camera and photodiode line sensor, which each publish their data as ROS topics. Also connected to ROS is SAVI ROS BDI, described above. Lastly, as required by SAVI ROS BDI, is the application node translator, which reformats sensor data as AgentSpeak perceptions and conversely translates action commands in AgentSpeak to the relevant topics being subscribed to by the `create_autonomy` package. Lastly, an AgentSpeak program is provided to the reasoning system, which defines the behaviour of the agent. The implementation of the perception and action translators, the drivers for the QR code camera and the line sensor, and the ASL program are available at [12].

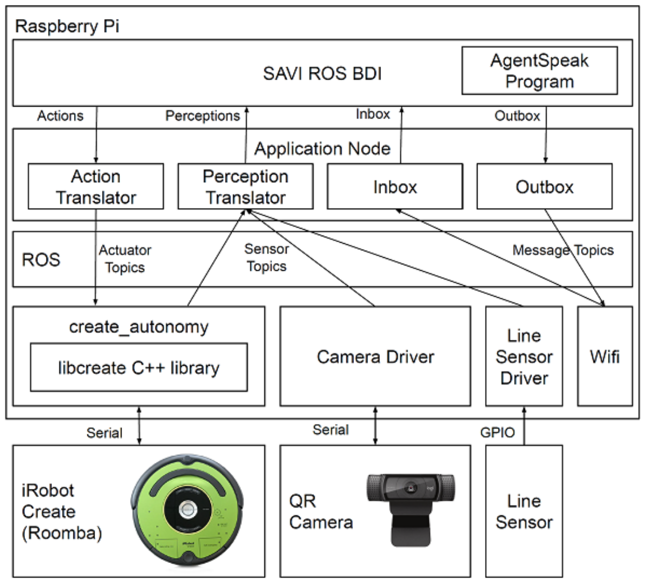


Figure 2: Mail delivery robot software architecture (robot image credit [14], camera image credit: [17]).

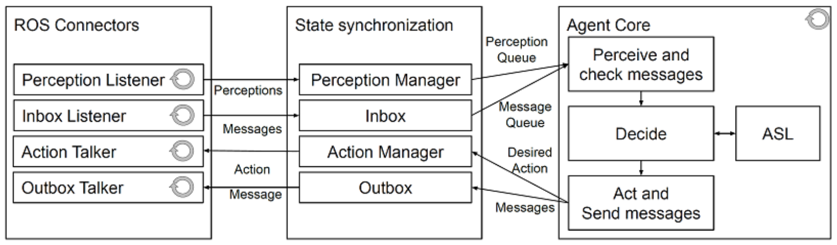


Figure 3: SAVI ROS BDI internal architecture.

5 Implementation

This section discusses the implementation of the various aspects of the system, shown in figure 4. The source code for this project can be found on GitHub [12, 27]. First, we show how we powered the on-board computer using the robot's power system, in section 5.1. Next we discuss the means of maneuvering the robot using line sensing, including the implementation of the line sensor and associated driver in section 5.2. The robot uses a system based on QR code for determining its position, given the lack of other methods such as GNSS. This is discussed in section 5.3. The user interface, implemented as an Android app, is discussed in section 5.4. The action translator, which handles the implementation of the robot's actuators is explained in section 5.5. Finally, the details of the implementation of the agent behaviour, in ASL are provided in section 5.6.

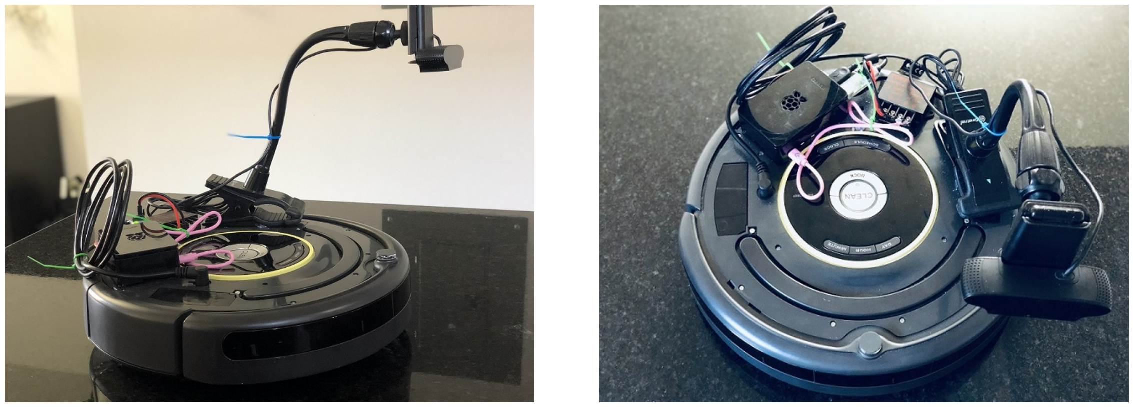

(a) Side view of the robot. (b) Top view of the robot.

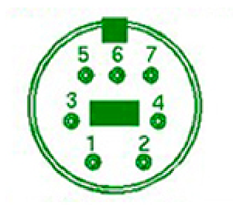
Figure 4: Assembled robot prototype.

5.1 System Power

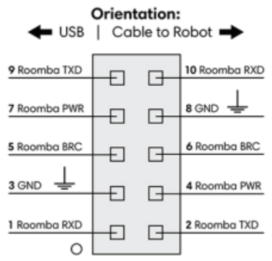
In order to power the robot's computer (a Raspberry Pi 4) without having it tethered to a socket in a wall, we utilized the iRobot Create 2's power system. This was possible as the serial connection between the computer and the robot also provides access to the robot's internal rechargeable battery. Conveniently, the serial cable used to connect the Create 2 to the Raspberry Pi exposes the robot's power bus through its RS232 pinout, as seen in figure 5 and table 1.

Although this pinout provides access to the robot's power supply, it must be converted from 16-20 V to the regulated 5 V required by the Raspberry Pi computer via its USB-C connector or its General-Purpose Input/Output (GPIO) pin. To get a stable 5 V for our Raspberry Pi, we used a Tobsun 15W DC to DC power converter by feeding power to its input (12 V/24 V positive and negative) terminal from pin 4 and pin 3 of the Serial to USB header described in figure 5b respectively. We connected the exposed wires of an improvised USB type C cable to the converter and then we plugged in the cable to the Raspberry Pi; when the RS232 end of the Serial-to-USB cable is plugged into the Create2 robot, the entire system is powered successfully.

With the robot and computer successfully powered by the robot's power supply, it is necessary for the reasoning system to have awareness of the battery charge state, so that it can report to a charging station if necessary. The `create_autonomy` package regularly publishes a ROS topic called `battery/chargeratio`



(a) RS232 pinout.



(b) USB pinout.

Figure 5: Serial connection pinout [23].

Table 1: Create 2 external serial port RS232 connector pinout.

Pin	Name	Description
1	Vpower	battery + (unregulated) 16 V to 20 V
2	Vpower	battery + (unregulated) 16 V to 20 V
3	RXD	0 V to 5 V Serial input to robot
4	TXD	0 V to 5 V Serial output from robot
5	BRC	Baud Rate Change
6	GND	battery ground
7	GND	battery ground

which indicates the percentage of charge left on the battery based on its capacity. The perception translator node, implemented in Python, subscribes to this topic and publishes a `batteryOK` or `batteryLow` string to the `perceptions` ROS topic. A `batteryOK` is published when the charge left in the battery is above 25% and a `batteryLow` otherwise. If the robot receives a `batteryLow` perception, it drops all other plans and looks for the closest docking station in order to recharge itself, explained in more detail in section 5.6.

5.2 Maneuvering with Line Sensing

As our robot operates in an indoor environment without the support of GNSS systems for navigation, a simple means of moving through the tunnels and navigation was required. As an initial implementation, a line sensor was used for the robot to follow lines on the tunnel floor. This sensor is implemented using three Photo-interrupter LTH 1550-01 diodes, shown in figure 6. Each sensor detects if the line is on the left, center or right of the robot's center. Two resistors were used per Photo-interrupter, a 220 Ω and a 33 kΩ. The 220 Ω was used as a limiting resistor for the LED within the sensor and the 33 kΩ as a voltage divider to enable us to measure the voltage across the resistor when light falls on the photo-transistor.

The sensors were connected to three different GPIO pins on the Raspberry Pi. The right sensor is connected to GPIO14 (pin8), the center sensor to GPIO15 (pin10) and the left sensor to GPIO18 (pin12). The sensor is powered from the Raspberry Pi; the VCC pins are connected together and then to the 5v pin of the Raspberry Pi, while the ground (GND) pins are connected together and then to the ground (GND) pin of the Raspberry Pi. When light falls on each of these sensors, their GPIO pins are set to HIGH, and

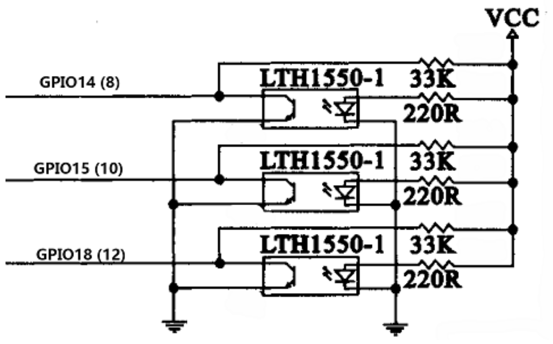
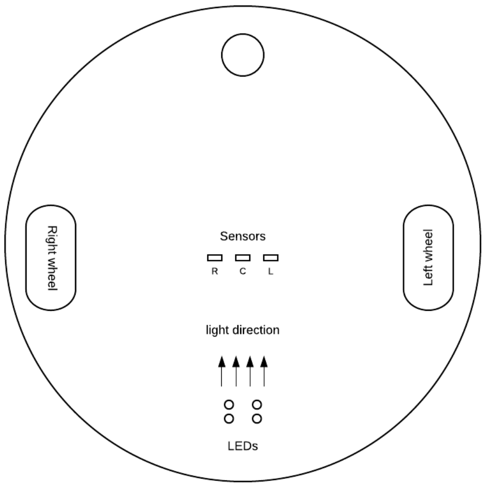


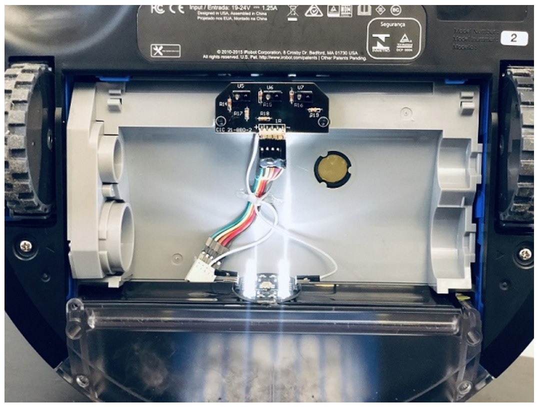
Figure 6: Line sensor circuit.

when the sensors are covered or faced with a non-reflective material or has no light falling on them, their GPIO pins are set to LOW.

The navigation track was designed using a reflective black tape, so that when it is faced by any of the sensors, the respective GPIO pin is set to HIGH, and then we know if the line is on the right, center or left depending on the pin that was set to HIGH or LOW. The sensors are mounted under the center of the Create2, in line with the right and left wheels, as seen in figure 7a. An image of the underside of the robot itself is provided in figure 7b. This is to ensure more navigation accuracy, because if the sensors are mounted in front, or behind the wheels, the line would be detected before or after the robot needs to make a navigation decision. For example, if the sensors are mounted in front of the wheels, and while the robot is in motion (following the line) the line changes direction; the change in direction is detected first by the sensors making the robot to turn and change its direction before it needs to, thereby making it go out of track.



(a) Robot base layout.



(b) Under the robot.

Figure 7: Layout of the underside of the robot.

The robot has a broad surface area and when on the floor, has little or no light underneath it. Since our sensors are mounted under the robot, they cannot function effectively because they need a certain amount of light in order to detect the line. A light source under the robot using four LEDs was added. These LEDs were mounted perpendicular to the sensors with their light directed at the sensor. With this in place, when the robot is on the floor, the light bounces off any reflective object or material placed on the floor, and is absorbed by non-reflective materials or objects.

The line tracks are created using tapes. To ensure enough contrast between the line to follow and the floor regardless of the environment, we had to create a track with two different types of tape, reflective and non-reflective, with the non-reflective tape in the center. This type of line track would work irrespective of location flooring. The robot is kept on the track, with the center sensor on the non-reflective tape, so when any of the sensors is faced with the non-reflective tape, we know the line track is in that direction.

5.2.1 Sensing and Publishing Line Location

With the line sensor hardware implemented, we needed to consider how the signals would be sent to the BDI reasoning system. This was implemented in a software driver implemented in Python (referred to as the line sensor driver in figure 2). This driver consists of a ROS node which runs in a 10 Hz loop, implemented using ROS's `rospy.Rate()` and `rospy.sleep()` functions, and interfaces with the hardware via the Raspberry Pi's GPIO library. The software and monitors if the signals from the GPIO pins are HIGH or LOW, indicating if the diodes of the line sensor are detecting the line under them.

The ROS node publishes strings to the `perceptions` topic. These include `line(center)`, meaning that the line was detected in the center of the sensor, `line(left)` and `line(right)` which means that the line was detected by the left or right sensor and possible also the center sensor. The perception `line(across)` is used to indicate that the line was detected by all of the sensors and the `line(lost)` indicates that the line is not visible to any of the sensors. These perceptions were then received by the BDI reasoner and interpreted as part of the agent reasoning cycle, discussed in section 5.6.

5.3 Location sensing with QR Codes

As the tunnel system in which the robot is expected to operate has no access to external navigation systems, such as GNSS, it was necessary for the robot to have another means of identifying its location. This was accomplished by posting QR code codes along the path of the robot but without obstructing the line track that the robot would be following. The camera used for scanning the codes was also positioned on the left side of the robot and ten inches from the floor because of its focal length; this was to enable the camera to capture the code properly. For two-directional travel, a QR code was placed on either side of the line, enabling the robot to detect a code that could also inform the robot of the direction of travel.

The QR code is scanned using a ROS node which is responsible for managing the camera, called the camera driver in figure 2. Implemented in Python, the camera driver scans for QR code codes at 2 Hz rate. When a code is detected, the location number included in the image is published to the `perceptions` ROS topic and saved in a location history in the camera driver. The format of this perception is: `postPoint(C, P)` where C is the current scanned location number, and P is the previously scanned location number. This predicate is received by the BDI reasoning system and processed using the AgentSpeak rules discussed in more detail in section 5.6.

5.4 User Interface

The user interface currently consists of a rudimentary mobile application for the Android platform, and a Python script that resides on the robot, responsible for relaying messages from the user to the `perceptions` topic. Upon start up, a Transmission Control Protocol (TCP) connection is established between these two applications. After a successful connection to the robot's communication node, the sender can then select the pickup post point and the destination post point. This is received by the node, and the destination is published to the perception's ROS topic as a predicate string `dest(D)` where D is the post location number of the destination. The user interface is available online [27].

5.5 Action translator

When the robot reasoning system requests that an action be performed by the robot, the action is published to the `actions` ROS topic. These messages are interpreted by the `action translator`, a Python script which subscribes to the `actions` topic and then publishes messages to the appropriate topics for the `create_autonomy` node to control the lower level hardware of the robot. The action messages that are currently supported include actions for driving the robot forward, turning left and right, and stopping the robot. These commands are: `drive(forward)`, `turn(left)`, `turn(right)`, `drive(stop)`. We also support actions for docking and undocking the robot from the charging station using the internal programming of the robot: `dock_bot` and `undock_bot`.

5.6 Agent behaviour

The reasoning system receives inputs via perceptions and actuates via actions based upon the results of its reasoning cycle. The agent behaviour is defined for the Jason BDI reasoner in ASL. The ASL program uses a hierarchy of behavioural goals, each of which have supporting plans providing the agent with a means of achieving the goals in a given context.

The testing environment map is shown in figure 8. The plans for the navigation goals of the robot use rules defined for `atDestination`, `DestinationBehind`, `DestinationAhead`, `DestinationLeft`, and `DestinationRight`. In turn these rules depend on internally held beliefs within the robot which define the location of the mail sender, mail receiver and the docking station. These beliefs are structured in the form of predicates defined as `senderLocation(post1)`, `receiverLocation(post4)`, and `dockStation(post5)`, where the terms in brackets are locations on the map. These beliefs can be hard coded in the ASL file or be provided to the agent via communication messages.

Examples of the rules used for this map are provided in listing 2. First we see the `atDestination` rule, where the agent should have the belief that the currently detected post point, provided by a perception, matches the `destination()` belief. In this case, the term `DESTINATION` must unify. The second term in the `postPoint()` is an underscore (_) as this term defines the previously detected post point, which is not relevant to this rule. Next we have an example of a rule for `DestinationRight` for the scenario where the robot's destination is `post4`, `post2` was the previously detected location and is currently detecting `post3`. These and all other rules (which will be automatically generated based on the map configuration in our next iteration) can be found at [12].

Listing 2: Example rules for the map.

```
atDestination :- destination(DESTINATION) & postPoint(DESTINATION,_).
DestinationRight :- destination(DESTINATION) & postPoint(CURRENT,PAST) &
CURRENT = post3 & PAST = post2 & DESTINATION = post4.
```

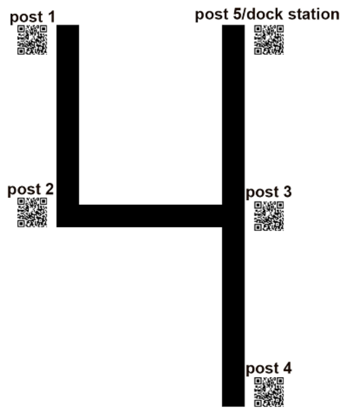


Figure 8: Map of the test environment.

The highest level goal for this robot is `!deliverMail`. Listing 3 shows some examples plans that are triggered by this goal. First, we see a plan for a scenario where the agent does not have `haveMail` in the knowledge base. Here, the context check unifies the location of the sender and verifies if the agent is already at the sender's location. The battery charge is also checked to make sure that there is sufficient power to continue. If this context is satisfied, the agent adds a predicate to set the destination as the sender's location. The agent then adopts the goal of `!goToLocation` and also readopts the goal of `!deliverMail`, as the goal of delivering the mail has not yet been achieved, also making this a recursive plan. The second plan that can trigger on this goal addresses the situation where the robot has arrived at the sender's location. In that situation, the robot adds the belief of `haveMail` to the knowledge base and also removes any previous destination. Again, this is recursive as the mail has not yet been delivered. Two additional, and very similar plans, are required for setting the destination as the receiver for the cases where the mail has been collected. Those plans are available at [12]. Next we see the scenario where the battery has an insufficient charge. In this situation, the destination is set to the docking station.

Listing 3: Example mail delivery plans.

```
+!deliverMail : ((not haveMail) & senderLocation(SENDER) &
receiverLocation(RECEIVER) & not postPoint(SENDER,_) & batteryOK)
<- +destination(SENDER); !goToLocation; !deliverMail.

+!deliverMail : ((not haveMail) & senderLocation(SENDER) &
receiverLocation(RECEIVER) & postPoint(SENDER,_) & batteryOK)
<- +haveMail; -destination(_); !deliverMail.

+!deliverMail : (batteryLow & dockStation(DOCK)) <- -destination(_);
+destination(DOCK); !goToLocation; !deliverMail.
```

Listing 4 provides the plans for navigating the robot to the destination in order to achieve the goal of `!goToLocation`, which is invoked by the plans that achieve the goal of `!deliverMail`. These plans are fairly simple thanks to the rules previously defined for their context definitions. The plans use the actions of `drive(stop)` to stop if the robot is at the destination, and `turn()` to turn in the direction of the destination. Otherwise, the plans invoke the goal of `!followPath` to follow the line path, and are all recursive if the agent is not yet at the destination.

Listing 4: Navigation plans.

```
+!goToLocation : destinationAhead <- !followPath.
+!goToLocation : atDestination <- drive(stop).
```

```
+!goToLocation : destinationLeft <- turn(left); !followPath.
+!goToLocation : destinationRight <- turn(right); !followPath.
```

The plans for achieving the goal of `!followPath` are shown in listing 5. The first plan commands the robot to drive forward when the line is detected in the center of the line sensor. Next we have a plan that stops the robot if the line is not visible. Lastly, there is a plan that turns the robot in the direction that the line is detected, using unification.

Listing 5: Path following plans.

```
+!followPath : line(center) <- drive(forward); !followPath.
+!followPath : line(lost) <- drive(stop).
+!followPath : line(DIRECTION) <- drive(DIRECTION); !followPath.
```

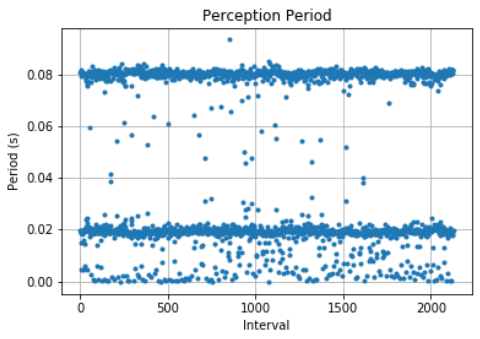
In the situation where the robot needs to dock to charge the battery, the agent adopts the goal of `!dock`. The plans for achieving this goal are provided in listing 6. Here, we have two plans, the first for going to the dock location, if we are not already there, and the second for initiating the `dock_bot` action.

Listing 6: Docking plans.

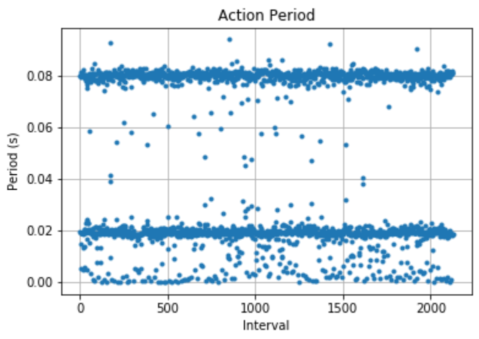
```
+!dock : dockStation(DOCK) & not postPoint(DOCK,_) <- !goToLocation; !dock.
+!dock : dockStation(DOCK) & postPoint(DOCK,_) <- drive(stop); dock_bot.
```

6 Evaluation

We evaluated our prototype robot in two ways: by observing it driving through a test course and by measuring the publication period of perceptions and actions to ROS topics. Our interest was to assess if the BDI reasoning system was running sufficiently fast in order to keep up with the perceptions being generated.



(a) Perception period



(b) Action period

Figure 9: Perception and action message periods.

Figure 9 shows the period at which perceptions and actions were published to their associated topics in ROS as the robot was executing the mail delivery task. As various perceptions are published at different rates, we can see a strong tendency for there to be messages approximately every 0.02 s and every 0.08 s. As the reasoning system for BDI generates an action every time a perception is received, we conversely see actions being published at approximately the same rate. This tells us that the BDI reasoning system was able to keep up with the perceptions generated by the various sensors. Further investigation will include profiling our system to determine where it spends the most time.

7 Conclusion

In this paper, we presented the work to date on the development of a robotic agent for performing autonomous mail delivery in a campus environment. We conclude with a description of our key accomplishments and a view toward our future work.

7.1 Key Accomplishments

We demonstrated the feasibility of using BDI in an embedded system. We accomplished this using the SAVI ROS BDI framework, linking Jason's BDI reasoning system to ROS. We also implemented our initial robot behaviours in BDI, navigating through the environment using line-following and QR code while also monitoring the battery state, seeking a charging station as needed.

We integrated the reasoning system onto a Raspberry Pi computer and connected it to the iRobot Create2, powering it from the robot's internal power. We integrated a line sensor and camera and developed the necessary nodes for providing their data to the BDI reasoner as perceptions via ROS. We provided a translator for the `create_autonomy` package, passing sensor data from the robot to the reasoning system, and actions back to the actuators.

7.2 Future Work

Our implementation uses line sensing and QR code to move around the environment and localization. This method was used as a *first iteration* for early development of our prototype system, but it has drawbacks, notably requiring the environment to have line tracks and QR code. One approach could be to add more docking stations, using them more than just as a charging station but also as guiding beacons (make each station emit a different code to make it distinguishable by the robot), and more generally turning them into full-blown stations for mail drop-off and pick-up. These spots could be placed where wi-fi is accessible so that the robot can receive its missions and notify the recipient that the delivery is ready. With the prototype working, a revisit to the ASL implementation is needed. In the current implementation, the ASL code could be refactored to be more idiomatic and less redundant now that we have demonstrated that the underlying components work. Additionally, this includes moving toward the use of the message-passing utilities of Jason. For example, messages from the user should be implemented as messages to the agent, not as perceptions. Another desire is to have multiple robots handling mail delivery together. The robots could work as a team, possibly handing off mail from robot to robot, and managing their battery levels. A user would not summon a specific robot to collect their mail, but would instead request the mail service, which would dispatch a robot to collect mail. From there, the robots could hand off the mail item amongst themselves while working together to deliver all mail that they have within their network. Individual robots may also carry multiple mail items. Lastly, the mobile app can be improved to have maps of segments of the tunnel and estimates for when the mail will be delivered. Furthermore, the implementation of delivery alerts should be completed, and provisions for the safety of mail should be made.

Acknowledgement

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), [funding reference number 518212].

Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), [numéro de référence 518212].

References

[1] Francesco Alzetta & Paolo Giorgini (2019): *Towards a Real-Time BDI Model for ROS 2*. In Federico Bergenti & Stefania Monica, editors: *Proceedings of the 20th Workshop "From Objects to Agents", Parma, Italy, June 26th-28th, 2019*, *CEUR Workshop Proceedings* 2404, CEUR-WS.org, pp. 1–7. Available at `http://ceur-ws.org/Vol-2404/paper01.pdf`.

[2] Rafael H. Bordini, Jomi Fred Hübner & Michael Wooldridge (2007): *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, Inc., USA, doi:10.1002/9780470061848.

[3] Michael Bratman (1987): *Intention, plans, and practical reason*. 10, Harvard University Press, doi:10.2307/2185304.

[4] R. Brooks (1986): *A robust layered control system for a mobile robot*. *IEEE Journal on Robotics and Automation* 2(1), pp. 14–23, doi:10.1109/JRA.1986.1087032.

[5] Ian Calaça, Tabajara Krausburg & Rafael Cauê Cardoso: *JROS*. `https://github.com/smart-pucrs/JROS`.

[6] Hui-Qing Chong, Ah-Hwee Tan & Gee Wah Ng (2009): *Integrated cognitive architectures: a survey*. *Artificial Intelligence Review* 28, pp. 103–130, doi:10.1007/s10462-009-9094-9.

[7] Alan Davoust, Patrick Gavigan, Cristina Ruiz-Martin, Guillermo Trabes, Babak Esfandiari, Gabriel Wainer & Jeremy James: *Simulated Autonomous Vehicle Infrastructure*. `https://github.com/NMAI-lab/SAVI`. Accessed: 2019-02-19.

[8] Alan Davoust, Patrick Gavigan, Cristina Ruiz-Martin, Guillermo Trabes, Babak Esfandiari, Gabriel Wainer & Jeremy James (2019): *An Architecture for Integrating BDI Agents with a Simulation Environment*. In: *Preproceedings of the 7th International Workshop on Engineering Multi-Agnet Systems (EMAS 2019)*, p. np. Available at `https://cgi.csc.liv.ac.uk/~lad/emas2019/accepted/EMAS2019_paper_22.pdf`.

[9] ElDivinCodino: *ROS2BDI*. `https://github.com/ElDivinCodino/ROS2BDI`. Accessed: 2020-06-29.

[10] Loris Fichera, Fabrizio Messina, Giuseppe Pappalardo & Corrado Santoro (2017): *A Python framework for programming autonomous robots using a declarative approach*. *Science of Computer Programming* 139, pp. 36 – 55, doi:10.1016/j.scico.2017.01.003. Available at `http://www.sciencedirect.com/science/article/pii/S0167642317300242`.

[11] Patrick Gavigan: *savi_ros_bdi*. `https://github.com/NMAI-lab/savi_ros_bdi`. Accessed: 2020-03-09.

[12] Patrick Gavigan & Chidiebere Onyedinma: *saviRoomba*. `https://github.com/NMAI-lab/saviRoomba`. Accessed: 2020-05-09.

[13] Jomi F. Hübner & Rafael H. Bordini: *Jason: a Java-based interpreter for an extended version of AgentSpeak*. `http://jason.sourceforge.net`. Accessed: 2019-02-16.

[14] iRobot: *iRobot Create2 Open Interface (OI) Specification based on the iRobot Roomba 600*. `https://www.irobotweb.com/-/media/MainSite/Files/About/STEM/Create/2018-07-19_iRobot_Roomba_600_Open_Interface_Spec.pdf`. Accessed: 2020-03-08.

[15] Samin Karim & Clint Heinze (2005): *Experiences with the Design and Implementation of an Agent-based Autonomous UAV Controller*. In: *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05, ACM, New York, NY, USA, pp. 19–26, doi:10.1145/1082473.1082799.

[16] Nilson Mori Lazarin & Carlos Eduardo Pantoja (2015): *A robotic-agent platform for embedding software agents using raspberry pi and arduino boards*. *9th Software Agents, Environments and Applications School*.

[17] logitech: *c920-pro-hd-webcam-refresh.png*. `https://assets.logitech.com/assets/65478/3/c920-pro-hd-webcam-refresh.png`. Accessed: 2020-03-09.

[18] Marcelo S. Menegol, Jomi F. Hübner & Leandro B. Becker (2018): *Evaluation of Multi-agent Coordination on Embedded Systems*. In Yves Demazeau, Bo An, Javier Bajo & Antonio Fernández-Caballero, editors: *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection*, Springer International Publishing, Cham, pp. 212–223, doi:10.1007/11775331_5.

[19] Marcelo Sousa Menegol: *UAVExperiments*. `https://github.com/msmenegol/UAVExperiments`. Accessed: 2019-05-24.

[20] Felipe Meneguzzi & Rodrigo Wesz: *Jason ROS Releases*. `https://github.com/lsa-pucrs/jason-ros-releases`. Accessed: 2019-07-17.

[21] Márcio Godoy Morais: *rason*. `https://github.com/mgodoymorais/rason/tree/master/jason_ros`. Accessed: 2019-07-17.

[22] NA: *Eurobot: International Students Robotic Contest*. `http://www.eurobot.org/`. Accessed: 2019-07-15.

[23] NA: *iRobot Create Cable Pinout*. `https://www.irobot.com/-/media/mainsite/pdfs/about/stem/create/create2-cablepinout2018.ashx`. Accessed: 2019-05-08.

[24] NA: *JaCaMo Project*. `http://jacamo.sourceforge.net/`. Accessed: 2019-05-16.

[25] NA: *rosJava*. `http://wiki.ros.org/rosjava`. Accessed: 2020-03-09.

[26] NA: *vooAgente4Wp*. `https://drive.google.com/file/d/0B7EcHgES6He8VEtwR0xPZjdBbk0/view`. Accessed: 2019-05-08.

[27] Chidiebere Onyedinma: *SAVI_Roomba_App*. `https://github.com/NMAI-lab/SAVI_Roomba_App`. Accessed: 2020-05-09.

[28] Open Source Robotics Foundation: *ROS*. `https://www.ros.org/`. Accessed: 2019-05-27.

[29] Carlos Eduardo Pantoja, Márcio Fernando Stabile, Nilson Mori Lazarin & Jaime Simão Sichman (2016): *ARGO: An Extended Jason Architecture that Facilitates Embedded Robotic Agents Programming*. In Matteo Baldoni, Jörg P. Müller, Ingrid Nunes & Rym Zalila-Wenkstern, editors: *Engineering Multi-Agent Systems*, Springer International Publishing, Cham, pp. 136–155, doi:10.1007/978-3-642-38700-5_4.

[30] Jacob Perron: *create_autonomy*. `http://wiki.ros.org/create_autonomy`. Accessed: 2020-03-08.

[31] Gustavo Rezende: *MAS-UAV*. `https://github.com/Rezenders/MAS-UAV`. Accessed: 2019-05-24.

[32] Gustavo Rezende & Jomi F. Hubner: *Jason-ROS*. `https://github.com/jason-lang/jason-ros`. Accessed: 2019-05-24.

[33] P. Wallis, R. Ronnquist, D. Jarvis & A. Lucas (2002): *The automated wingman - Using JACK intelligent agents for unmanned autonomous vehicles*. In: *Proceedings, IEEE Aerospace Conference*, 5, pp. 5–5, doi:10.1109/AERO.2002.1035444.

[34] Michael Wooldridge (2009): *An Introduction to MultiAgent Systems*, 2nd edition. Wiley Publishing.