electronics

MDPI

Article

Agent in a Box: A Framework for Autonomous Mobile Robots with Beliefs, Desires, and Intentions

Patrick Gavigan *,† and Babak Esfandiari *,†

Department of Systems and Computer Engineering, Carleton University, Ottawa, ON K1S 5B6, Canada
* Correspondence: patrickgavigan@sce.carleton.ca (P.G.); babak@sce.carleton.ca (B.E.)
† These authors contributed equally to this work.

Abstract: This paper provides the Agent in a Box for developing autonomous mobile robots using Belief-Desire-Intention (BDI) agents. This framework provides the means of connecting the agent reasoning system to the environment, using the Robot Operating System (ROS), in a way that is flexible to a variety of application domains which use different sensors and actuators. It also provides the needed customisation to the agent's reasoner for ensuring that the agent's behaviours are properly prioritised. Behaviours which are common to all mobile robots, such as for navigation and resource management, are provided. This allows developers for specific application domains to focus on domain-specific code. Agents implemented using this approach are rational, mission capable, safety conscious, fuel autonomous, and understandable. This method was used for demonstrating the capability of BDI agents to control robots for a variety of application domains. These included simple grid environments, a simulated autonomous car, and a prototype mail delivery robot. From these case studies, the approach was demonstrated as capable of controlling the robots in the application domains. It also reduced the development burden needed for applying the approach to a specific robot.

Keywords: Belief-Desire-Intention (BDI); Jason; Robot Operating System (ROS); robotic agents

check for updates

Citation: Gavigan, P.; Esfandiari, B. Agent in a Box: A Framework for Autonomous Mobile Robots with Beliefs, Desires, and Intentions. *Electronics* **2021**, *10*, 2136. https://doi.org/10.3390/electronics10172136

Academic Editors: Angel García-Olaya and Andrea Orlandini

Received: 27 July 2021
Accepted: 27 August 2021
Published: 2 September 2021

1. Introduction

The field of autonomous mobile robotics has become an area of significant development. These technologies have a tremendous potential to impact and change many aspects of society, with application domains including defence, security, infrastructure inspection, passenger travel, freight, and mail delivery. This paper provides the Agent in a Box for developing autonomous mobile robots using Beliefs-Desires-Intentions (BDI) agents.

An autonomous agent can be defined as a system that pursues its own agenda, affecting what it senses in the future, by sensing the environment and acting on it over time [1]. Autonomous agents should be 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, by only accepting and working toward goals that it believes are achievable. Developers of autonomous mobile robotics are interested in how to program these agents while guaranteeing reliability and resilience.

Agents implemented using BDI are rational, with knowledge, behaviours, and goals defined using a declarative, symbolic programming language. These agents maintain a set of beliefs about themselves and their environment, and have desires, or goals, that they need to achieve. To do this they have a set of plans, behaviour programs, that they can choose to execute based on their context. When an agent selects a plan to execute, it has set its intention. Assuming a correct symbolic representation of the environment and the agent's behaviours, decisions are made using rational reasoning. This means that the agent's actions will always move the agent closer to its goals. The BDI approach was first developed in the field of cognitive science by Bratman in the 1980s as a means of modelling human agency [2]. This model for defining agency was adapted for use in computer science using modal logic to underpin rational decision-making [3]. Bordini et al. [4] highlighted that BDI agents have distinct advantages in that they provide the capability of interleaving plan selection and action selection, and can recover gracefully from failure. Bordini et al. explained that BDI's method of providing intention-driven behaviour makes it easier to understand what the agent is doing, why it is doing it and what the agent is going to do next. Bordini et al. also highlighted several limitations, such as the need for a developer to define the agent's plans due to the lack of support for generating plans on its own and the needed effort to adapt the default BDI reasoning cycle for advanced applications. There are many examples of BDI agents being used. For example, they provide a popular method for competitors in the Multi-Agent Programming Contest [5]. In addition to examples of BDI agents being used for controlling mobile robots and drones, which will be discussed in Section 3, Hofmann et al. demonstrated the how a BDI agent can be used for image analysis in remote sensing [6].

Naturally, there are many other ways the software for autonomous mobile robotic systems can be developed. Software developers who prefer to work with more popular procedural or object-oriented languages, such as C, C++, Python, Java, etc., may suggest that they could develop software using these approaches which could also provide the desired features. In short, they would be correct. Alternatively, declarative approaches come with the strengths which underpin logic-based systems, including guarantees with respect to the agent behaving rationally built into the approach. Custom-made software would not come with such a guarantee without additional effort to validate this property. With these apparent advantages of using BDI, and declarative programming in general, it would be expected that this would be a popular way of implementing autonomous mobile robots; however, as will be seen in the state-of-the-art, there has been a limited amount of work in this area.

There is significant excitement in the area of machine learning, especially in reinforcement learning and deep learning. Although machine learning approaches have very enticing attributes, they come with some major drawbacks. For example, deep learning requires massive amounts of data, in the form of example runs, used for training the agent. Furthermore, this method is a black box approach, meaning that the internals of the method are not setup in a way that is easily understood by humans, complicating the development of such models. This work seeks a simpler alternative by using BDI agents instead.

There are several implementations of the BDI paradigm. One of the most popular is Jason [1,7], which uses the AgentSpeak language [8] for defining the agent's beliefs, rules, and plans. This work uses Jason and the AgentSpeak language. This paper demonstrates how the mission of an agent-based mobile robot could be defined as a plan that can be monitored, executed, suspended, and resumed depending on changing contexts. It will further demonstrate that there are generic aspects to the missions of mobile robots which can be provided by the Agent in a Box. This includes the handling of mission management, navigation, safety, and energy autonomy. It also includes the prioritisation of the agent's behaviour and how the agent connects to the environment, using Robot Operating System (ROS). This paper will demonstrate how this framework can be used for several different applications, different robots equipped with different sensors and actuators, highlighting the flexibility and performance of this approach. In doing so, the components that a domain-specific developer needs to provide will be identified.

2. Background

The key tools and methods used for setting up the Agent in a Box are explained in this section. The definition of a software framework is provided in Section 2.1. With the concept of a software framework defined, BDI agents are explained in Section 2.2. This includes a high-level overview of the BDI paradigm, the Jason reasoner, and the AgentSpeak language used for defining an agent's behaviour. The Agent in a Box uses ROS, a popular system for implementing distributed robots for connecting the agent reasoner to the robot. ROS is discussed in Section 2.3. Last, the Subsumption Architecture is discussed in Section 2.4. This architecture provided a useful inspiration for behaviour prioritisation included in the Agent in a Box.

2.1. Software Frameworks

A software framework can be defined as being a "reusable design of a program or part of a program" [9,10]. It is a mixture of both concrete and abstract software, promising users "higher productivity and shorter time-to-market through design and code reuse" [11]. The defining characteristic of a software framework is the concept of "inversion of control" [12]. In software development, without the use of a software framework, the program's thread of control is specified by the developer. Therefore, programs will include library calls which are called by the code written by the developer. By contrast, software frameworks serve as "extensible skeletons" [12]. To use a software framework, a developer must provide functions for the software framework to call. This customises the software framework's generic algorithms to specific applications. The control of the application has been inverted: the software framework is in control of when these functions are called. The developer does not choose when the methods are called, they only specify what they do. In order to develop software in this way, the software framework imposes a structure on the developer. By following the software framework's structure, the developer can take advantage of the software framework's features. The use of the software framework should eliminate any duplication of effort between different applications, as the software framework handles the generic aspects of the application.

2.2. BDI Agents, Jason, and AgentSpeak

The Beliefs-Desires-Intentions (BDI) approach was first introduced by Bratman as a means of modelling the cognitive processes which give rise to agency [2]. Shoham provided a logical foundation for rational reasoning using Agent-Oriented Programming (AOP) [13]. Shoham defined the agent as having a mental state, consisting of beliefs, decisions, capabilities, and obligations. A complete AOP system includes a restricted formal language with syntax and semantics for describing the agent's mental state using modalities such as belief and commitment, an interpreted programming language in which to define and program agents, and an "Agentifier" for converting neutral devices into programmable agents. Building from this, a rational agent should only commit to the goals it believes that it can achieve. It should also give up its goal if the goal has become impossible to achieve, or if the motivation for that goal no longer exists [1]. Rao and George [3] built on this foundation, developing BDI as a type of symbolic Artificial Intelligence (AI). 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. By building off these foundations, BDI agents can provide the properties of rational reasoning to software agents.

A BDI agent makes rational decisions through the use of its reasoning cycle, starting with the agent perceiving the task environment and receiving messages. From this information, the agent can then decide on a plan suitable to the context provided by those perceptions, the agent's own beliefs, messages received, and desires. Once this plan has been selected, it sets this plan as an intention for itself. These plans can include updating the belief base, sending messages to other agents, and taking some action. As the agent continues to repeat its reasoning cycle, it can reassess the applicability of its intentions as it perceives the environment, dropping intentions that are no longer applicable [1,7]. There have been several implementations of BDI. Examples include JACK [14], Jason [1,7,15], and LightJason [16,17]. Jason has also been included in the JaCaMo framework, which includes Jason for AOP in BDI, Cartago for programming environment artefacts, and Moise for setting up multi-agent organisations [18]. Gwendolen is another BDI framework which emphasises the use of model checking for verifiable agents [19]. This paper uses the Jason implementation of BDI.

Agents developed for BDI systems using Jason are programmed using a language called AgentSpeak [1,8]. AgentSpeak is a post-declarative programming language that bears some similarities to Prolog [1]. It is said to be post-declarative, as it is interpreted as part of the agent reasoning cycle, unlike how a declarative language, such as Prolog, is typically executed [1]. The syntax provides a means for specifying initial beliefs for the agent to have, rules that can be applied for reasoning and plans that can be executed. The Extended Backus–Naur form (EBNF) description of AgentSpeak can be found in Appendix A.1 of the Jason textbook [1].

Listing 1: AgentSpeak Plan Syntax.

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

In general, AgentSpeak plans are written using the syntax shown in Listing 1. Each of the terms in the listing are logical literals forming beliefs, goals, etc. A triggering event is the addition or deletion of a belief, achievement goal, or a test goal. To differentiate goals from beliefs, achievement goals begin with an exclamation mark (!) and test goals begin with a question mark (?). 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 for the plan to be applicable based on the state of the agent's belief base. The context is made up of a logical sentence that can use both beliefs as well as rules, which are further discussed below. 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. These actions can either have some effect on the environment or can be internal actions, which are Java functions that the agent can call, typically used for performing a calculation. Optionally, plans can be provided with user-specified names and annotations on the preceding line. The syntax for these names is simply `@name`. A plan name, belief, or triggering event can also be provided with annotation, a type of metadata which can be used for providing additional information to the reasoner. For example, a plan annotated with `[atomic]` after its plan name signals to the reasoner that it must be run to completion; it cannot be interrupted. As the AgentSpeak language follows the AOP paradigm, as long as the symbolic representation of the environment is correct, the plan's context is correct, and the plan's body is correct, the plan selected by the agent reasoning system will move the agent closer to its goal [1].

In addition to plans, the AgentSpeak language supports the definition of rules. These rules can be used as part of the context for plans in the plan base, and are structured as logical implications as shown in Listing 2. In this listing, the `conclusion` is implied by the `condition` [1].

Listing 2: AgentSpeak Rule Syntax.

```
conclusion :- condition.
```

2.3. Robot Operating System (ROS)

ROS is a package for developing software for robotic applications [20]. The Agent in a Box used ROS for connecting the BDI agent to the environment. ROS operates using a tuple-space architecture where software nodes publish and subscribe to topics using socket-based communications instead of communicating with other nodes directly. This removes the need for developers of individual nodes to concern themselves with which nodes they are interacting with, they need only concern themselves with the topics that they use. 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.

ROS has an active community supporting a variety of robotic platforms, sensors, and actuators. For example, there are nodes which provide image processing using OpenCV. Additionally, the Point Cloud Library [21,22] can be used for processing 3D data from sensors such as a laser imaging, detection, and ranging (LIDAR) sensor. There is also integration with the MoveIt! library [23,24] for planning algorithms as well as tools for industrial robotics with ROS-Industrial. Simultaneous Localisation and Mapping (SLAM)-based navigation support is available as part of the ROS navigation stack [25–27]. Truong and Ngo developed a a complimentary navigation node for ROS that adds social awareness to a robot's movement [28]. By building robotic applications that are compatible with ROS, developers enable their applications to be compatible with other devices and software nodes supported by the community. 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 system developers to use those modules without concern as to how the nodes are implemented.

2.4. Subsumption Architecture

The reactive Subsumption Architecture was proposed by Brooks [29,30]. Brooks claimed that explicit representations or abstract reasoning, required by other types of symbolic AI, were not needed. Instead, the intelligence of the agent can be an emergent property of the agent's behaviour. This is implemented using layers of situation and action pairs where lower layers have higher priority and can inhibit higher level layers. Sensor inputs are provided to each of the layers of the architecture. Each layer provides an action for execution as long as it is able. The action provided by the lowest level layer subsumes the actions provided by the higher level layers and takes precedence. For example, consider an autonomous driving application where a collision avoidance action would be required to subsume regular navigation to a final destination. The Subsumption Architecture provides the inspiration for the behaviour prioritisation method used by the Agent in a Box.

3. State of the Art

In this section, the state-of-the-art in application of BDI for autonomous mobile agents with a concentration on robotic applications is considered. This differs from the more common focus of using BDI, or AOP in general, for simulated problems used in grid-based environments, as can be seen in several iterations of the Multi-Agent Programming Contest [5]. Although work in grid-based environments can produce valuable insight into the properties of software agents and Multi-Agent System (MAS), the works surveyed are applications where the focus is on the development of a BDI agent for a real-world environment, or a simulated environment where a transition to the real world is an objective are the focus. Although these surveyed works have demonstrated the feasibility of using BDI for mobile robots, generally they were found to use ad hoc approaches to implement their agents. A concern of computational performance of the agent was raised by several of the works. Also provided in this section are works which provided the beginnings of an infrastructure for autonomous mobile robots using BDI reasoning.

The first surveyed work is from when the Australian military performed trials using a BDI agent to control a Codarra Avatar fixed wing Unmanned Aerial Vehicle (UAV) [31,32]. Although this project demonstrated that their BDI agent, implemented with JACK, could fly the UAV, the project did not demonstrate flexibility for different sensors or actuators. Although the agent did perform some basic navigation, this seemed to be limited to using an autopilot to direct the aircraft to specified locations.

A more recent example of a BDI agent flying a UAV was the JaCaMo UAV, which used a Jason BDI agent to control the drone using ROS [18,33–36]. Although they did perform successful flights with their UAV, they did not demonstrate any flexibility for using different sensors and actuators or different platforms. They also did not provide any behaviour framework or prioritisation scheme for the agent's behaviour. There was no mention of obstacle avoidance or resource management. The agent's navigation was limited to travelling to locations that were hard-coded into the agent's beliefs.

Another example of Jason being connected to ROS is the JaCaROS project which used Jason agents connected to robots simulated in Gazebo [37]. In this case, they demonstrated that they could control a simulated turtle bot in different scenarios with various simulated sensors and actuators. The works' limitation was that the environments were all focused on the use of a simple tiled floor environment, with the lines between the tile squares serving as paths between nodes that the agent could move between, limiting the realism of the environment. Although there was no mention of any behaviour framework, behaviour prioritisation, or obstacle avoidance, they did demonstrate battery management behaviour. The robot would monitor a simulated battery and would recharge it as needed. Although some of the test scenarios involved navigation, the agent only needed to move between corners of a square on a grid. This was not considered a significant demonstration of navigation behaviour.

Python RObotic Framework for dEsigning sTrAtegies (PROFETA) used a Python-implemented BDI agent for controlling mobile robots [38–40]. They demonstrated their work for both the Eurobot challenge [39] and for a logistics robot. This means that this approach has been demonstrated with different sensors and actuators and on two different platforms. Navigation using Dijkstra's algorithm and path planning were the primary roles of the agent. The agent also had to perform obstacle avoidance due to the presence of uncooperative agents operating in the same environment. KC and Chodorowski further demonstrated the use of PROFETA with ROS for a proactive social communication robot which used OpenCV to detect the presence of people to determine if they may need assistance from the robot [41]. There was no mention of PROFETA providing any behaviour framework or prioritisation.

The ARGO project explored using a Jason agent for controlling a small robotic car using their Javino library [42,43]. The car included sensors for distance, light, and temperature and a motor for driving the car. There was no discussion of this approach being used with any other sensors or platforms. The ARGO experiments focused on the performance of the agent, specifically if it could stop before hitting a wall. The concern was that this obstacle avoidance behaviour may not be triggered in time if the agent was overly burdened by updates from irrelevant sensor data. There was no discussion of behaviour frameworks, prioritisation, resource management, or navigation.

The final example is the concept of "Abstraction Engines", which provides the means for a robot's continuous sensors and actuators to be abstracted from the agent so that it does not get overwhelmed with sensor data [44,45]. This approach has been demonstrated using both the GWENDOLEN and Jason BDI reasoners, and has been demonstrated with a number of robot platforms with various sensors, both with and without the use of ROS. The main focus of this approach was how the sensors and actuators should be abstracted from the agent so that the reasoner does not get bogged down by continuous sensor data. Although this provided a significant strength to the approach, there was very little discussion of how the agent's behaviour should be designed.

Table 1 provides a summary of the features of the various projects. The focus was on how the method connected the agent to the environment and the agent's behaviour. In considering the connection to the environment, there is an interest in agents which demonstrate flexibility with different sensors and actuators, ideally demonstrating their use with multiple platforms. As mentioned in Section 2.3, ROS is a popular package for developing distributed robotics. As such, methods that use ROS, and by extension have the potential of being used with other ROS packages, are of interest. The choice of BDI implementation used was considered. Specifically, there was interest in whether the method used Jason rather than some other BDI reasoner. As Jason follows the strict properties of AOP, enforced with the AgentSpeak language and the reasoner, agents that take advantage of Jason's enforcement of these properties were preferred. Next, the agent's behaviour was considered, specifically whether the agent's behaviour, written in AgentSpeak or some other similar language, was developed with the use of an agent behaviour framework which provided some generic behaviour for different types of mobile agents. Of particular interest is whether the approach provided support for prioritising the agent's behaviour, for example, ensuring that plans for obstacle avoidance were considered at the highest priority. This goes beyond the agent simply demonstrating that such behaviour is possible, which is necessary is that the agent actively prioritise such behaviours. The final element considered was the specific behaviour that was demonstrated by these projects, such as obstacle avoidance, resource management (such as battery recharging), and navigation; features that are common to a wide variety of mobile robots. Methods that fully provided these features were given a ✓ in Table 1; otherwise, they were given a ✗.

Table 1. Summary of the state-of-the-art.

		Codarra Avatar	**JaCaMo UAV**	**JaCaROS**	**PROFETA**	**ARGO**	**Abstraction Engines**
Connecting to the Environment	Flexible for different sensors and actuators	✗	✗	✓	✓	✗	✓
	Multiple platforms	✗	✗	✗	✓	✗	✓
	Controlled mobile robot	✓	✓	✓	✓	✓	✓
	Uses popular robotics architecture (ex: ROS)	✗	✓	✓	✗	✗	✓
Agent Behaviour	Uses a popular BDI reasoner (ex: Jason)	✗	✓	✓	✗	✓	✓
	Behaviour framework	✗	✗	✗	✗	✗	✗
	Behaviour prioritisation	✗	✗	✗	✗	✗	✗
	Obstacle avoidance	✗	✗	✗	✓	✓	✗
	Resource management	✗	✗	✓	✗	✗	✗
	Navigation	✓	✗	✗	✓	✗	✗

The work surveyed was successful at demonstrating the feasibility of using BDI agents to control mobile robots. In fact, every one of the experiments demonstrated this to some degree. Unfortunately, this was the only feature that each of the works surveyed was able to demonstrate, and none of them demonstrated all of the features of interest, as seen in Table 1. Missing from all of the surveyed works was any discussion of a framework for the agent's behaviour. Such a framework would have provided the agent with a generic behaviour that could be mapped to mission-specific behaviour for different platforms; also missing was any discussion of the prioritisation of the agent's behaviour. The default selection method in Jason, for example, is to select the first applicable plan for the first event in the event queue. This means that the agent's plan selection would have been tied to the order in which the plans were loaded in the plan base, rather than deliberately selected based on what the most appropriate or highest priority plan for any given circumstance should be. Despite the lack of detail with respect to the agent behaviour, the Abstraction Engines approach did demonstrate all the desired features for connecting to the environment, specifically for handling the sensors and actuators. In general, it was found that many of the surveyed projects used ad hoc approaches for their implementation. Additionally observed in the works surveyed was the use of hard coding, for example, the use of hard coded locations that the agent should visit. This seems to have been the result of the limitations of sensors. Most of the projects seemed focused on sensors provided by some sort of autopilot component which can provide attitude, position, and velocity data. There was less emphasis, however, on payload sensors which would be used for mission activities.

4. Materials and Methods

For the Agent in a Box to be useful for the development of a variety of mobile robots, there are a number of requirements that must be considered. The Agent in a Box must provide useful behaviours for controlling a variety of mobile robots. The Agent in a Box must also provide the means for the reasoner to appropriately select which behaviours to set as intentions while balancing the concerns for safety, health, and the agent's domain-specific mission. Last, the Agent in a Box needs a generic method for connecting to the environment, enabling it to be used with a variety of sensors and actuators for controlling a variety of mobile robots. The Agent in a Box should also use tools that are familiar to developers. In this case, Agent in a Box uses Jason, a popular BDI reasoning system which uses AgentSpeak. It also uses ROS, a popular framework for the development of distributed robotics.

Regardless of the mobile robot, there is a need for the robot to receive updates from a set of sensors and to be able to control a set of actuators. All of these application domains need some means of sensing the position of the robot, detecting possible collision risks, and of controlling movement. Different domains use different types of sensors and actuators, meaning that there needs to be flexibility in how the Agent in a Box connects to them, allowing different devices can be connected easily. Furthermore, it is desirable to use tools that are familiar to the robotics community. Therefore, the Agent in a Box connects the agent reasoner to the environment using ROS, a tool that should be familiar to a wide community of robotics developers and has a large library of other nodes that may be useful for specific domains. The architecture of the Agent in a Box is discussed in Section 4.1.

With the connection between the agent and the environment handled, it is necessary to consider the agent's reasoning system and how it prioritises the agent's behaviour. Jason selects which plan should be set as an intention by first selecting which of the triggering events will be used, and then selecting which of the applicable options will be used. By default, the first triggering event is chosen, as is the first applicable plan option for that event. For relatively simple agents, where the order of the plans in the AgentSpeak program can be easily maintained, this default selection may be sufficient. Simpler agents may also have plans with mutually exclusive context checks, ensuring that there is only one applicable plan available for the agent to select. In the case of relatively simple behaviour designs, Jason's default event and option selection functions, which choose the first event and option available, may be sufficient. In the case of the Agent in a Box however, where there are behaviours provided by the framework as well as by the developer of the agent for a specific application domain, there is no guarantee that the order of the behaviours in the plan base will properly reflect the relative priority of these plans. Thus, there is a need to ensure that Jason selects the appropriate plan at any given circumstance. A method for selecting the plan selection is discussed in more detail in Section 4.2.

The final aspect of the Agent in a Box is the behaviour of the agent itself. The behaviour of BDI agents is defined by a set of plans written in AgentSpeak. The goal with the Agent in a Box is to provide generic plans which handle the generic aspects of controlling a mobile robot. The robot's missions generally include some combination of moving to one or more destination locations and performing some actions at those locations. A mobile robot should perform this mission while avoiding obstacles while also ensuring that it has sufficient resources, such as a charged battery, while working. On its route, if it encounters a blocked path, it should update its map and find an alternate route. These features are common to any mobile robot. This means that the Agent in a Box should provide the common elements for the agent to manage its mission, navigate to a destination, update its map, manage its resources, avoid obstacles, and manoeuvrer through the environment. The Agent in a Box provides the needed support for ensuring that the domain-specific plans, beliefs, and rules that a developer provides are properly mapped to the framework's generic plans. This structure also ensures that the behaviour is properly prioritised by the reasoning system. By doing so, the developer makes use of the generic behaviour for their specific application domain. This paper provides the details of the behaviour framework in Section 4.3.

With the Agent in a Box established, including how it connects to the environment, how the behaviour is prioritised, and the generic behaviour defined, it is necessary to map the framework to application domains. In Section 4.4, the details of three case studies where the Agent in a Box was applied are provided: The first is a fairly simple environment: a grid environment. This is followed by a more complex and realistic environment: an autonomous car that was simulated with AirSim. The third environment is a prototype mail delivery robot, implemented using an iRobot Create controlled with a Raspberry Pi. Each section outlines the environments, sensors, and actuators that are available to the agents; the agent's mission parameters; and how the framework is mapped to these domains. Despite the apparent differences in these domains, this paper demonstrates how the Agent in a Box can be mapped to these specific applications.

4.1. Agent in a Box: Connecting to the Environment

The Agent in a Box provides a means for connecting a BDI agent using Jason and the AgentSpeak language to mobile robots. The goal is to provide the agent with a generic method of connecting to the environment using a method that is familiar in the field of mobile robotics. Given its popularity, and the availability of a variety of modules that may be useful to developers, the Agent in a Box uses ROS for connecting the reasoner to the environment. The architecture for connecting the agent to the environment is shown in Figure 1. At a high level, this architecture includes an environment interface, containing the nodes connected to the sensors and actuators; application nodes, which include the user interface and translators for the agent. It also includes the agent reasoner. The details of these components are provided in the following paragraphs.

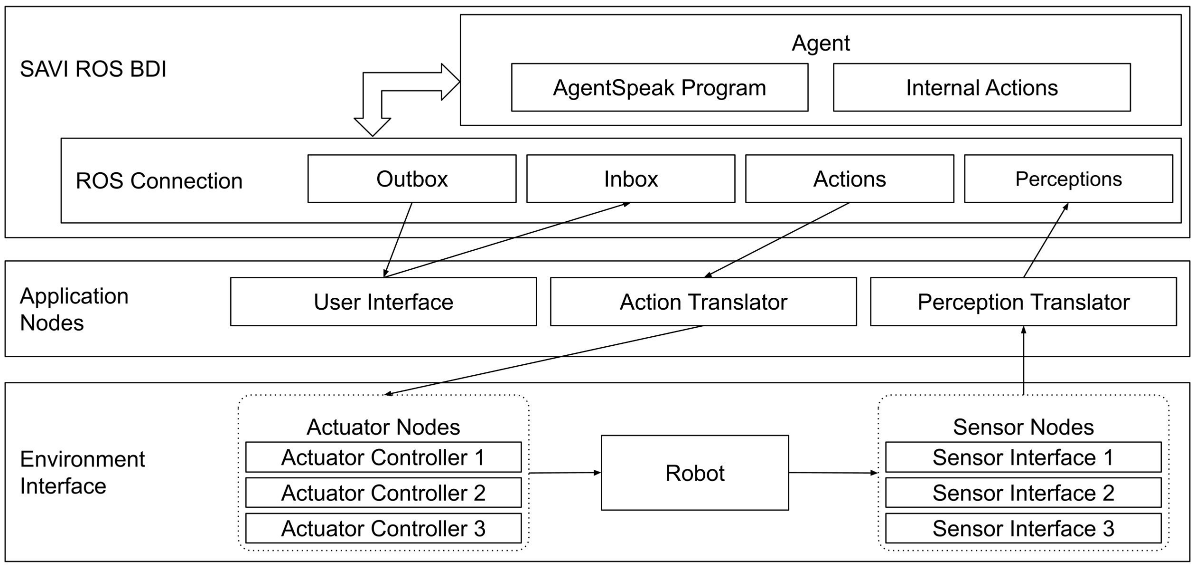


Figure 1. Agent in a Box framework—connecting to the environment.

Davoust et al. [46] identified an *impedance mismatch* problem [47] stemming from the differences in how BDI frameworks connect to their environments, and software simulators that lack the sophistication for modeling BDI agents. *Impedance mismatch* refers to the "conceptual and technical issues faced when integrating components defined using different methodologies, formalisms or tools" [46]. To resolve this problem, Davoust et al. proposed the Simulated Autonomous Vehicle Infrastructure (SAVI) architecture, which connected Jason agents to a custom simulation by decoupling the agent reasoning cycle from the simulation cycle [46]. Recognising that a similar challenge exists for connecting BDI agents to robotic systems, this Agent in a Box architecture uses the SAVI ROS BDI, which built on the solution proposed by the SAVI project, for connecting a Jason BDI agent to ROS. This software is available open source on GitHub [48].

The component of the architecture responsible for connecting the Jason reasoner to ROS is the SAVI ROS BDI node. It provides the necessary abstraction that a BDI agent requires to be suitable for connecting any variety of sensors and actuators. This framework makes the agent available for multiple platforms. SAVI ROS BDI decouples the agent's reasoning cycle from the implementation of the sensor and actuator interfaces. This ROS node subscribes to the `perceptions` and `inbox` topics and publishes to the `actions` and `outbox` topics. The recommended method for the developer to provide the sensor perceptions and monitor for actions is discussed below in the explanation of the environment nodes and the application nodes.

The agent behaviour is implemented as an AgentSpeak program. This program, containing both the framework provided behaviour and the mapping to the application specific behaviour, is interpreted by the Jason reasoner. The agent also has access to a variety of internal actions, primarily used for performing more complex calculations. The internal architecture of this node is shown in Figure 2. It consists of three main components: the agent core, which contains the Jason reasoner; the ROS connectors, separate threads that are responsible for publishing and subscribing to the ROS topics; and the state synchronisation module, which is the mechanism for data sharing between the ROS connectors and the agent core. Running the Agent Core and the ROS connectors in separate threads decouples the communication with ROS from the agent's reasoning cycle. The perception and inbox listeners receive data from their respective ROS topics and provide these data to the state synchronisation module's perception manager and inbox. There, the messages are validated and made available to the agent core, which checks for perceptions and messages at the beginning of each reasoning cycle. Any actions or messages that the agent generates are provided to the action manager and the outbox, in the state synchronisation module. The action and outbox talkers each monitor the state synchronisation module, publishing data on their topics when it becomes available.

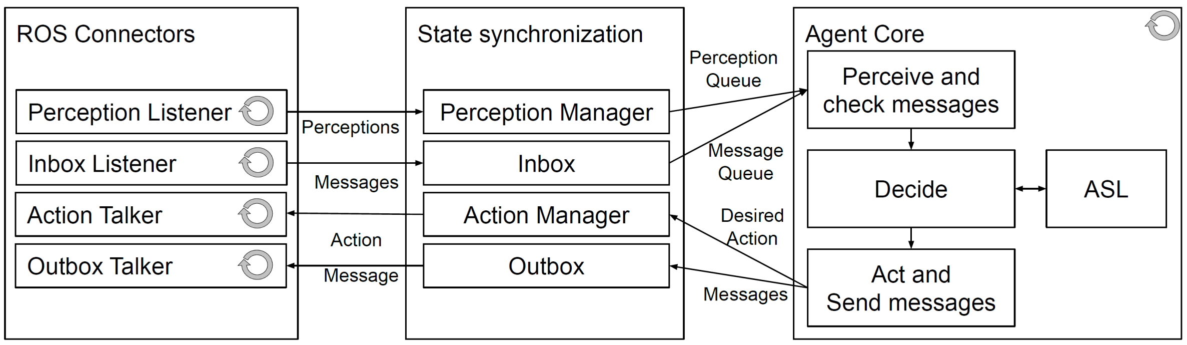


Figure 2. SAVI ROS BDI internal architecture.

The role of the environment interface is to connect the robot's sensors and actuators to ROS so that the agent can perceive the sensor data and control the actuators. This can involve the use of third-party nodes, perhaps available for specific hardware, or they could be custom-built nodes. This approach to the environment interface bears some similarity to the approach provided by the "Abstraction Engines" project discussed in the state-of-the-art section [44,45]. In the Agent in a Box, the sensors are encapsulated in a set of individual nodes which publish their data to relevant ROS topics. The reverse was done for the actuators, encapsulating their controllers as ROS nodes which subscribe to relevant topics for their commands and settings. Each of these nodes is intended to be designed for simplicity, abstracting any underlying implementation details of these components from the broader architecture. This design allows sensor and actuator nodes to be used in a plug-and-play fashion, allowing the flexibility to use unique domain-specific sensors and actuators for different applications.

The application nodes are domain-specific nodes which include a user interface and translators for the agent's perceptions and actions. The user interface uses agent communication for providing the agent with goals, by publishing to the `inbox` topic, and monitoring the agent's progress, by subscribing to the agent's updates on the `outbox` topic. The perception translator subscribes to the various sensor data topics and translates the data into a set of predicates. These predicates are the perceptions that will be provided to the agent. The perception translator collects these perceptions and publishes them to the `perceptions` topic. The perceptions for all the sensors are provided in a single message. This is important as the agent may have plans which use data from multiple sensors in their context checks. If these perceptions are not provided together, there is a possibility that these plans may never be applicable. Last, the action translator subscribes to the `actions` topic. Similar to the perceptions, the agent's actions are provided in the form of first-order predicates, containing the name of the action and any relevant parameters. The action translator identifies the provided action and extracts the parameters, publishing them to the appropriate actuator topicfor the relevant actuator node.

4.2. Agent in a Box: Prioritisation of Behaviour

With the architecture for the agent to connect to the robot's sensors and actuators defined, the focus shifts to how the agent selects which plan will be set as intentions. As was discussed at the beginning of this section, mobile robotic agents are expected to manage a mission of some sort, navigate to a destination, and move through the environment. While doing this, the agent is expected to maintain the safety of itself and others around it by avoiding obstacles. It also needs to maintain its health by managing its consumable resources. In the event that an agent finds an inconsistency between its map and its observations of the environment, it needs to be able to update its map.

In considering these types of activities, it is proposed that the agent's plans have a relative priority. For example, the agent should, as its top priority, maintain safety by avoiding obstacles before executing any other behaviour. Inspired by the Subsumption Architecture, which was discussed in Section 2.4, the Agent in a Box's agent architecture uses a prioritisation scheme with respect to the selection of which applicable plan will be added to the agent's intentions. As AgentSpeak provides a full programming language, subsumption behaviours can be implemented, but also other behaviours if needed. The provided prioritisation scheme for the framework is provided in Figure 3. The prioritisation focuses on having belief-triggered plans for safety, such as for obstacle avoidance, as the highest priority. This is followed by plans which maintain the health of the agent, such as the resource management behaviours, and map update behaviours. Lower priorities are the achievement-triggered plans for mission management, navigation, and movement of the agent. This relative prioritisation is implemented in a method that can be overridden by a developer if necessary.

In considering how to incorporate the behaviour prioritisation into Jason's reasoning framework, it is first necessary to revisit Jason's reasoning cycle in more detail. Shown in Figure 4, the focus is on how the reasoner determines what plans should be set as intentions. When an event occurs, which can be a change in goals or beliefs, these events are passed to an event selection function S_E, shown in diamond 5 in the figure, highlighted with a red marking. This function selects which event will receive the attention of the reasoner for that cycle. Similar to the event selection function, Jason's default event selection function is to select the first event in the queue. With the event selected, all plans associated with this event are unified and their contexts are checked so that the applicable plans can be identified. Applicable plans are then loaded to the option selection function S_O, shown in diamond 8 in the figure, again highlighted with red. Jason's default option selection is to select the first applicable plan in the queue of applicable plans. This applicable plan is then loaded to the top of the intention stack. [1]

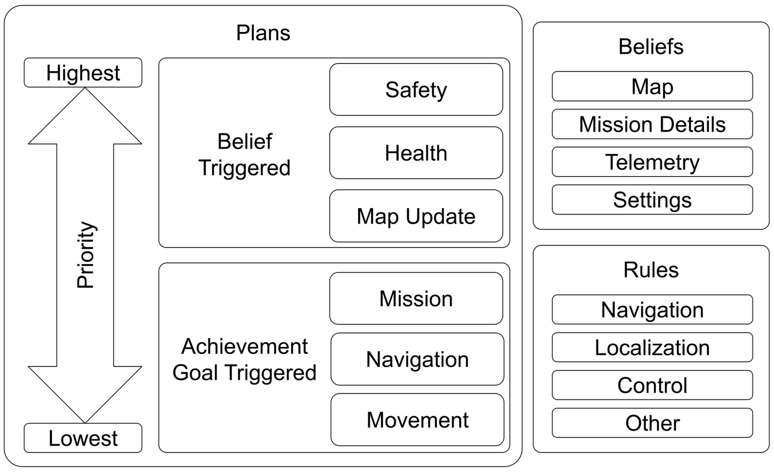


Figure 3. Agent in a Box behaviour prioritisation.

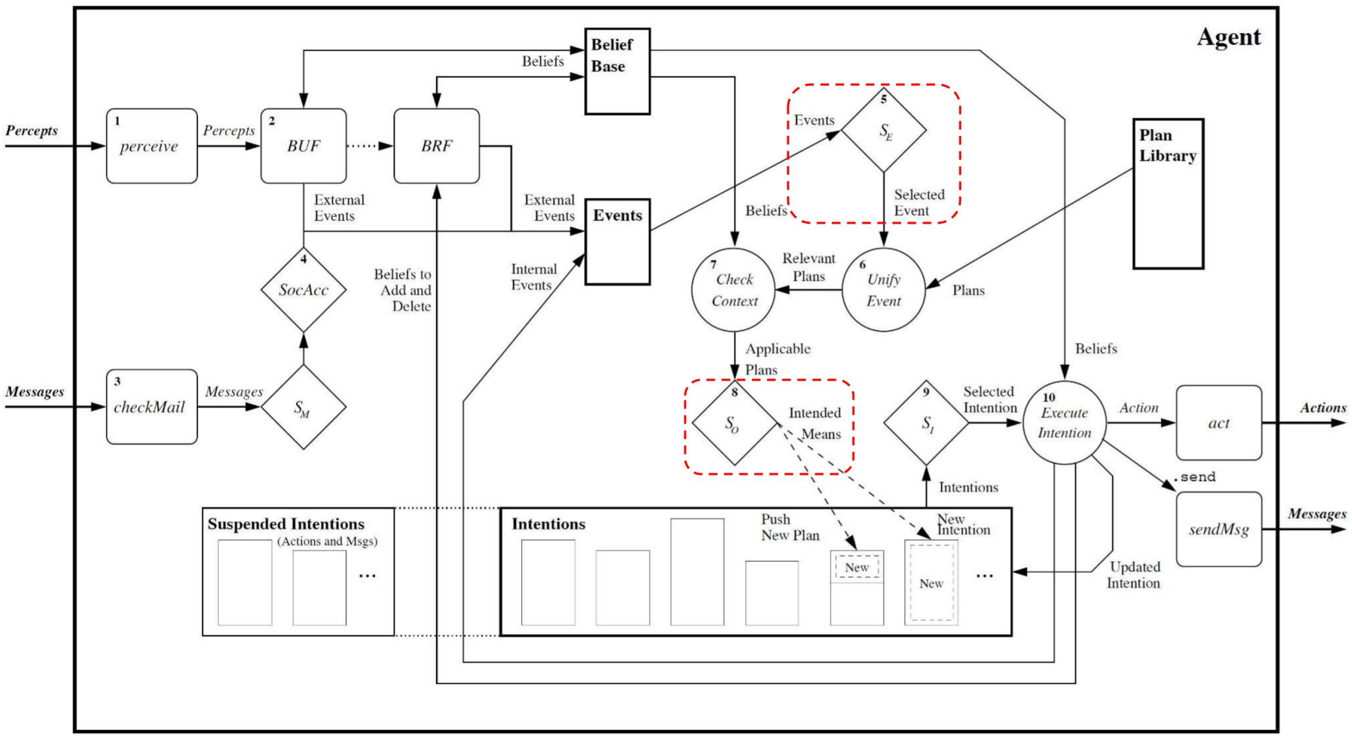


Figure 4. BDI Reasoning Cycle in Jason with Event and Option Selection Highlighted [1].

In the case of relatively simple behaviour designs, Jason's default event and option selection functions, which choose the first event and option available, may be sufficient. In the case of the Agent in a Box, however, where there are behaviours provided by the framework as well as by the developer of the agent for a specific application domain, there is no guarantee that the order of the behaviours in the plan base will properly reflect the relative priority of these plans. This is further complicated by the possibility that a developer may add new behaviours to the agent in subsequent releases of the agent software. Therefore, there is a risk that the developer may inadvertently modify the ordering of the plans in the agent's plan base, causing a dramatic change in the agent's behaviour. To reduce this risk, the Agent in a Box provides the overridden event and option selection functions which prioritise the agent's behaviour. The event selection function must first select the highest priority triggering event, and the option selection function must select the most appropriate plan triggered by that event.

In order for the event selection function to select the highest priority event, it is necessary for this function to have a means of identifying the type of behaviour the event can trigger. One approach could be to provide annotated event triggers. This approach could be used by the event selection function for prioritising and then selecting the event. These annotations could use categories, such as `health` or `mission`, to specify the type of behaviour associated with each event. Unfortunately, there is still a significant downside to this approach: the triggering events need to be annotated, not the plans, meaning that the triggering events for achievement goals and for belief-triggered plans need to be annotated at the source. This could be particularly troubling with respect to the belief triggered plans which use perception-generated beliefs. The perception translator, which was discussed in Section 4.1, would need to generate perceptions with annotations which reflected how the agent would use them. It also means that any achievement goals needed to be annotated when the goal was adopted, which was often in the plan body of other plans. This could mean that some event triggers could be missed in the process of refactoring. Ultimately, the use of trigger annotations was rejected for these reasons.

Instead of using event annotations, the proposed solution is to have a set of beliefs that the agent has with respect to which event triggers are used for which kind of behaviours. This provides a mechanism for the developer of a domain-specific agent to specify what event triggers are associated with what type of behaviour. A sample of these beliefs are shown in Listing 3. In Listing 3, the trigger names for different types of behaviours are identified. These can then be prioritised by the event selection function so that the appropriate event trigger is selected. The terms coloured in blue are Agent in a Box provided beliefs or event triggers and the terms coloured in green are expected or used by the Agent in a Box. Section 4.3 discusses the behaviour associated with the triggers defined by the Agent in a Box or by the developer of an agent for a specific domain.

Listing 3: Behaviour Prioritization Beliefs.

```
safety(pedestrian).
health(resource).
map(mapUpdate).
map(obstacle).
mission(mission).
navigation(navigate).
movement(waypoint).
```

Now that the agent has knowledge of the types of behaviour each event can trigger, the reasoner's event selection function can use these beliefs for selecting the highest priority triggering event for the agent's current circumstance. The implementation of this prioritisation is illustrated in Figure 5. This function is called by the Jason reasoner to select which event to use from a list of events in the agent's current circumstance. It uses a list of `eventPriorities` which specifies the relative priority of the different types of behaviours with their highest priority behaviours listed first. The function uses a set of nested loops to find the highest priority event. The outer loop iterates through the list of priority behaviour types. The next loop level iterates through the event queue, specifically the event functors (the term before the first bracket in a predicate). The following loop level iterates through the belief base looking for beliefs with functors that match the particular behaviour priority level. The function then checks any matching beliefs for terms that match the event trigger. If there is a match, the event is returned, as this is guaranteed to be the highest priority event. If there is not a match, the nested loops continue. If the loops finish without finding an event to return, Jason's default event selection function is used to select the event, protecting this function from failure.

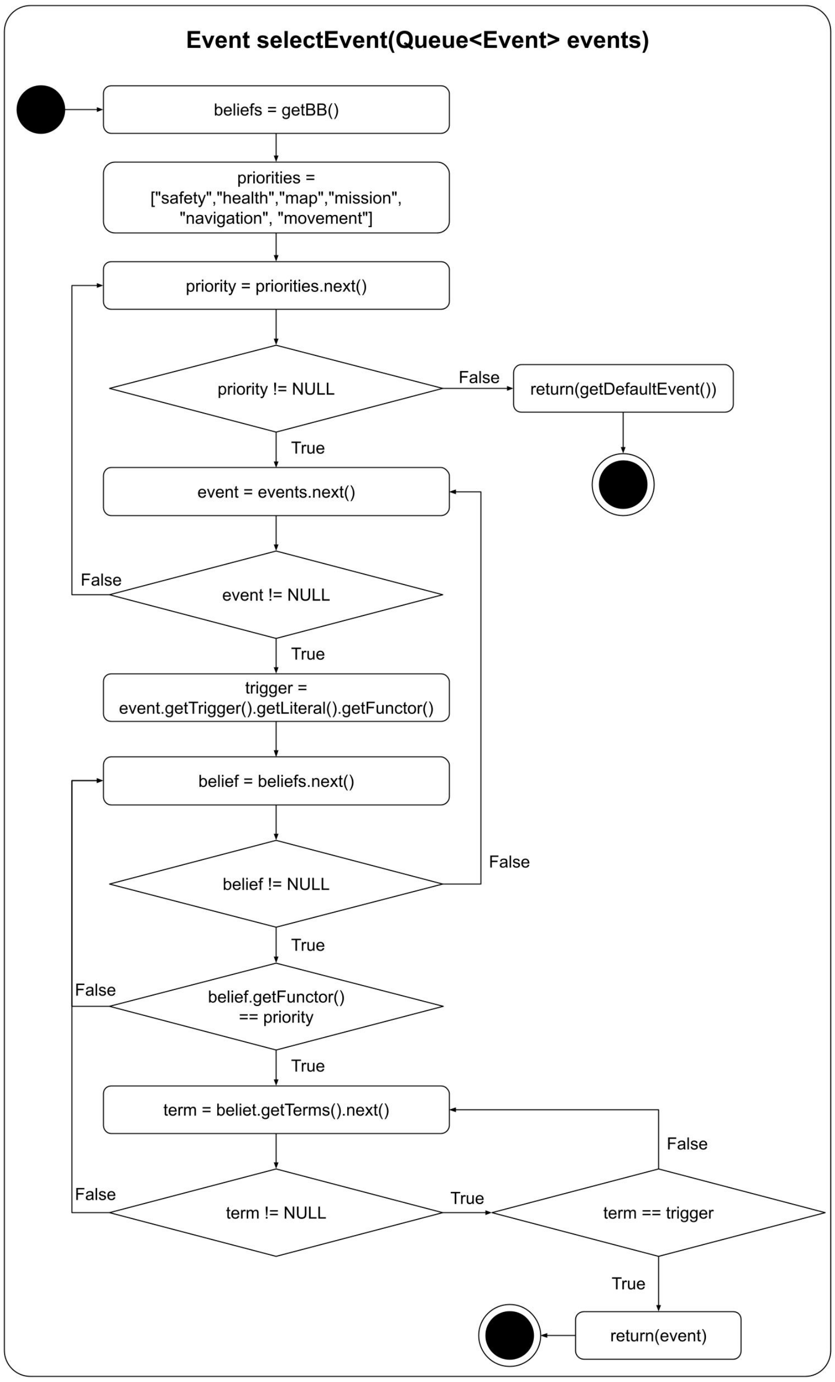

Event selectEvent(Queue<Event> events)
beliefs = getBB()
priorities = ["safety","health","map","mission", "navigation", "movement"]
priority = priorities.next()
priority != NULL
False
return(getDefaultEvent())
True
event = events.next()
False
event != NULL
True
trigger = event.getTrigger().getLiteral().getFunctor()
belief = beliefs.next()
belief != NULL
False
True
False
belief.getFunctor() == priority
True
term = beliet.getTerms().next()
False
False
term != NULL
True
term == trigger
True
return(event)

Figure 5. Event selection function.

With the highest priority event having been chosen, it is necessary for the reasoner to select the best option for the intention queue. Once the reasoner has selected which event to use, as discussed above, the reasoner checks the context of all plans triggered by the selected event. The plans that pass this context check become the list of options that the agent can set as intentions. The reasoner then calls the option selection function to select which option should be set as the next intention for the agent. This is necessary as there could be more than one plan applicable to the selected event. For example, default plans, which provide the agent with behaviour in the event that no other plan is applicable, are by definition always applicable. Even though these plans are always applicable, they only provide useful behaviour if it is the only plan applicable to the agent's context at that time.

If there is another applicable plan, it should be selected instead. This leads to a dilemma: how should the option selection function identify the highest priority option?

Ideally, the selected option should be the most applicable or most specific plan. In practice, making this determination can be difficult. Is the most specific plan the one with the most terms in the context? Perhaps yes, however it may not always be the case. That said, one must wonder, how often are there multiple applicable plans for the highest priority event? In examining the contexts of the plans in the Agent in a Box, it was noted that there were generally no more than two options available to the option selection function, one of which was a default plan. This observation provides a simple means for selecting the most appropriate option: a default plan should not be selected if a non-default plan is available. The implementation of the option selection function is illustrated in Figure 6. This function iterates through the options and checks the length of the plan context. If the context is not null, the plan is not a default plan, and the option is selected. Otherwise, the next plan is checked. If the only available plan is a the default plan, this option is selected.

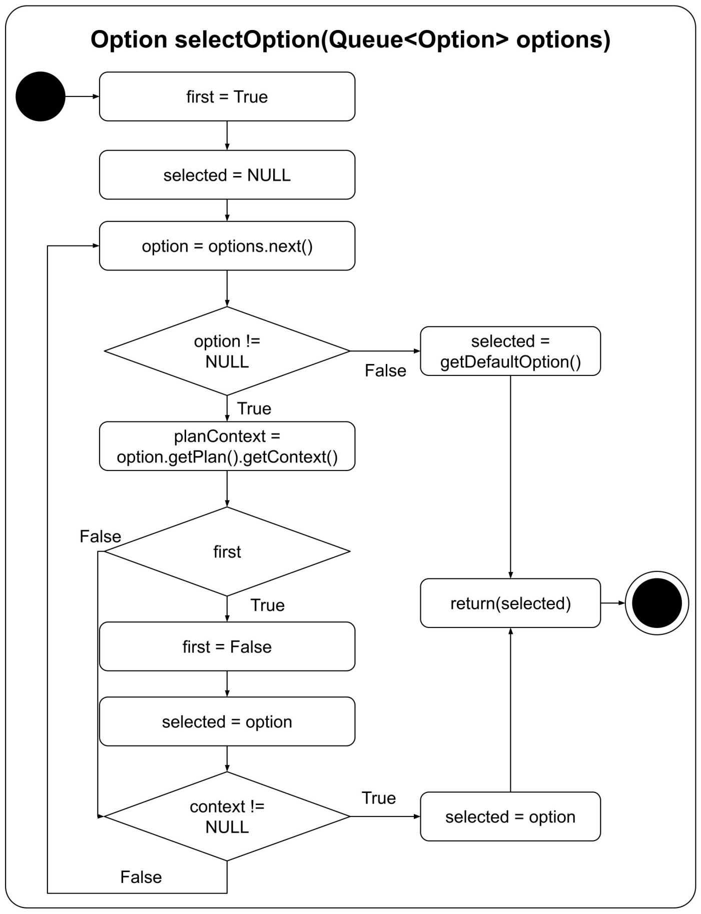


Figure 6. Option selection function.

With the overridden event and option selection functions complete, the Agent in a Box has a method for selecting the highest priority event and an applicable plan option for the agent to add to its intention queue. The agent now needs a set of plans which define the agent's behaviour. The behaviour framework for the Agent in a Box is discussed in the next section.

4.3. Agent in a Box: Behaviour Framework

The Agent in a Box provides a generic behaviour that is useful for a variety of mobile robotic agents. This includes a set of plans which provide the agent with the ability to navigate, update its map, and manage consumable resources. The Agent in a Box also provides support for agent movement, obstacle avoidance, and mission management. As was discussed in the previous section, there is also a mechanism for the developer to specify the type of behaviour that is associated with each event trigger, enabling the reasoner's behaviour prioritisation to select the highest priority plan for any given context. Developers of domain-specific agents are not required to use every type of behaviour provided by the Agent in a Box, if they are not all necessary.

This section describes the behaviour framework provided by the Agent in a Box. It also outlines how the developer of an agent for a specific application domain must refine, specialise, or customise the generic behaviours for their domain. This includes providing the agent with specific beliefs, plans, and rules for the agent to use. For example, the developer must provide a definition of the map, the implementation of plans for moving the agent and performing obstacle avoidance appropriate for their domain, and defining the agent's mission in a way that the generic plans can use. In this section, a variety of AgentSpeak listings are provided. These listings detail the implementation of various agent behaviours. The listings are colour-coded, showing the beliefs and event triggers that are provided by the Agent in a Box in blue, and the beliefs and event triggers that a developer must provide when developing for their specific domain are coloured green.

4.3.1. Navigation

No matter the application domain, all mobile robotic agents need the ability to plan a route to a destination location. The Agent in a Box provides this behaviour, including how the route is generated as well as how the map is defined, using a BDI agent navigation framework [49]. The navigation framework provides the implementation of the plan for the `!navigate(Destination)` achievement goal, which is shown in Listing 4. This plan uses Jason's A* implementation [50] to fetch the route from the agent's perceived position to the destination. It also includes the behaviour belief `navigation(navigate)`, which specifies the event trigger associated with navigation, used by the behaviour prioritisation methods. The generated path is a list of location waypoints for the agent to follow. These are adopted as as sequence of `!waypoint(NextPosition)` achievement goals. In order for the agent to move, the developer for the specific domain must provide an appropriate implementation of the `!waypoint(_)` achievement goal. Example implementations of this are provided in the case studies section, Section 4.4.

Listing 4: Navigation Plan [49].

```
{ include("a_star.asl") }
navigation(navigate).
+!navigate(Destination) : position(X,Y) & locationName(Current,[X,Y])
<-  ?a_star(Current,Destination,Solution,Cost);
for (.member( op(Action,NextPosition), Solution)) {!waypoint(NextPosition);}.
```

The navigation plan requires several domain-specific parameters to work. This includes a map, written in AgentSpeak which includes predicates that specify the names and coordinates of the points of interest, using the `locationName(Name,[X,Y])` predicate, and possible paths that an agent can navigate between the locations, using the `possible(A,B)` predicate. The navigation plan also requires successor state predicates, of the form `suc(CurrentState,NewState,Cost,Operation)`, which provides A* the cost of moving between any two nodes, and a heuristic predicate, which provides the estimated cost for moving from any given node to the destiation. The heuristic takes the form of `h(CurrentState,Goal,H)` [49]. Last, the developer must provide plans for how the agent should achieve the `!waypoint(_)` goal in the specific application domain. This depends on the actions that are available to the agent based on the specific actuators available. Example map definitions, successor state predicates, heuristic predicates, and movement plans are provided in Section 4.4.

By specifying an application domain-specific map, successor state predicates, heuristic predicate, and movement plans as discussed above, a developer can make use of the navigation achievement goal `!navigate(_)` to send the agent to a location on the map. Now that the agent has the ability to navigate to a destination, how the agent's mission should be defined will be discussed next.

4.3.2. Mission Definition

For the Agent in a Box to be useful to users, it needs to be able to complete domain-specific missions. These missions need to be implemented so that they can be interrupted and resumed by other plans in the framework as needed. To illustrate this point, consider a simple example. Assume that an agent is at location *A* in the environment shown in Figure 7 and has been given the goal of moving to location *D*. This is easily achieved using the navigation plan discussed in the previous section by adopting the goal of `!navigate(D)`. To achieve `!navigate(D)`, the agent adopts achievement goals for `!waypoint(B)`, followed by `!waypoint(C)`, and `!waypoint(D)`. What if the mission is complicated by adding an additional challenge to this scenario, such as the agent realising that it needs to charge its battery at a station located at *E* after having already adopted its waypoint goals? Fortunately, the behaviour framework provided by the Agent in a Box includes resource management behaviour. Although the details of this behaviour will be discussed in more detail later, in Section 4.3.4, one can presume that the agent should navigate to location *E* to charge its battery before moving to location *D*. The issue is that, once the battery has been recharged, the agent's intention queue will still contain the waypoint goals for moving the agent from location *A* to location *B*, and so on, however the agent is now at location *E*; the agent's mission context has changed. This means that the agent needs to regenerate these goals based on this new starting location.

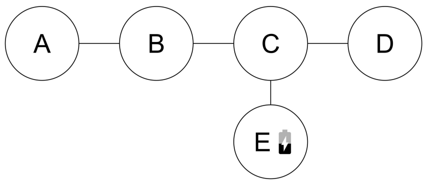


Figure 7. Mission management scenario.

In the case of this simple example, the agent could drop these outdated intentions and readopt the navigation goal, in effect regenerating the waypoint goals. The challenge here is that the mission may be a domain-specific mission that has been specified using achievement goals and parameters not known at the time that the framework was developed. This means that the framework's battery recharging plan needs to readopt an unknown, domain-specific achievement goal from its generic, framework provided plan. Therefore, a generic mission-level achievement goal is needed so that other framework provided plans can adopt them.

Now that the need for a generic mission-level achievement goal has been established, consideration will be placed on what this goal needs to accomplish. It needs to specify a specific mission, possibly among many possible missions that the agent is capable of, and it also needs to specify any needed parameters that are relevant to that mission. This needs to be accomplished using a generic achievement goal that can be adopted by other plans in the framework. This mission also needs to be implemented so that it can be interrupted, suspended, and readopted. Therefore, the agent will need to set a belief with the mission parameters as well as beliefs on the details of the agent's progress through the mission, so that the agent doesn't needlessly repeat any parts of the mission that have already been completed.

Listing 5 shows an implementation of the mission management behaviour for a navigation mission using the generic `!mission(Goal,Parameters)` goal. This achievement goal allows for the specification of a specific mission goal with the first parameter. The second parameter is a list of parameters that need to be provided to this mission. The context checks for the mission type, allowing the plan body to be used for setting a mission level mental note with the mission parameters, useful for readopting the mission if necessary, adopting the mission specific achievement goal, and then finally dropping the mental note once the mission has been completed. The navigation mission shown in this listing is included with the Agent in a Box, as any mobile agent could be expected to complete a simple navigation mission. A developer of an agent for a specific domain can add additional domain-specific missions for their custom application by providing other plans that achieve the `!mission(_,_)` in this way. As long missions are implemented using this type of plan they can be interrupted and resumed seamlessly by the plans provided by the Agent in a Box.

Listing 5: Mission Management.

```
+!mission(Goal,Parameters) : Goal = navigate & Parameters = [Destination]
<-  +mission(Goal,Parameters); !navigate(Destination); -mission(Goal,Parameters).
```

Now that the Agent in a Box has the ability to navigate through the environment, and can be provided with domain-specific missions, the Agent in a Box needs to be able to avoid obstacles.

4.3.3. Obstacle Avoidance and Map Update

As the agent moves through the environment, there is a possibility that the agent may need to avoid obstacles. There are generally two types of obstacle avoidance scenarios. In the first scenario, the obstacle can be avoided without a significant impact on the mission, for example, stopping to wait for a pedestrian that has wandered into the agent's path. Once the pedestrian has moved out of the way, the agent can continue its mission without issue. In the second type of obstacle avoidance scenario, the obstacle causes a significant change in the agent's mission context and the presence of the obstacle has invalidated the agents intentions in some way. For example, if the agent is navigating to a destination and finds that a road on its path has been closed unexpectedly, the agent's waypoint goals will no longer be valid. Avoiding this second type of obstacle will require a change to these goals.

A sample of the first obstacle avoidance scenario is provided in Listing 6. Assume that the agent has a perception associated with the type of obstacle, in this case `pedestrian(blocking)`. A further assumption is that the agent has an action which stops the robot: `drive(stop)`. Using this perception-generated belief and avoidance action, the obstacle avoidance plan is a simple belief-triggered plan: stop the robot when there is a pedestrian blocking the way. To ensure that this plan is executed to its completion, without any interruption, this plan is implemented with the `[atomic]` annotation. The event trigger must also be provided using `safety(pedestrian)` in order for this event to be prioritised by the reasoner. By providing this belief, the developer can add any needed obstacle avoidance behaviour using the sensors and actuators available for their application.

Listing 6: Simple Obstacle Avoidance.

```
safety(pedestrian).
@obstacleAvoid [atomic]
+pedestrian(blocking) <- drive(stop).
```

This first type of obstacle avoidance works well for obstacles that can be easily avoided without any significant impact on the mission, or obstacles that only provide a short disruption. By contrast, for longer term obstacles which interrupt the mission, as described in in obstacle avoidance scenario two above, the agent needs to update its map to reflect the obstacle and regenerate the waypoint achievement goals that are in its intention queue.

The implementation of this map update is provided in Listing 7. Here, the agent has observed an obstruction that was not on the map by perceiving `obstacle(Next)`. This belief needs to be generated by the perception translator based on the robot's sensor data. With the obstacle observed, the agent may need to update the map. The context check verifies if this obstacle contradicts any belief that the agent can move from its current location to the location of the obstacle. If it does, the agent needs to update the map and suspend and readopt any mission that the agent was working on. The first line in the listing provides the agent with a belief of what type of behaviour is triggered by `obstacle`. The plan body then checks if the agent was on a mission, which would require the agent drop any invalidated intentions before readopting them. This is done with a simple conditional statement.

Listing 7: Map Update.

```
map(obstacle).
+obstacle(Next)
:    position(Location)
& locationName(Current, Location)
& possible(Current,Next)
<-  -possible(Current,Next);
 if (mission(Name,Parameters)) {
.drop_all_intentions;
!mission(Name,Parameters);
}.
```

Obstacle avoidance is not the only type of issue that can interrupt the agent's progress on a mission, the Agent in a Box must also manage its resources, such as the charge of a battery, while working on its mission.

4.3.4. Management of Resources

A common aspect for mobile robots is the need for resource management. This could be to replenish some consumable resource, such as fuel, battery state, or even to seek a maintenance garage for more significant maintenance. Resource management behaviour requires the agent to stop whatever it may have been doing and then navigate to a maintenance station to either refuel, recharge, or seek some other type of maintenance. Once this maintenance is complete, the agent can continue its interrupted mission, however the agent will not be where it was when the mission was interrupted. Therefore, the agent's context will have changed significantly. This means that the management of resources requires that the agent be able to drop out-of-date intentions and then readopt mission goals once the maintenance has been completed.

Listing 8 provides the Agent in a Box's mechanism for replenishing a depleted resource. The first plan in the listing triggers on the addition of the `resource(State)` belief, generated by the perception translator node based on the data from a sensor monitoring the resource. In order for the agent to properly prioritise this event, the agent needs to be provided the `health(resource)` belief so that the event selection function can identify this trigger as being associated with a health-related behaviour. The plan contexts limit the plans to being applicable only when the resource has been depleted using the `lowResource(State)` rule, also provided in the example, and if the agent is not already replenishing the resource. The plan body contains a check for any mission that may have been running when the resource was depleted. If there was, the intentions are dropped before adopting the goal of `!replenishResource` and then readopting the mission goal. If there was no mission in progress, the agent adopts `!replenishResource` without issue.

There are three plans responsible for replenishing the resource: The first is for moving the agent to the maintenance station, using `!navigate(Station)`, and then docking with the station. This plan then readopts the achievement of the `!replenishResource` goal so that the progress can be monitored. The second plan is responsible for undocking from the station once the resource has been fully replenished, using the `fullResource(State)` rule, also defined in the listing. The last of the plans is a default plan which ensures that the goal of replenishing the resource is readopted while the agent works on replenishing the resource.

Listing 8: Resource Management Plans and Rules.

```
health(resource).
health(replenishResource).
:   docked(false)
& lowResource(State)
<-  if (mission(Name,Parameters)) {
.drop_all_intentions;
!replenishResource;
!mission(Name,Parameters);
} else {
!replenishResource;
}.
+!replenishResource : lowRersource(_) & docked(false) & stationLocation(Station)
<-  !navigate(Station); station(dock); !replenishResource.
+!replenishResource : fullResource(_) & docked(true)
<-  station(undock).
+!replenishResource
<-  !replenishResource.
lowResource(State) :- resource(State) & resourceMin(Min) & State <= Min.
fullResource(State) :- resource(State) & resourceMax(State).
```

In order for the agent to properly use the resource management behaviours provided by the Agent in a Box, a number of things are needed. First, any mission that can be interrupted should have an associated mission level mental note on any relevant mission parameters. This way, the mission can be readopted once the resource management process has been completed. Additionally, the agent needs to perceive the status of the resource using the `resource(State)` perception, which specifies the state of the resource; also needed are perceptions of whether or not the agent is docked at the station using both `docked(true)` and `docked(false)`. The agent also needs to know the location of the station, using the `stationLocation(Station)` belief, and the maximum and minimum acceptable state of the resource, using `resourceMin(Min)` and `resourceMax(Max)`. Last, the action for docking and undocking the agent with the station, `station(dock)` and `station(undock)` is needed.

4.3.5. Summary of the Behaviour Framework

This section has detailed Agent in a Box's behaviour framework, including the generic behaviour that is provided by the framework and what needs to be done to use the framework for a specific application domain. The Agent in a Box can be applied to a variety of mobile robotic agents, providing the needed skeleton for the agent to perform missions, navigate, move, update its map, avoid obstacles, and manage its resources. Additionally, the Agent in a Box depends on elements that the developer of an agent for specific application domains need to provide. For example, the navigation behaviour requires a map, written in AgentSpeak, as well as domain-specific plans for the `waypoint(_)` goal. Also needed are specific perceptions, actions, and beliefs that the agent needs in order to work. These include perceptions of the agent's position, resource state, docking station status, and an action for docking the robot at the maintenance station.

Now that the Agent in a Box has been provided, it is time to demonstrate its use for controlling agents in different application domains. The behaviour of the agent in these domains will be used for evaluating the behaviour and usefulness of this approach.

4.4. Case Studies

To validate the usefulness of the Agent in a Box, it has been used to implement mobile robots in three application domains. The domains were chosen to expose how agents implemented with the Agent in a Box work under different scenarios. This ensures that the results observed in the experiments were not tied to the properties of any specific environment, but rather demonstrate the usefulenss of the Agent in a Box for mobile robots in general. Additionally, the domains highlight the level of effort needed for an agent to be developed for different application domains while leveraging the features of the Agent in a Box framework. The first application domain explored is a grid-based domain, discussed in Section 4.4.1. Building on this environment, an autonomous car, simulated in AirSim, was driven through a neighbourhood environment, discussed in Section 4.4.2. Lastly, a prototype mail delivery robot is discussed in Section 4.4.3.

4.4.1. Grid Environment

The simplest environment that the Agent in a Box was applied to was a grid environment. Shown in Figure 8, this four by four grid was used for testing the basic principles of the Agent in a Box. In this environment, the agent was situated at a starting location and given the task of moving to a destination location. The agent was given a map showing the possible paths between the locations on the map, however this map did not include the locations of all of the obstacles. Therefore, the agent needed to update its map as it ran. The agent was also equipped with a simulated battery which reduced in charge over time, causing the agent to recharge the battery at a charging station located on the map. There were also pedestrians on the map, obstacles which would block the agent's path until the agent honked a horn at them. The agent perceived the environment with perceptions of the battery state, map locations, obstacles, and pedestrians adjacent to the agent. The agent was given the ability to honk a horn, connect and disconnect from the charging station, and move to adjacent grid locations. There were two versions of this environment: a synchronised and an asynchronised version.

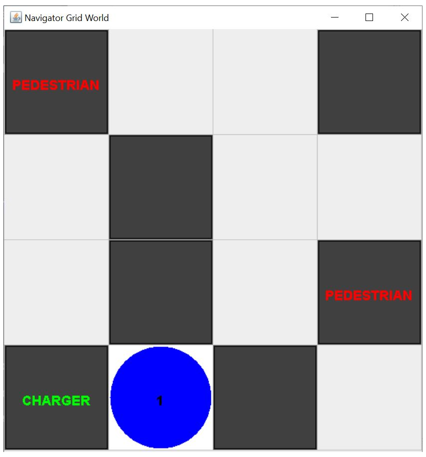


Figure 8. Grid environment.

The synchronised version of the environment used a single thread of control, meaning that the environment update was tied to the reasoning cycle. The environment would update, then provide perceptions to the agent. The agent would then reason about those perceptions and provide an action, which would then be used to update the environment, then repeating the process. The synchronised version of the grid environment architecture is shown in Figure 9. The software, implemented using Jason's agent architecture and environment tools, is available on GitHub [51].

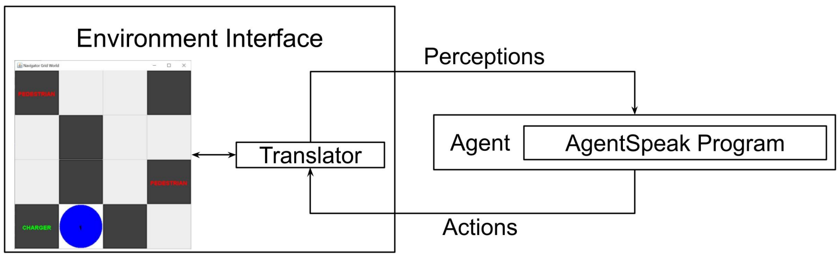


Figure 9. Synchronised grid environment architecture.

In contrast to the synchronous environment, the asynchronous version used a separate ROS node for the environment, which updated in its own time. The agent perceived the environment via a set of sensors which periodically monitored the environment and published their data to ROS topics. The agent was connected using SAVI ROS BDI, the Agent in a Box connection scheme discussed in Section 4.1 and shown in Figure 1. In this case, a single translator was used for connecting to the environment and passing symbolic perceptions to the agent while also passing actions back to the environment. The software for this version is available on GitHub [52]. The asynchronous version of the architecture is shown in Figure 10.

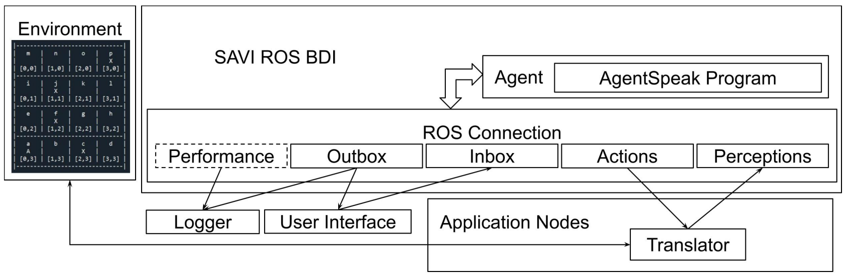


Figure 10. Asynchronised grid environment architecture.

The grid agent made good use of the Agent in a Box, using all of the provided behaviour features and priority levels. Its primary mission was navigation, provided by the Agent in a Box. It also used the resource management behaviour provided by the Agent in a Box for maintaining the charge of its battery and the map update behaviour to maintain the map used by the navigation plan. These methods all worked by ensuring that the translator node responsible for generating the perceptions and interpreting the actions used the same perception and action names used by the Agent in a Box. Some additional work was needed, however, in order for the agent to navigate, move, and avoid pedestrians. This involved providing the map, movement plans, and pedestrian avoidance behaviour.

The navigation behaviour provided by the Agent in a Box requires a map written in AgentSpeak, successor state predicates, and a heuristic predicate. An excerpt of the map is provided in Listing 9, specifying the coordinates of each grid location as well as the possible paths between locations.

Listing 9: Partial Grid Agent Map.

```
locationName(a,[0,0]).
locationName(b,[1,0]).
possible(a,b).
```

A sample successor state predicate, which uses a supporting rule, along with the heuristic predicate are provided in Listing 10. The successor state predicate provides the cost to the agent for moving to an adjacent grid square. The example shown provides the rule definition for this predicate for grid squares in the up direction. This uses a rule for getting the location coordinates from the belief base. Last is the heuristic predicate, defined by the last rule in the listing. This rule calculates the Euclidean distance between a given location on the grid and the goal location. This is a cost estimate used by the A* search algorithm which is then used by the navigation plan.

Listing 10: Grid Agent Successor State and Heuristic Definitions.

```
suc(Current,Next,1,up) :- ([X2,Y2] = [X1,Y1-1])
& possible(Current,Next) & nameMatch(Current,[X1,Y1],Next,[X2,Y2]).
nameMatch(Current,CurrentPosition,Next,NextPosition) :-
locationName(Current,CurrentPosition)
& locationName(Next,NextPosition).
h(Current,Goal,H) :- H = math.sqrt(((X2-X1)*(X2-X1))+((Y2-Y1)*(Y2-Y1)))
& nameMatch(Current,[X1,Y1],Goal,[X2,Y2]).
```

With the map defined, the agent needs plans for achieving the `!waypoint(_)` goal, which the navigation plan will adopt to move the agent along the route that has been generated. The movement plans are responsible for moving the agent to the specified location using the domain-specific actions available to the agent. The grid agent's movement was designed as shown in Listing 11. The first line in the listing provides the behaviour type for the reasoner to properly prioritise this trigger as being related to movement behaviour. This is followed by the plan for moving the agent. The agent perceives its position with the `position(X,Y)` predicate and moves using the `move(Direction)` action which moves the agent in the desired direction. The plan context makes use of the `locationName(Current,[X,Y])` and `possible(Current,Next)` predicates which are map related predicates. It also uses rules for determining the direction that the agent needs to move using simple arithmetic. In this case, the grid is assumed that the `X` value increases as the agent moves to the right and that the `Y` value increases as the agent moves down. Note that this plan is only responsible for moving the agent, it is not concerned with health and safety related concerns, such as if there is an obstacle in the path.

Listing 11: Example Movement Plan and Rules.

```
movement(waypoint).
+!waypoint(Next)
:   position(X,Y) & locationName(Current,[X,Y]) & possible(Current,Next)
& direction(Current,Next,Direction)
<-  move(Direction).
direction(Current,Next,up)
:-  possible(Current,Next) & locationName(Current,[X,Y]) & locationName(Next,[X,Y-1]).
direction(Current,Next,down)
:-  possible(Current,Next) & locationName(Current,[X,Y]) & locationName(Next,[X,Y+1]).
direction(Current,Next,left)
:-  possible(Current,Next) & locationName(Current,[X,Y]) & locationName(Next,[X-1,Y]).
direction(Current,Next,right)
:-  possible(Current,Next) & locationName(Current,[X,Y]) & locationName(Next,[X+1,Y]).
```

With the robot able to navigate and move through the environment, the last needed piece of this implementation is to provide the pedestrian avoidance behaviour. The Agent in a Box provides a mechanism for prioritising safety-related behaviour. The agent perceives the presence of pedestrians through the `pedestrian(_)` belief. The desired obstacle avoidance action in this environment is for the agent to honk a horn to signal the pedestrian to get out of the way. This can be easily implemented as a belief triggered plan, as shown in Listing 12. First, the behaviour belief was provided, identifying to the reasoner that a safety related behaviour will be triggered by pedestrian events. Next, the plan name was specified as `pedestrianAvoidance`, further specified to be an atomic plan, meaning that it must be run to completion, without interruption. Last, the belief-triggered plan is provided. By using this design approach, the implementation of the other movement-related plans do not need to be concerned with obstacle avoidance as the agent's prioritisation method will select this plan as the highest priority.

Listing 12: Grid Agent Obstacle Avoidance.

```
safety(pedestrian).
@pedestrianAvoidance [atomic]
pedestrian(_) <- honk(horn).
```

A video of the grid behaviour is available on YouTube [53]. With the grid agent complete, it is time to apply the Agent in a Box to a more realistic environment: a simulated autonomous car.

4.4.2. Simulated Autonomous Car Environment

Although the grid agents provided an opportunity to demonstrate all features of the Agent in a Box, it was an overly simplistic environment. If the Agent in a Box is to be useful, it needs to work in an environment more representative of the real world. The Agent in a Box was therefore used to drive a simulated autonomous car.

AirSim is a simulator maintained by Microsoft and available open source [54,55]. Originally developed using the Unreal Engine, but now also available using the Unity

Engine, it was developed to provide a realistic outdoor environment for the development of UAVs, although it is also useful for the development of autonomous cars [56]. The development focus was to drive the car in AirSim's neighborhood environment, in Figure 11. This environment provides an enclosed suburban neighbourhood, with roads and parked cars that need to be avoided. There are several intersections for the car to navigate. In this environment, the agent's mission was to drive the car to a specified destination in the environment. The car has a variety of sensors and actuators, including a speedometer, Global Positioning System (GPS), magnetometer (useful as a compass), camera, LIDAR, throttle, brakes, and steering. The software for the agent in this environment is also available on GitHub [57].

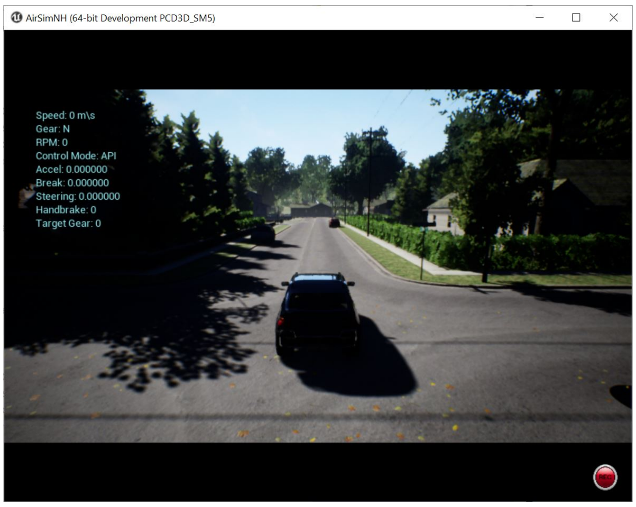


Figure 11. AirSim neighbourhood environment.

The architecture of the car agent using the Agent in a Box, is shown in Figure 12. The car was configured with a variety of sensors, including a GPS sensor, a magnetometer, a speedometer, a LIDAR, and a camera. Each of these sensors was monitored by a set of nodes. The GPS node provided the latitude and longitude coordinates of the car, the magnetometer was used for calculating the compass heading of the car, and the speedometer provided the car's speed in meters per second. The LIDAR sensor was used as an obstacle detection sensor and the camera was used for a lane-keep assist module which provided the recommended steering for the car to follow the road. The topics published by these sensor nodes were subscribed to by the perception translator, which assembled this data into predicates and published to the `perceptions` topic for the reasoner. The agent's control of the car was through setting the desired speed of the car and providing a steering setting. The speed setting was passed to a speed controller which controlled the accelerator and brake of the car. The steering, accelerator, and brake settings were all subscribed by a controller node, which commanded the car in AirSim. There were also modules for logging the behaviour of the reasoning performance and a user interface used for commanding the destination of the car. Within the agent were a number of internal actions, Java functions that the AgentSpeak program could call. These included actions for calculating the range and bearing to specified locations using latitude and longitude coordinates.

Although the behaviour implementation for the car required a number of domain-specific implementations, the mission behaviour for navigation was used as the main behaviour for the agent. Otherwise, the car needed a map of latitude and longitude coordinates for the navigation plan to use, plans for achieving the `waypoint` goal by controlling the car's steering and speed, and collision avoidance behaviour for avoiding parked cars.

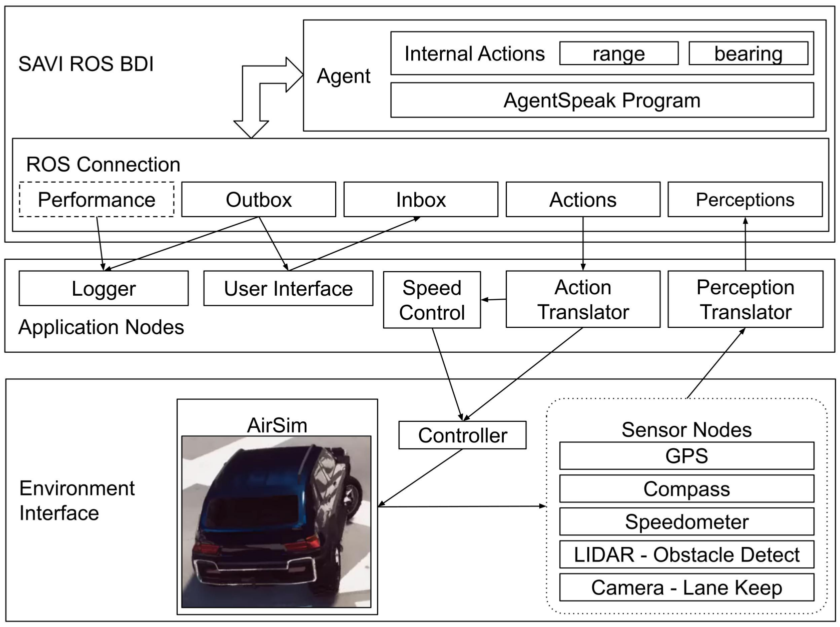


Figure 12. AirSim autonomous car architecture.

The map definition for this environment followed the structure defined for the Agent in a Box's navigation framework. Shown in Listing 13, the main difference was that the coordinate locations used latitude and longitude instead of simple grid locations.

Listing 13: Excerpt of the AirSim Car Map Definition.

```
locationName(post1,[47.6414823712,-122.140364991]).
locationName(post2,[47.6426242556,-122.140354517]).
possible(post1,post2).
```

The successor state rule and the heuristic rule needed to be specialised to work with latitude and longitude. The definition of these rules is shown in Listing 14. The successor state rule was simplified, only specifying that it was possible for the car to drive between the specified locations. The heuristic rule required an internal action for calculating the range between the coordinate locations.

Listing 14: AirSim Car Navigation Successor State and Heuristic Definitions.

```
suc(Current,Next,Range,drive)
 :-   possible(Current,Next)
& locationName(Current,[CurLat,CurLon])
& locationName(Next,[NextLat,NextLon])
& navigation.range(CurLat,CurLon,NextLat,NextLon,Range).
h(Current,Goal,Range)
 :-   locationName(Current,[CurLat,CurLon])
& locationName(Goal,[GoalLat,GoalLon])
& navigation.range(CurLat,CurLon,GoalLat,GoalLon,Range).
```

The agent has a collision avoidance capability, using the obstacle detection feature with the LIDAR sensor. This functionality was implemented using a belief triggered atomic plan, shown in Listing 15. First, the behaviour triggered by the `obstacle` was identified as being for safety behaviour. This belief is used by the reasoner to prioritise this event trigger as the agent's top priority. Next, the plan was annotated as an atomic plan meaning it must run to completion. Finally, the collision avoidance behaviour is provided. The behaviour steers the car away from the parked car obstacles along the side of the road when the distance to an obstacle is less than 5 m.

Listing 15: AirSim Car Obstacle Avoidance Behaviour.

```
safety(obstacle).
@obstacleAvoidance [atomic]
+obstacle(Distance) : (Distance < 5) <- steering(-0.3).
```

With navigation and obstacle avoidance handled, it is time to provide plans for driving the car. These are provided in Listing 16. The plans for the `!waypoint(Location)` achievement goal are responsible for moving the car toward locations on the map specified with latitude and longitude coordinates. These plans are implemented recursively, meaning that the only plan which does not readopt this achievement goal is for the context where the car has arrived at the location. The first plan in the listing stops the car when it has arrived at the location by adopting the achievement goals for setting the speed to zero and setting the steering to zero with the lane-keep assist turned off. Next is the plan responsible for slowing the car as it approaches a location. Here, the car slows to 3 m/s, and the steering is set to the direct bearing to the location with the lane-keep disabled, as the vehicle is close the the location before readopting the goal. The third plan is applicable when the car is not near the location and needs to drive toward it. In this case, the lane-keep assist is enabled, and the vehicle speed is set to 8 m/s, the desired cruising speed for the car. Again, the goal is readopted so that the car continues driving toward the location. Last is a default plan which keeps the car driving toward the location using recursion.

Listing 16: AirSim Car Driving Behaviours.

```
movement(waypoint).
+!waypoint(Location)
:   atLocation(Location,_)
<-  !controlSpeed(0); !controlSteering(0, lkaOff).
+!waypoint(Location)
:   nearLocation(Location,_) & (not atLocation(Location,_))
& destinationBearing(Location,Bearing)
<-  !controlSteering(Bearing, lkaOff); !controlSpeed(3); !waypoint(Location).
+!waypoint(Location)
:   (not nearLocation(Location,_)) & destinationBearing(Location,Bearing)
<-  !controlSteering(Bearing, lkaOn); !controlSpeed(8); !waypoint(Location).
+!waypoint(Location) <- !waypoint(Location).
```

These plans are rule supported, using a set of rules, shown in Listing 17, for assessing if the car is near or at a location. There is also a rule for finding the nearest location to the car, which searches the map beliefs for the location with the smallest range to the current location of the car. There are also rules for calculating the destination range and the destination bearing. All of these rules use internal actions for calculating the range and bearing between locations.

Listing 17: AirSim Car Localization Rules.

```
nearLocation(Location, Range) :- gps(CurLat,CurLon) & locationName(Location,[Lat,Lon])
& navigation.range(CurLat,CurLon,Lat,Lon,Range) & Range < 20.
atLocation(Location, Range) :- gps(CurLat,CurLon) & locationName(Location,[Lat,Lon])
& navigation.range(CurLat,CurLon,Lat,Lon,Range) & Range < 7.
nearestLocation(Location,Range) :- gps(CurLat,CurLon) & locationName(Location,[Lat,Lon])
& locationName(OtherLocation,[OtherLat,OtherLon])
& OtherLocation \== Location & navigation.range(CurLat,CurLon,Lat,Lon,Range)
& navigation.range(CurLat,CurLon,OtherLat,OtherLon,OtherRange)
& Range < OtherRange.
destinationRange(Location,Range) :- locationName(Location,[DestLat,DestLon])
& gps(CurLat,CurLon) & navigation.range(CurLat,CurLon,DestLat,DestLon,Range).
destinationBearing(Location,Bearing) :- locationName(Location,[DestLat,DestLon])
& gps(CurLat,CurLon) & navigation.bearing(CurLat,CurLon,DestLat,DestLon,Bearing).
```

Listing 18 provides the implementation of the speed controlling behaviours. First, the mental note of the speed setting is provided for when the agent starts and the car is not moving. This mental note allows the agent to remember the speed setting so that the cruise controller does not need to be reset needlessly. The behaviour belief identifies the behaviour type for the reasoner's event selection function. Next is a plan for updating the speed setting if the speed needs to be changed. A plan is also provided for setting a speed setting in the event that one has not yet been specified or in case the mental note has been lost for some reason. The `setSpeed(_)` action is received by a cruise controller module which controls the accelerator and brake to maintain the speed of the car. The last plan in Listing 18 is the default plan, which is only applicable if the speed setting does not need to be changed.

Listing 18: AirSim Speed Control Behaviours.

```
speedSetting(0).
movement(controlSpeed).
+!controlSpeed(Speed) : speedSetting(Old) & (Old \== Speed)
<- -speedSetting(_); +speedSetting(Speed); setSpeed(Speed).
+!controlSpeed(Speed) : not speedSetting(_) <-  +speedSetting(Speed); setSpeed(Speed).
+!controlSpeed(_).
```

Listing 19 provides the rules and plans for controlling the car's steering. The first item in the listing is a belief with respect to magnetic declination, needed for converting a magnetic bearing to a true bearing. This course correction is used to calculate a steering setting using the rules which define the steering setting rules. The format of this predicate is `steeringSetting(TargetBearing, SteeringSetting)`. These rules set the steering setting to the maximum magnitude when the course correction is greater than 20°. If the magnitude is smaller, the steering setting is dampened by dividing it by 180, a crude but effective way of controlling the car's steering. This is followed by a rule for using the lane-keep assist for steering the car. The lane-keep assist node provides a perception with a steering recommendation for the car, as well as parameters representing the slope and intercept points of the curbs on the road. If the lane is detected, these slopes are nonzero. If the lane is not detected, then this predicate is not available to the agent.

There are three steering plans in Listing 19, triggered on the addition of `!controlSteering(_,_)`. First, the behaviour belief identifying that these plans are movement-related, enabling the reasoner to properly prioritise this behaviour, is provided. The first steering plan is used for steering the car with magnetic bearing angles only. This is the case when the lane-keep assist is either disabled or the lane has not been detected by the lane-keep assist module. The second plan uses the lane-keep assist to follow the road. Lastly, there is a default plan, which is provided for completeness.

Listing 19: AirSim Car Steering Behaviours.

```
declination(7.5).
courseCorrection(TargetBearing, Correction)
:-  compass(CurrentBearing) & declanation(Declanation)
& (Correction = TargetBearing - (CurrentBearing + Declanation)).
steeringSetting(TargetBearing, 1)
:-  courseCorrection(TargetBearing, Correction) & (Correction >= 20).
steeringSetting(TargetBearing, -1)
:-  courseCorrection(TargetBearing, Correction) & (Correction <= -20).
steeringSetting(TargetBearing, Correction/180)
:-  courseCorrection(TargetBearing, Correction)
& (Correction < 20) & (Correction > -20).
lkaSteering(Steering) :- lane(Steering,A,B,C,D) & ((not (C == 0)) | (not (D == 0))).
movement(controlSteering).
+!controlSteering(Bearing,LkaSetting)
:   steeringSetting(Bearing, Steering)
& ((not lkaSteering(_)) | (LkaSetting == lkaOff))
<-  steering(Steering).
+!controlSteering(Bearing,lkaOn) : lkaSteering(Steering) <- steering(Steering).
+!controlSteering(_,_).
```

The Agent in a Box was successfully used for controlling the car simulated with AirSim. This was accomplished by steering the car using a combination of a lane-keep assist module and a magnetic bearing to the destination. The agent was also able to perform collision avoidance. The car agent made good use of the Agent in a Box, which enabled the development of the sensor and actuator nodes to be completed in isolation from the scope of the rest of the agent. A video of a navigation scenario is available on YouTube in [58] along with another focusing on the lane-keep and collision avoidance behaviour is available in [59].

4.4.3. Mail Delivery Robot

With the features of the Agent in a Box demonstrated in simulated environments, a third case study was completed with a prototype mail delivery robot for delivering mail in institutional settings. A prototype of this robot, building on previously published work [60,61], was implemented with an iRobot Create and a Raspberry Pi 4 computer, as shown in Figure 13, and tested in a lab environment. The robot was controlled using the `create_autonomy` node [62], which acts as a connector for the robot to ROS, providing a variety of publishers of the robots various sensors, and a subscriber for its actuators. This work used the bumper sensor for collision detection, battery charge ratio, and the odometer for the robot's wheels. The robot is controlled by sending parameters for the robot's wheels, specifically the desired linear and rotational speed. The sensor data were provided to the reasoner in the form of perception predicates, formatted by the perception translator. The action translator generated the appropriate commands for the robot's wheels from the action predicates provided by the agent. A user interface node provided the means for a user to command the robot to complete mail missions by specifying the locations of the sender and receiver on a predefined map. The software implantation of the mail delivery agent, used for controlling the robot, is available on GitHub [63].

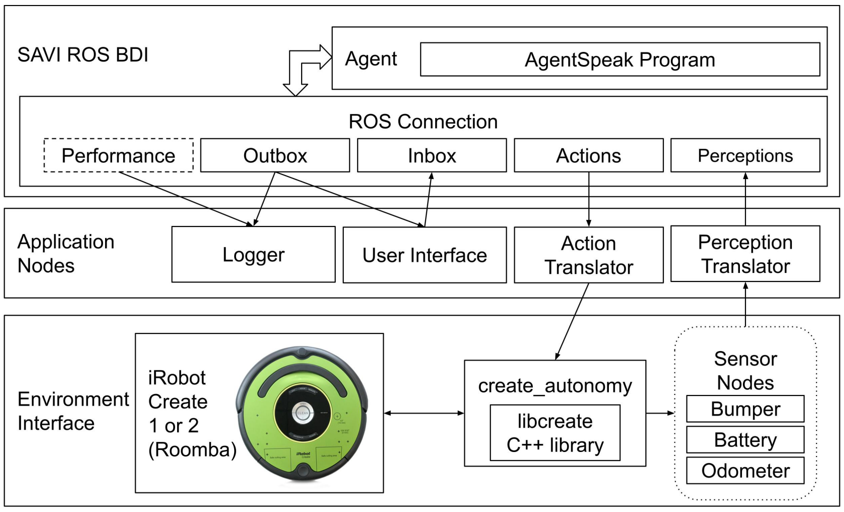


Figure 13. Prototype mail delivery robot architecture.

The mission for the agent controlling the mail robot is provided in Listing 20. This listing provides the implementation of the `!mission(_,_)` goal for mail missions that can be commanded by a user. The parameters specify the sender and receiver locations on the map. This plan adopts a mission related mental note and achieves the mail mission using the `!collectAndDeliverMail(_,_)` goal, which in turn uses the `!collectMail(_)` and `!deliverMail(_)` goals, also defined in the listing. There are two plans for collecting mail and two more plans for delivering mail using the `!navigate(_)` goal provided by the Agent in a Box. The first plan sends the robot to the sender's location to collect the mail if the robot has not already collected it and the second is a default plan provided for the scenario where the robot already has the mail. Third is the plan for delivering mail that the robot has collected by moving to the receiver's location. Last is another default plan for the scenario where there was no mail for the agent to deliver.

Listing 20: Mail Robot Mission Definition.

```
mission(mission).
+!mission(Goal,Parameters)
:   Goal = 'collectAndDeliverMail' & Parameters = [Sender,Receiver]
<-  +mission(Goal,Parameters); !collectAndDeliverMail(Sender,Receiver);
-mission(Goal,Parameters).
mission(collectAndDeliverMail).
+!collectAndDeliverMail(Sender,Receiver) <- !collectMail(Sender); !deliverMail(Receiver).
mission(collectMail).
+!collectMail(Sender) : not haveMail <- !navigate(Sender); +haveMail.
+!collectMail(Sender).
mission(deliverMail).
+!deliverMail(Receiver) : haveMail <- !navigate(Receiver); -haveMail.
+!deliverMail(Receiver).
```

The map used by the mail robot was defined using a set of grid coordinates for various locations in the lab environment. These locations were defined in the same manner as the locations were defined for the grid agents, as shown in Listing 9. In addition to the definition of the map locations and possible movements between these locations, the map includes the successor state and heuristic rules used by the navigation plans, a rule for calculating the range between locations using Euclidean distance, and a rule for determining the location of the robot using the agent's beliefs about the grid position and map location names. These are provided in Listing 21.

Listing 21: Mail Robot Map Rules.

```
suc(Current,Next,Range,drive) :- possible(Current,Next) & locationName(Current,[X1,Y1])
& locationName(Next,[X2,Y2]) & range(X1,Y1,X2,Y2,Range).
h(Current,Goal,Range) :- locationName(Current,[X1,Y1]) & locationName(Goal,[X2,Y2])
& range(X1,Y1,X2,Y2,Range).
range(X1,Y1,X2,Y2,Range)
:- Range = math.sqrt( ((X2-X1) * (X2-X1)) + ((Y2-Y1) * (Y2-Y1)) ).
atLocation(Location,0) :- locationName(Location,[X,Y]) & position(X,Y).
```

With the mission and map defined, the agent needs the ability to move the robot. The Agent in a Box provides the `!navigate(_)` goal, which moves the agent to a specified destination location using a route obtained with A* search. The plans that achieve this goal move the agent between the locations using the domain-specific `!waypoint(_)` goal. The plans which implement this goal are provided in Listing 22. The listing provides three plans, one for stopping the agent with the `drive(stop)` action when the robot is at the location, a second for driving the robot toward the location using several subgoals and position belief updates, and finally a default plan.

Listing 22: Mail Robot Movement - Waypoint.

```
movement(waypoint).
+!waypoint(Location) : atLocation(Location,_) <- drive(stop).
+!waypoint(Location) : locationName(Location,[X,Y])
<-  !faceNext(Location); !drive(forward);
-position(_,_); +position(X,Y); !waypoint(Location).
+!waypoint(Location).
```

The subgoals used for moving the robot include the `!faceNext(_)` goal, used for turning the robot to face the next location it will be moving to, and the `!drive(_)` goal, which drives the robot linearly forward or backward. Excerpts of the plans and rules which implement these goals are provided in Listing 23. First, the listing provides the initial location of the robot and the direction that it is facing. This is followed by a plan which implements the achievement of the `!faceNext(_)` goal using the `direction(_)` belief and the `directionTonext(_)` rule, which provides the direction that the agent needs to face to move to the next destination. A sample of this rule is provided for the scenario where the agent needs to face `e`. The rules for the `w`, `n`, and `s` directions have not been included in the listing to save space, however they are available on Github [63]. The `!faceNext(_)`

triggered plan also uses a rule to determine the turn action parameter for turning the robot to face the required direction. The provided rule is for the case where the agent needs to turn to the left, although there are other rules needed for defining if the agent needs to turn to the right, turn around, or if there is no turn needed.

Listing 23: Mail Robot Movement - Excerpts of FaceNext

```
position(0,0).
direction(e).
movement(faceNext).
+!faceNext(Next) : direction(CurrentDirection) & directionToNext(Next,NewDirection)
& face(CurrentDirection,NewDirection,TurnAction)
<- !turn(TurnAction); -direction(_); +direction(NewDirection).
+!faceNext(Next).
directionToNext(Next,e) :- position(X1,_) & locationName(Next,[X2,_]) & X1 < X2.
face(OldDirection,NewDirection,left)
:- ((OldDirection == s) & (NewDirection == e)) | ((OldDirection == e)
& (NewDirection == n)) | ((OldDirection == n)
& (NewDirection == w)) | ((OldDirection == w) & (NewDirection == s)).
movement(drive).
```

There are two subgoals used for actually moving the robot: `!turn(_)` and `!drive(_)`. The implementations of these are provided in Listing 24. The goal for turning the robot takes the direction of the turn as a parameter, either `left`, `right`, or `around`. These use the `turnRate(X,Y)` belief, which specifies the linear and rotational parameters for turning the robot approximately 45°. To execute the turn, the agent adopts the `!move(X,Y,N)` goal, which sends the movement command to the robot `N` times. The plans for `!drive(_)` drives the robot forward or backward using the movement parameters specified in the `driveRate(_,_)` belief, also using the `!move(_,_,N)` goal, which is the last provided plan in the listing. The plan for this goal is implemented using recursion, using the movement action to control the robot and repeating the goal with a reduced counter after each iteration.

Listing 24: Mail Robot Movement - Turn, Drive, and Move.

```
movement(turn).
+!turn(left) : turnRate(X,Y) <- !move(X,Y,2).
+!turn(right) : turnRate(X,Y) <- !move(0-X,0-Y,2).
+!turn(around) : turnRate(X,Y) <- !move(X,Y,4).
+!turn(_).
movement(drive).
+!drive(forward) : driveRate(X,Y) <- !move(X,Y,5).
+!drive(backward) : driveRate(X,Y) <- !move(0-X,0-Y,5).
movement(move).
+!move(X,Y,Count) : Count > 0 <- drivexy(X,Y); !move(X,Y,Count-1).
turnRate(0,4.1).
driveRate(0.5,0).
```

With the plans for supporting the agent's movement defined, there is a need to provide the agent with the plans needed for obstacle avoidance and for maintenance of the robot's battery. The Agent in a Box provides support for this functionality by prioritising it over the agent's other activities. The obstacle avoidance for this agent is provided in Listing 25. The obstacle avoidance is a reactive behaviour implemented using a belief-triggered plan. This is triggered when the the robot's bumper sensor is activated. The resulting actions are to turn the robot to the left and attempt to the move the robot in an arc around the obstacle before turning to its original direction. As this is implemented as an atomic plan, the agent runs this plan to completion before moving to other activities.

Listing 25: Mail Robot Obstacle Avoidance.

```
safety(bumper).
@Bumper [atomic]
+bumper(pressed) <- !turn(left); !move(0.5,-3,10); !turn(left).
```

The final aspect of the agent's behaviour is the need for the agent to recharge the robot when it runs low on power. This was accomplished by providing the agent with the resource perceptions and actions for docking and undocking the robot that are expected by the Agent in a Box. By doing so, the agent can send the robot to the charging station when the battery needs to be recharged.

The Agent in a Box was successful at controlling the prototype mail delivery robot. Videos of the robot are available on Youtube [64–66].

4.4.4. Implementation Summary

The details of how the Agent in a Box can be used to control mobile robots in three environments have been provided. These environments included a grid-based environment, an autonomous car using AirSim, and a prototype mail delivery robot implemented with an iRobot Create and a Raspberry Pi. Using the framework's provided behaviours for mission management, navigation, map update, and resource management, all that was needed to complete the agent behaviour was to add domain-specific plans for movement and obstacle avoidance. For connecting the agent to the environment, the asynchronous grid environment, the car, and the mail robot all used ROS and the Agent in a Box approach, with sensor nodes, actuator nodes, and translator nodes. This design provided a consistent design for connecting the agent to the environment.

5. Results

The validation of the Agent in a Box presents the challenge of needing to validate the performance of a framework. This section provides the experimental results from validation experiments conducted with robotic agents implemented with the Agent in a Box. These experiments have been designed to highlight the properties of the Agent in a Box itself. These properties include the agent behaviours provided by the behaviour framework and behaviour prioritisation, and the runtime performance of the agents. The goal is to demonstrate that the Agent in a Box provides useful properties to robotic systems without a significant performance burden.

The first set of validation experiments focuses on the properties of the Agent in a Box's behaviour framework and the behaviour prioritisation added the the reasoner. These results come from runtime logs generated by the case study agents discussed in Section 4.4. The results highlight how the Agent in a Box's generic behaviours were successfully used to complete the missions in each of the case studies. It also shows how, despite the apparent differences in each of the robots, the missions were successfully interrupted and resumed as needed, for map update, resource management, and collision avoidance behaviours. The plans that implemented these interrupting behaviours were all successfully run during the agent's missions, despite the fact that the missions were implemented without specific concern for these issues.

The agents' missions, implemented as goal-directed plans, are validated first in Section 5.1. This section highlights how the agents missions were achieved using a combination of generic and domain-specific software. Included in the generic code, provided by the Agent in a Box, were the navigation plans which generated paths for the robot to follow as sequences of achievement goals which could be run, monitored, and suspended as needed. These paths were all domain-agnostic, simply telling the agent to achieve a series of waypoint goals. The implementation of these goals was domain-specific for each of these case studies. This highlights how a developer can easily map the generic navigation plans to their specific domain without concern for resource management or collision avoidance.

The next validation focus was the reactive behaviour of the case study agents. Discussed in Section 5.2, this included how the agents handle unexpected situations, such as incorrect maps, collision avoidance, and resource management. How the agent addressed each of these concerns was implemented separately from the missions. This means that the agent's missions were implemented without concern for how the agent would avoid collisions, manage resources, or handle inaccuracies on its map. These behaviours simply needed to work, interrupting whatever else the agent was doing, and then allowing the agent to continue with whatever mission was underway, without any specific knowledge of what that mission may have been.

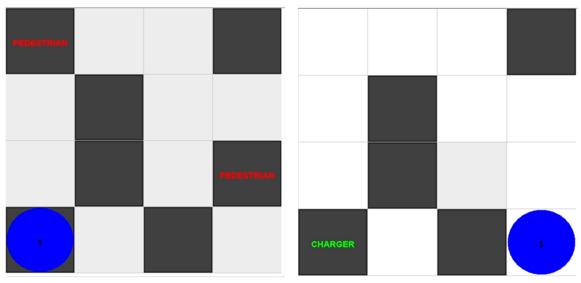
The last set of validation experiments focus on the performance of the Agent in a Box. These results are provided in Section 5.3. The focus of these experiments is to assess if there is a performance bottleneck resulting from the use of the Agent in a Box for controlling mobile robots. This experiment provided profile measurements which focused on assessing if the agent's reasoning imposed a performance bottleneck. The concern was that the decision-making process performed by the BDI agent would run too slowly to be useful. This was tested and validated by profile testing the reasoner and examining the relative length of the agent's deliberation relative to the loading of the agent's perceptions and actions. These results show that the BDI agent's decision-making was not the most significant bottleneck to runtime performance, it ran faster then the functions responsible for loading the perceptions and actions to and from the agent.

5.1. Mission Behaviour

In this section, the mission behaviour of each of the agents is discussed. This serves as a means of validating the generic features that the Agent in a Box provides for the agents goal-directed behaviour. Highlighted in this section is how each of the case study agents used a generic mission level plan, which included a need to navigate to specific points of interest using generic navigation plans. These plans generated a route for the agent to follow. These routes took the form of plans, sequences of waypoint achievement goals. The first case study discussed is the grid agent, discussed in Section 5.1.1. This is followed by the autonomous car in Section 5.1.2 and the mail robot in Section 5.1.3. Each of these agents used generic mission level plans which could be interrupted, suspended, and resumed as needed, but without concern for what those interruptions may be. These interruptions are discussed in the following section.

5.1.1. Grid Agent Mission

The first scenario considered was the grid agent, in both synchronous and asynchronous environments. In this scenario, the agent was at the starting location, at the bottom left of the grid map, called location `a`, and given the task of navigating to the grid square at the bottom right, location `d`, as shown in Figure 14. This can be achieved using the Agent in a Box's navigation mission. As discussed in Section 4.4.1, this meant that the Agent in a Box needed a map and the domain-specific plans for achieving the `waypoint(_)` goal. Using its automatically generated route, the Agent in a Box used the domain-specific plans for moving the agent to each of the locations on the specified route. The implementation of `waypoint(_)` was done without concern for how the agent would handle inaccurate map information, obstacles, and recharging its battery; these concerns have been separated from the implementation of the agent movement plans. The validation of these interrupting behaviours is discussed in Section 5.2.



(**a**) (**b**)

Figure 14. Grid Agent Run: Initial (**a**) and Goal (**b**) states.

The Agent in a Box has been successful at moving the grid agent through its mission using its generic plans. Although this result is encouraging, it is a relatively simple test scenario. To further validate that the Agent in a Box is useful for controlling mobile robots, the Agent in a Box was used to drive a simulated autonomous car. This mission is discussed next.

5.1.2. Autonomous Car Mission

Although the control of the autonomous car is much more complicated that the control of the grid, discussed previously, the Agent in a Box was able to use the same generic mission plans to generate a route to a destination, in the form of `waypoint(_)` achievement goals, and drive the car. With the route generated, the Agent in a Box called the domain-specific implementations of the `waypoint(_)` goal to drive the car down the street, turn the corner, and drive to its destination, as shown in Figure 15.

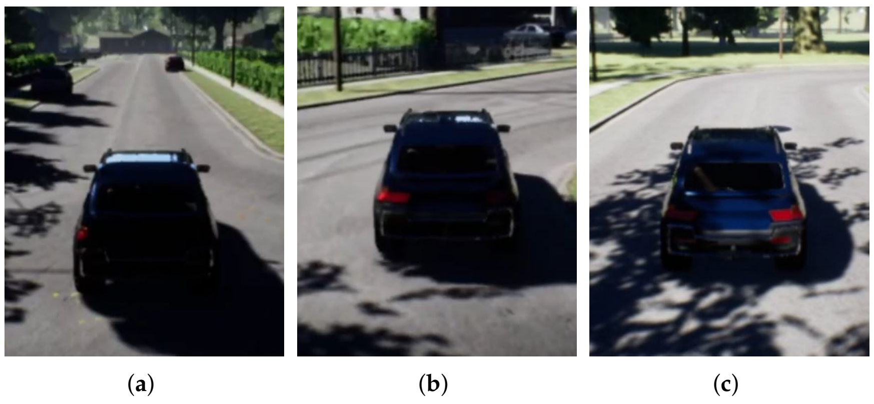


Figure 15. Autonomous Car Navigation: getting route (**a**), turning (**b**), and arriving (**c**).

Although the Agent in a Box was successful at driving the car in the environment and managed to get the car to its destination using its generic navigation plan, this version of the car was still simplistic, using only compass bearings and GPS coordinates to drive the car in the environment. This second test with the car makes use of a distance sensor, pointed forward, and a lane-keep assist camera for tracking the lane markings on the road. The car was then driven in the environment. The goal was to expose how the Agent in a Box works with more complex sensor inputs. This is shown in Figure 16.

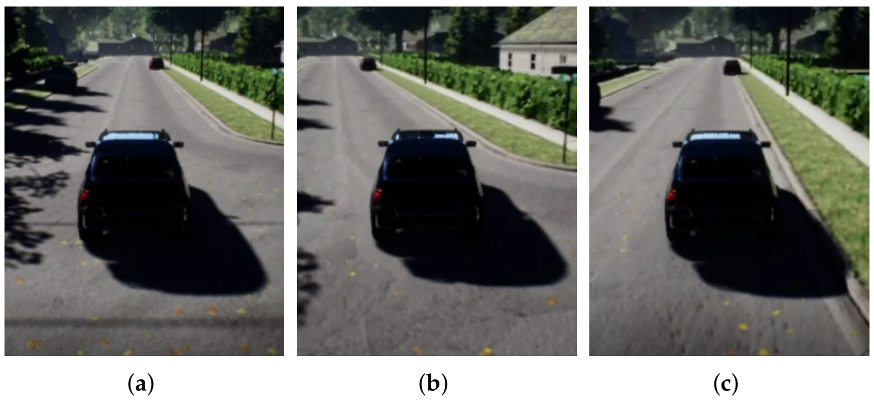


Figure 16. Autonomous Car Lane-Keep: initial (**a**), turning (**b**), and following lane (**c**).

Using the same generic plans, the Agent in a Box was able to drive the car using the lane-keep assist. The only difference was that the modification of the domain-specific plans for achieving the `waypoint(_)` goal. As was the case with the grid agent, these driving plans were implemented without concern for how the car would avoid collisions, which was implemented as a separate behaviour which is validated in Section 5.2.2.

5.1.3. Mail Delivery Mission

So far, the Agent in a Box has controlled both a grid agent and a car by driving them to their destinations using its generic behaviour. That said, in both of these cases, the Agent in a Box used the generic navigation mission that it provides. Although these results have been encouraging, some mobile robots have more complicated missions, where the robot needs to travel to more than one location and do something when it arrives at those locations. This section provides an example where the Agent in a Box was used to control a prototype mail delivery robot.

The primary mission of the mail delivery robot was to move to a sender location to collect mail and then move to a receiver location to deliver it. This is a domain-specific mission, which uses the Agent in a Box's navigation plans. For this to work, the developer provided plans for implementing the mission, the map, and waypoint goals needed by the Agent in a Box. The implementation of the mail agent, using the Agent in a Box, was discussed in Section 4.4.3. Figure 17 shows the prototype mail delivery robot maneuvering in a test environment. Using the domain-specific mission provided, the Agent in a Box used the same generic plans to generate routes between the locations, using the domain-specific plans to drive the robot between waypoints.

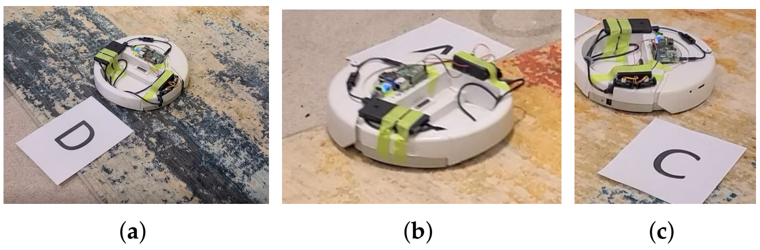


Figure 17. Mail Delivery Mission: initial (**a**), sender location (**b**), and receiver location (**c**).

5.1.4. Mission Behaviour Summary

The Agent in a Box has been demonstrated by controlling a grid agent, a simulated autonomous car, and a prototype mail delivery robot. This was done by leveraging the Agent in a Box's generic navigation plan, which generates a route as a sequence of waypoint achievement goals. In order for a developer to use the Agent in a Box, they must provide a domain-specific map of the environment and implementation of the waypoint achievement goals. If the mission involves more than simply moving to some destination location, as was the case with the mail delivery robot, the developer must provide a mission level plan which implements the desired behaviour, using the Agent in a Box's provided navigation plan when needed. The elegance of the Agent in a Box is that these plans were all implemented without concern for what should be done to avoid collisions, update its map, or manage resources by recharging a battery or refueling. These behaviours have been implemented separately, and are validated in the next section.

5.2. Reactive Behaviour

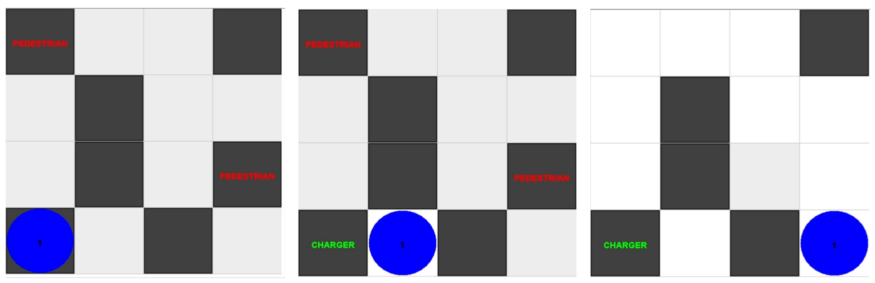
The previous section demonstrated how the Agent in a Box can use a combination of generic and domain-specific plans and beliefs to control a grid agent, a simulated autonomous car, and a prototype mail delivery robot. As before, this provided confidence that the Agent in a Box can successfully control a variety of mobile robots in a variety of environments. Although this is encouraging, these tests focused purely on how the Agent in a Box used goal-directed reasoning, using plans that did not include any behaviour for avoiding collisions, managing resources, or updating the map; yet, despite these behaviours being omitted from the plans for moving the robots, the Agent in a Box is able to successfully handle these types of interruptions. This section provides a validation of this claim.

First, in Section 5.2.1, a scenario where the Agent in a Box was provided an inaccurate map is provided. In this case, the Agent in a Box needed to update its map and regenerate its achievement goals given its updated context in order to complete the mission. Next, in Section 5.2.2, the Agent in a Box is demonstrated in a variety of collision avoidance scenarios. In these cases, the Agent in a Box was moving through the environment and needed to react to a collision risk. Last, in Section 5.2.3, the Agent in a Box attempts to complete its mission with insufficient power in its battery. The battery must be recharged before the mission can be completed.

5.2.1. Map Update

In the map update scenario, the Agent in a Box was demonstrated in the grid environment with an inaccurate map. This scenario is similar to the scenario discussed in Section 5.1.1. The key difference in this case was that the map did not show that there was an obstacle directly between the agent's starting location and the goal location. This meant that the route that the Agent in a Box would generate for moving the agent to the destination would involve attempting to move through an obstacle. As mentioned previously, the domain-specific plans for implementing the waypoint goals for the agent had no checks to verify that the path was clear. Even with this limitation in the implementation of the waypoint plans, the Agent in a Box was able to detect the inaccuracy in the map, update the map, generate a new route, and move to the destination successfully.

A visualisation of the key moments in this run are provided in Figure 18. The Agent in a Box began by generating a route to the destination, as it did in the scenario discussed in Section 5.1.3, however this time the route it generated would send the agent across the bottom of the grid and into the obstacle. When the Agent in a Box perceived this obstacle, the map update plan became applicable as well as the waypoint plan for moving the agent to the inaccessible square, as the waypoint plans do not include obstacle avoidance. As the reasoner prioritises obstacle plans over waypoint plans, the Agent in a Box did not run into the obstacle. Instead, it dropped the incompatible map belief and then used its generic navigation plan to generate a new route. The new route was again generated as a sequence of waypoint achievement goals to the destination.



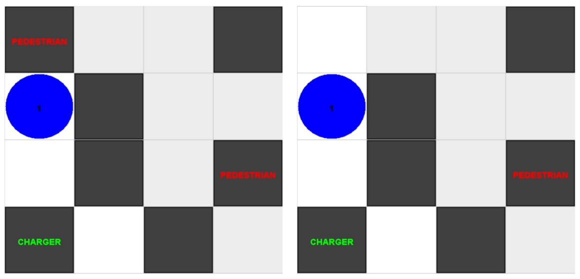
(**a**) (**b**) (**c**)

Figure 18. Grid Map Error: Initial (**a**), at obstacle (**b**), and goal (**c**) States.

This test has validated that the Agent in a Box's provided generic behaviours, and the behaviour prioritisation, appropriately handled a situation where the provided map did not reflect all of the obstacles between the starting and destination location. In this case, the Agent in a Box properly prioritised the map update plan over the waypoint plan, which also did not contain any collision avoidance behaviour in it. Granted, not all collision avoidance requires the updating of the map. The next section provides examples of this type of behaviour.

5.2.2. Collision Avoidance

Each of the case studies contained collision avoidance situations. The first collision avoidance test was performed with the grid agent. In this case, the Agent in a Box needed to move to a destination location, however there were pedestrians that could be blocking the way. As was also the case with the map update, in the previous section, the Agent in a Box was moving through the map by following the waypoint achievement goals generated by the generic navigation plan. These plans did not contain any checks for pedestrians that may be blocking the way. Instead, a high-priority safety plan was provided to the Agent in a Box. Defined in Listing 12, this simple plan forces the agent to honk its horn if pedestrians are perceived. As this was a belief triggered plan, where the its event was identified as triggering a safety behaviour, the reasoner selected this plan to run instead of the waypoint plan, which was also applicable. A visualisation of this is provided in Figure 19.



(**a**) (**b**)

Figure 19. Grid Collision Avoidance: Pedestrian observed (**a**) and pedestrian avoided (**b**).

The Agent in a Box was also in a collision avoidance scenario when it was driving the car. In this case, there was a parked car on the side of the road which needed to be avoided, shown in Figure 20. As was also the case with the grid agent, the implementation of the waypoint goals for the car did not include any checks for possible obstructions. Instead, the Agent in a Box depended on a simple high-priority belief-triggered plan to avoid this obstacle. The obstacle avoidance behaviour, previously provided in Listing 15, is triggered by the perception of an obstacle, detected by the distance sensor. The plan context includes a threshold which ensures that the plan is only applicable if the obstacle is within 5 m of the car. In the first instance, the car had not quite passed that threshold, so the car continued to steer using the lane following. Once the car was within 5 m of the obstacle, the avoidance behaviour was applicable. As this was a triggered by a higher priority event, the reasoner selected this plan for the intention queue over the waypoint plans. The result was that the car turned to avoid the obstacle. Once the obstacle had been avoided, the agent could then steer the car normally using the plans that implement the waypoint goal. This first meant using the compass to realign the car with the direction it should be driving. As the car began to realign, the lane-keep system reacquired the lane and recommended new steering settings for the agent to use.

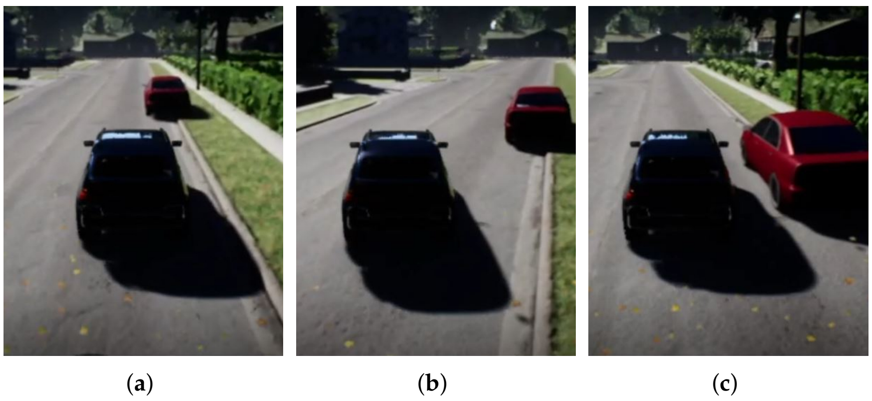


Figure 20. Autonomous Car Collision Avoidance: Approaching (**a**), avoidance (**b**), and recovery (**c**).

Last, the mail agent was also provided with a collision recovery behaviour. In this case, the plan was triggered by the perception of the bumper sensor having been pressed, indicating that the robot had found an obstacle. Again, the trigger for this plan was marked as being at the highest priority, therefore the reasoner selected this plan as the next intention over the waypoint goals which were already running. Figure 21 shows the bumper being manually pressed to trigger the collision avoidance behaviour.

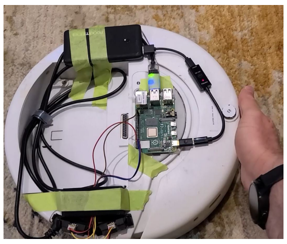


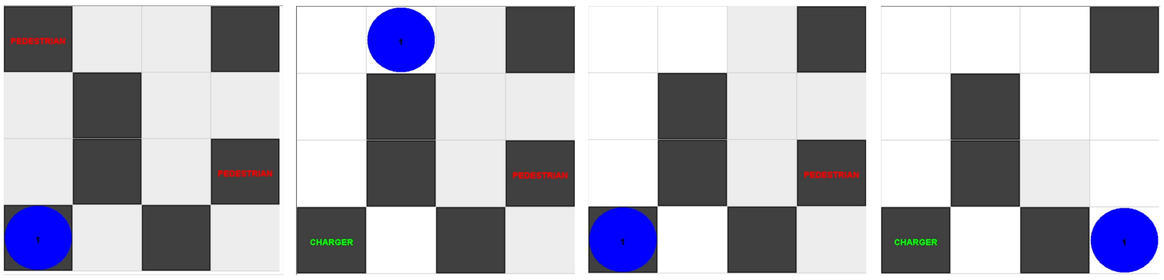
Figure 21. Mail agent: collision recovery.

This section has provided a validation of how the Agent in a Box avoids collisions. This was demonstrated with the grid agent, the simulated autonomous car, and the prototype mail delivery robot. In each of the cases, the Agent in a Box was driving the agent by executing the domain-specific plans that achieve the `waypoint(_)` goal. In all cases, this goal was implemented without any collision avoidance checks. Instead, the Agent in a Box provided the collision avoidance, by using high-priority belief-triggered plan which the reasoner selected over the `waypoint(_)` plans, which were also applicable in those contexts. As a result, the Agent in a Box has provided collision avoidance as an emergent property of the mobile robots that it controls.

5.2.3. Resource Management

As was the case with the map update and collision avoidance behaviours, the Agent in a Box provides resource management behaviour for mobile robots. This functionality was triggered by addition of a `resource(_)` belief, as discussed in Section 4.3.4, which was marked as a health-related plan. With the plan trigger marked with this priority, the reasoner could select this plan over the mission, navigation, or movement plans. This plan's context contains a check to ensure that the resource was depleted, meaning that the plan would not run otherwise. The following paragraphs and figures demonstrate that the Agent in a Box was able to properly interrupt its mission to recharge the robot's battery and then resume the mission. This has been demonstrated with both the grid agents and the prototype mail delivery robot.

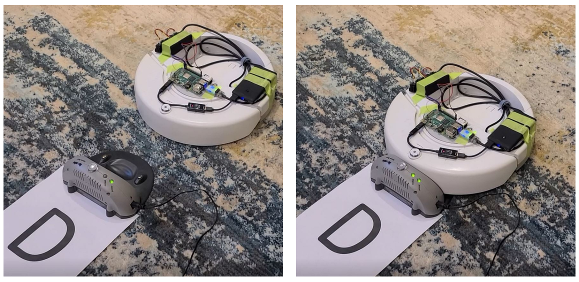
The run instance for the situation where the grid agent was low on battery power is provided in Figure 22. The agent started by dropping intentions, the achievement goals associated navigating to the destination. The Agent in a Box can readopt those goals once the battery has been recharged using the the parameters set in the mental note at the beginning of the agent's run. The agent then set a mental note with respect to the management of the battery, so that the battery charging plans would not keep triggering. The Agent in a Box then navigated the robot to the charging station at location a, using the generic navigation plan, generating a set of waypoint goals and then achieving them using domain-specific plans. Once at the charging station, the Agent in a box docked the robot to the station to recharge. The Agent in a Box waited for the battery to be fully charged before disconnecting from the charging station. It then dropped the mental note that the agent was managing the battery and resumed the previous mission, using the mission level mental note.



(**a**) (**b**) (**c**) (**d**)

Figure 22. Grid Resource Management: Initial (**a**), Low battery (**b**), recharging (**c**), and mission complete (**d**).

The Agent in a Box was also able to recharge the battery of the prototype mail delivery robot. This was accomplished using the same generic plans provided by the Agent in a Box, as shown in Figure 23.



(**a**) (**b**)

Figure 23. Mail Agent Docking to Charge the Battery: At docking station (**a**), docked (**b**).

The Agent in a Box was successful at using generic resource management plans to send the robot to a charging station and recharge the battery. Any interrupted mission could then be readopted so that the Agent in a Box could complete its mission.

5.2.4. Reactive Behaviour Summary

This section provided validation for how the Agent in a Box handled an inaccurate map, collision avoidance, and resource management. These interruptions all occurred while the Agent in a Box was working on its primary mission goal, which, as mentioned earlier, did not include any checks for these types of concerns. Through a combination of reactive, belief-triggered plans, and the appropriate prioritisation of the triggering events associated with those plans in the reasoner, the Agent in a Box was able to provide map updates, collision avoidance, and resource management to the mobile robots that it controlled. The effect of this property was that the domain-specific software needed to only include a map definition, the ability to move the robot between waypoints, and belief-triggered plans for collision avoidance (with the appropriate prioritisation belief). The domain developer also needs to provide specific perceptions and actions for monitoring the resources and docking the robot to use the resource management plans. With these provided, the Agent in a Box could mesh the goal-directed and reactive behaviour required for controlling the robots safely.

5.3. Reasoner Performance Profile

To assess the runtime performance costs of using the Agent in a Box, the performance of the reasoning system was measured using JProfiler. The profiler was configured to take snapshots of each reasoning cycle. The reasoning cycle consisted of three main activities: sensing the environment, deliberating, and taking action. The profile results, showing the duration of each of these aspects of the reasoning cycle, for each agent are provided in Figure 24.

Qualitatively, each agent was observed to have a degraded performance when the agent was run with the profiler. The reasoner was clearly running slower, as the profiler added significant processing burden. Therefore, in examining the profile results, the focus should not be on the absolute length of time that was taken for each part of the reasoning cycle to run, but rather the proportion of time spent in each aspect of the reasoning cycle. The key assessment here is to identify if the deliberation portion of the reasoning cycle is causing a performance bottleneck for the system.

Figure 24a provides the profile results for the synchronised grid agent, where the environment update and the agent reasoning cycle were run in a single thread of execution. In this case, the sense aspect of the reasoning cycle was the fastest, followed by the deliberation. The action phase of the reasoning cycle took the longest, as its performance was tied to the update of the environment visualisation, whereas the sense portion simply queried parameters from the environment as environment objects, a relatively fast process.

Figure 24b provides the performance profiles for the asynchronous grid environment using a desktop computer running Windows 10 with an Intel Core i7-5820K CPU @ 3.30 GHz with 64 GB of system RAM and an NVIDIA GTX 970 with 4 GB of RAM. This test was also run on a Raspberry Pi 4, as shown in Figure 24c. As the asynchronous grid environment was connected to the agent using ROS, the environment update was not tied to the agent's reasoning cycle. There is a clear performance difference between the desktop computer and the Raspberry Pi computer. The action portion of the reasoning cycle was still the longest portion of the reasoning cycle. Again, the reasoner was not a bottleneck for the performance.

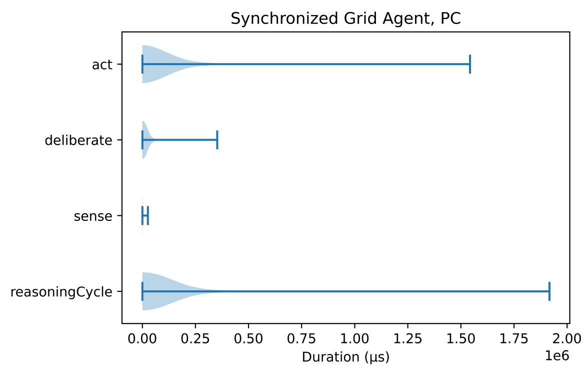
The sensing aspect of the reasoning cycle dominates the execution of both the autonomous car agent and the mail delivery agent, as shown in Figure 24d,e respectively. This was the result of the agent waiting for new perception information before proceeding to the rest of the reasoning cycle. This implies that the agent is able to outperform the sensor update rate for both the car and the mail robot. The agent's deliberation was not a bottleneck for the system performance.

5.4. Results Summary

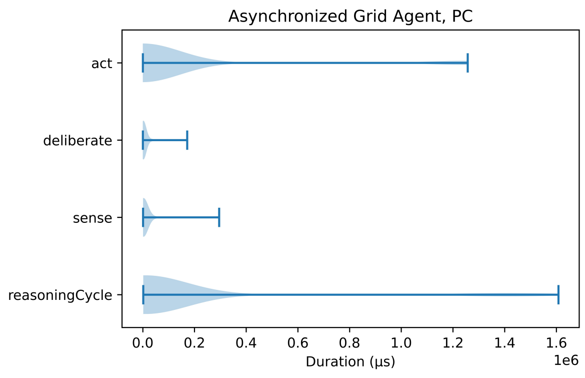
Detailed results of the validation experiments of the use of BDI for controlling autonomous robotics were presented in this section. The environments included a synchronised and asynchronised grid environment, an autonomous car simulated with AirSim, and a prototype mail delivery robot.

The first validation tests presented focused on the properties of the Agent in a Box. This included the behaviour framework, which controls mobile robots using generic plans that provide the robot with navigation, resource management, and collision avoidance. An application domain developer need only provide a map for of the environment, the domain-specific plans for moving the agent between waypoints (without concern for obstacle avoidance or resource management), and belief-triggered plans for collision avoidance; also needed are specific perceptions and actions for the Agent in a Box to be able to dock the robot to replenish a resource, such as a depleted battery. Using its behaviour framework and prioritisation functions, the Agent in a Box was successful at meshing the combination of generic and domain-specific plans as well as goal-directed and reactive behaviours to successfully control each of the robots tested. In effect, the Agent in a Box was found to be a useful method for implementing the behaviour of mobile robots, reducing the development burden for the developer.

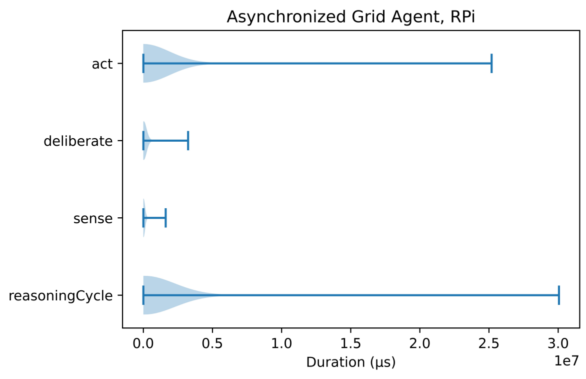
With the desired behaviour of the Agent in a Box validated, the concern shifted to the runtime performance of the Agent in a Box, specifically of the agent's reasoning cycle. This was accomplished by profile testing of the reasoner and demonstrating that the reasoner's decision-making was not the primary bottleneck to runtime performance.



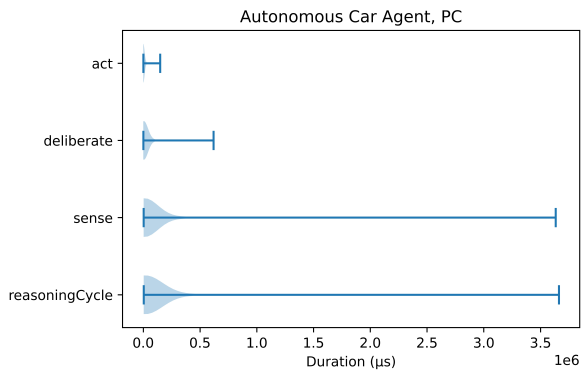
(**a**)



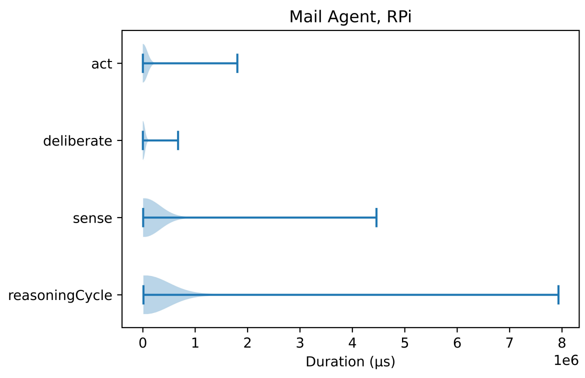
(**b**)



(**c**)



(**d**)



(**e**)

Figure 24. Reasoning system performance profiles textit(Continued from previous page): Synchronized Grid on PC (**a**), asynchronized grid on PC (**b**), asynchronized grid on RPi (**c**), autonomous car on PC (**d**), mail agent on RPi (**e**).

6. Discussion

This work has demonstrated the Agent in a Box, a framework for developing autonomous mobile robots with BDI. The framework provides the behaviour common to a variety of different mobile robots, such as mission management, navigation, map update, and resource management. A developer for a specific application domain need only provide the behaviour needed for the movement and obstacle avoidance of their specific robot. Additionally provided by the Agent in a Box is a modification to the reasoning system so that the appropriate behaviour is prioritised. This prioritisation was inspired by the Subsumption Architecture, which was discussed in Section 2.4. The Agent in a Box provides a means for connecting the agent to the environment with SAVI ROS BDI and using a set of sensors, actuators, and translator nodes. This method of connecting to the environment bears some similarity to the Abstraction Engines method, discussed in the state-of-the-art.

This work builds on the state-of-the-art through the generalisation of the approach for different mobile robots. Additionally provided was a framework for the behaviour common to various mobile robots, the customised reasoning system, and the means of connecting the agent to the environment. None of the works discussed in the state-of-the-art, in Section 3, has provided all of these elements. Furthermore, this work demonstrated that the Agent in a Box can successfully control different robots while making good use of the framework's features meshing a combination of goal-directed and reactive behaviours and using a combination of generic and domain-specific agent code. The agents were demonstrated to successfully complete their missions while avoiding obstacles. Through profiling, it was demonstrated that agent deliberation was not a performance bottleneck on either a desktop computer and on a Raspberry Pi.

Although this approach worked well with the robots that were tested, it cannot be claimed that the agent will work with every possible mobile robot. Granted the results are encouraging, it is necessary to extend this work to demonstrate that the approach can be useful in more application domains. One planned future test is to control a drone for infrastructure inspection and precision agriculture, demonstrating that the approach is useful with not only terrestrial robots but aerial robots. Additionally, work is planned to move beyond the use of simulated and prototype robots by driving a real car and by actually delivering mail using an upgraded mail robot. Work is also planned on developing a method for improving the ability of the agent to provide an explanation of its state of mind and activities. This is useful for development as well as for gaining the trust of users who may appreciate the agent providing an explanation of its behaviour. Last, this paper did not consider the development of human-like behaviour for robots, nor did it include any machine learning components. The Agent in a Box depends on being provided suitable AgentSpeak implementations of the `waypoint(_)` goal, among other requirements, in order to function. The development of more complicated behaviour, where the agent may need to consider the social acceptability of its actions, or perform more complex navigation, perhaps with the use of the ROS navigation stack, have not been examined. These are proposed as potential future directions for this work.

7. Conclusions

This paper has provided the Agent in a Box, a framework for controlling autonomous mobile robots using BDI. SAVI ROS BDI was used for connecting Jason's reasoning system to the environment using ROS. Included in this node is a behaviour prioritisation, inspired by the Subsumption Architecture, which ensures that the agent's highest priority behaviour was selected for any given context. The approach provides the nodes needed for the agent to perceive the environment using a variety of sensors as well as control a variety of different actuators. The Agent in a Box included behaviour common to all mobile robotic agents, including mission management, navigation, map update, and resource management, and guidance for how a developer should design the behaviour for moving the robot and collision avoidance for their specific application. This allows the Agent in a

Box to use the domain-specific plans for controlling the robot while also using the common behaviours provided by the Agent in a Box when appropriate. The Agent in a Box was demonstrated using three application domains: a grid-based environment, an autonomous car simulated with AirSim, and a prototype mail delivery robot. The behaviour of the agents was demonstrated using the run-time logs generated by the agent and through profiling of the agent's reasoner. These showed that the agents were successful at completing their missions while also handling interruptions appropriately. The reasoner's deliberation was not a performance bottleneck.

Author Contributions: Conceptualisation, P.G. and B.E.; Methodology, P.G.; Software, P.G.; Validation, P.G.; Formal analysis, P.G.; Investigation, P.G.; Resources, P.G., and B.E.; Data curation, P.G.; Writing–original draft preparation, P.G. and B.E.; Writing–review and editing, P.G. and B.E; Visualisation, P.G.; Supervision, B.E.; Project administration, B.E.; Funding acquisition, P.G. and B.E. Both authors have read and agreed to the published version of the manuscript.

Funding: 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].

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All software developed for this study is available open source. The SAVI ROS BDI software is available at [48]. The synchronized grid software is available at [51]. The asynchronized grid software is available at [52]. The autonomous car software is available at [57]. The prototype mail delivery robot software is available at [63]. The scripts and tools used for the profiling experiments are provided at [67].

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

AI	Artificial Intelligence
AOP	Agent-Oriented Programming
BDI	Beliefs-Desires-Intentions
EBNF	Extended Backus–Naur form
GPS	Global Positioning System
LIDAR	laser imaging, detection, and ranging
MAS	Multi-Agent System
PROFETA	Python RObotic Framework for dEsigning sTrAtegies
RAM	Random Access Memory
ROS	Robot Operating System
SAVI	Simulated Autonomous Vehicle Infrastructure
SLAM	Simultaneous Localisation and Mapping
UAV	Unmanned Aerial Vehicle

References

1. Bordini, R.H.; Hübner, J.F.; Wooldridge, M. *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*; John Wiley &; Sons Ltd.: England, UK, 2007.
2. Bratman, M. *Intention, Plans, and Practical Reason*; Harvard University Press: Cambridge, MA, USA, 1987; Volume 10.
3. Rao, A.S.; George, M.P. BDI agents: From theory to practice. In Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, CA, USA, 12–14 June 1995; pp. 312–319.
4. Bordini, R.H.; El Fallah Seghrouchni, A.; Hindriks, K.; Logan, B.; Ricci, A. Agent programming in the cognitive era. *Auton. Agents-Multi Syst.* **2020**, *34*. [CrossRef]
5. Multi-Agent Programming Contest. Available online: https://multiagentcontest.org/2019/ (accessed on 28 May 2019).

6. Hofmann, P.; Lettmayer, P.; Blaschke, T.; Belgiu, M.; Wegenkittl, S.; Graf, R.; Lampoltshammer, T.J.; Andrejchenko, V. Towards a framework for agent-based image analysis of remote-sensing data. *Int. J. Image Data Fusion* **2015**, *6*, 115–137. [CrossRef] [PubMed]
7. Hübner, J.F.; Bordini, R.H. Jason: A Java-Based Interpreter for an Extended Version of AgentSpeak. Available online: http://jason.sourceforge.net (accessed on 16 February 2019).
8. Rao, A.S. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*; Van de Velde, W., Perram, J.W., Eds.; Springer: Berlin, Germany, 1996; pp. 42–55.
9. Johnson, R.E. Documenting Frameworks Using Patterns. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA '92; Association for Computing Machinery: Vancouver, BC, Canada, 18-22 October1992; pp. 63–76. [CrossRef]
10. Johnson, R.E. Documenting Frameworks Using Patterns. *Sigplan Not.* **1992**, *27*, 63–76. [CrossRef]
11. Riehle, D. Framework Design: A Role Modeling Approach. Ph.D. Thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2000.
12. Fowler, M. Inversion Of Control. Available online: https://martinfowler.com/bliki/InversionOfControl.html (accessed on 10 May 2021).
13. Shoham, Y. Agent-oriented programming. *Artif. Intell.* **1993**, *60*, 51–92. [CrossRef]
14. AOSGroup. JACK. Available online: http://www.aosgrp.com/products/jack/ (accessed on 4 February 2019).
15. Bordini, R.H.; Hübner, J.F.; Wooldridge, M. Programming Multi-Agent Systems in AgentSpeak Using Jason (Lecture Slides). 2007. Available online: http://jason.sourceforge.net/jBook/SlidesJason.pdf (accessed on 27 June 2019).
16. Aschermann, M.; Kraus, P.; Müller, J.P. LightJason: A BDI Framework Inspired by Jason. In *Technical Report IfI Technical Report IfI-16-04*; Department of Computer Science, TU Clausthal, Clausthal-Zellerfeld, Germany, 2016.
17. LightJason. Available online: https://lightjason.org/ (accessed on 18 March 2019).
18. JaCaMo Project. Available online: http://jacamo.sourceforge.net/ (accessed on 16 May 2019).
19. Muller, B.; Dennis, L. Gwendolen: A BDI Language for Verifiable Agents. In Proceedings of the AISB 2008 Symposium: Logic and the Simulation of Interaction and Reasoning, Aberdeen, Scotland, 3–4 April 2008.
20. Open Source Robotics Foundation. ROS. Available online: https://www.ros.org/ (accessed on 27 May 2019).
21. Rusu, R.B.; Cousins, S. 3D is here: Point Cloud Library (PCL). In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA): Shanghai, China, 9–13 May 2011.
22. Rusu, R.B.; Cousins, S. Point Cloud Library. Available online: https://pointclouds.org/ (accessed on 6 May 2019).
23. Coleman, D.; Şucan, I.; Chitta, S.; Correll, N. Reducing the Barrier to Entry of Complex Robotic Software: A MoveIt! Case Study. *J. Softw. Eng. Robot.* **2014**, *5*, 3–16. [CrossRef]
24. Sucan, I.A.; Chitta, S. MoveIt. Available online: https://moveit.ros.org/ (accessed on 6 May 2019).
25. Available online: move_base. http://wiki.ros.org/move_base (accessed on 19 April 2021).
26. Marder-Eppstein, E.; Berger, E.; Foote, T.; Gerkey, B.; Konolige, K. The Office Marathon: Robust Navigation in an Indoor Office Environment. In Proceedings of the International Conference on Robotics and Automation: Anchorage, Alaska, 3-8 May 2010.
27. Navigation. Available online: http://wiki.ros.org/navigation (accessed on 14 June 2021).
28. Truong, X.T.; Ngo, T.D. Toward Socially Aware Robot Navigation in Dynamic and Crowded Environments: A Proactive Social Motion Model. *IEEE Trans. Autom. Sci. Eng.* **2017**, *14*, 1743–1760. [CrossRef]
29. Brooks, R. A robust layered control system for a mobile robot. *IEEE J. Robot. Autom.* **1986**, *2*, 14–23. [CrossRef]
30. Wooldridge, M. *An Introduction to MultiAgent Systems*, 2nd ed.; John Wiley &; Sons Ltd.: Chichester, UK, 2009.
31. Wallis, P.; Ronnquist, R.; Jarvis, D.; Lucas, A. The automated wingman - Using JACK intelligent agents for unmanned autonomous vehicles. In Proceedings of the IEEE Aerospace Conference, Big Sky, MT, USA, 9–16 March 2002; Volume 5, p. 5. [CrossRef]
32. Karim, S.; Heinze, C. 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, 25–29 July 2005; pp. 19–26. [CrossRef]
33. Menegol, M.S.; Hübner, J.F.; Becker, L.B. Evaluation of Multi-agent Coordination on Embedded Systems. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection*; Demazeau, Y., An, B., Bajo, J., Fernández-Caballero, A., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 212–223.
34. Menegol, M.S. vooAgente4Wp. Available online: https://drive.google.com/file/d/0B7EcHgES6He8VEtwR0xPZjdBbk0/view (accessed on 8 May 2019).
35. Rezende, G.; Hubner, J.F. Jason-ROS. Available online: https://github.com/jason-lang/jason-ros (accessed on 24 May 2019).
36. Rezende, G. MAS-UAV. Available online: https://github.com/Rezenders/MAS-UAV (accessed on 24 May 2019).
37. Wesz, R. Integrating Robot Control Into The AgentSpeak(L) Programming Language. Master's Thesis, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil 2015. Available online: http://repositorio.pucrs.br/dspace/bitstream/10923/9007/1/000480471-Texto%2BCompleto-0.pdf (accessed on 1 September 2021).
38. Fichera, L.; Messina, F.; Pappalardo, G.; Santoro, C. A Python framework for programming autonomous robots using a declarative approach. *Sci. Comput. Program.* **2017**, *139*, 36–55. [CrossRef]
39. Eurobot Association. Eurobot: International Students Robotic Contest. Available online: http://www.eurobot.org/ (accessed on 15 July 2019).
40. Unict Team. Unict Team Website. Available online: http://unict-team.dmi.unict.it/ (accessed on 15 July 2019).

41. K. C., U.; Chodorowski, J. A Case Study of Adding Proactivity in Indoor Social Robots Using Belief–Desire–Intention (BDI) Model. *Biomimetics* **2019**, *4*, 74. [CrossRef]
42. Pantoja, C.E.; Stabile, M.F.; Lazarin, N.M.; Sichman, J.S. ARGO: An Extended Jason Architecture that Facilitates Embedded Robotic Agents Programming. In *Engineering Multi-Agent Systems*; Baldoni, M., Müller, J.P., Nunes, I., Zalila-Wenkstern, R., Eds.; Springer International Publishing: Cham, Switzerland, 2016; pp. 136–155.
43. Lazarin, N.M.; Pantoja, C.E. A robotic-agent platform for embedding software agents using raspberry pi and arduino boards. In Proceedings of the 9th Software Agents, Environments and Applications School, Niterói, Brazil, 1–3 June 2015.
44. Dennis, L.A.; Aitken, J.M.; Collenette, J.; Cucco, E.; Kamali, M.; McAree, O.; Shaukat, A.; Atkinson, K.; Gao, Y.; Veres, S.M.; et al. Agent-Based Autonomous Systems and Abstraction Engines: Theory Meets Practice. In *Towards Autonomous Robotic Systems*; Alboul, L., Damian, D., Aitken, J.M., Eds.; Springer International Publishing: Cham, Switzerland, 2016; pp. 75–86.
45. 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*; Bassiliades, N., Chalkiadakis, G., de Jonge, D., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 191–205.
46. Davoust, A.; Gavigan, P.; Ruiz-Martin, C.; Trabes, G.; Esfandiari, B.; Wainer, G.; James, J. An Architecture for Integrating BDI Agents with a Simulation Environment. In *Engineering Multi-Agent Systems*; Dennis, L.A., Bordini, R.H., Lespérance, Y., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 67–84.
47. Singh, D.; Padgham, L.; Logan, B. Integrating BDI Agents with Agent-Based Simulation Platforms. *Auton. Agents Multi Syst.* **2016**, *30*, 1050–1071. [CrossRef]
48. Gavigan, P. SAVI_ROS_BDI. Available online: https://github.com/NMAI-lab/savi_ros_bdi (accessed on 1 September 2021).
49. Gavigan, P.; Esfandiari, B. BDI for Autonomous Mobile Robot Navigation. In Proceedings of the 9th International Workshop on Engineering Multi-Agent Systems, London , UK, 3–4 May 2021.
50. Hubner, J.F. Jason Search Demo. Available online: https://github.com/jason-lang/jason/tree/master/demos/search. (accessed on 19 February 2021).
51. Gavigan, P. Jason Mobile Agent. Available online: https://github.com/NMAI-lab/jasonMobileAgent (accessed on 19 February 2021).
52. Gavigan, P. Jason Mobile Agent ROS. Available online: https://github.com/NMAI-lab/jason_mobile_agent_ros (accessed on 19 February 2021).
53. Gavigan, P. Agent in a Box Demo-Grid Environment. Available online: https://youtu.be/bsr3K4U3wd8 (accessed on 19 February 2021).
54. AirSim. Available online: https://github.com/Microsoft/AirSim (accessed on 27 March 2019).
55. Shah, S.; Dey, D.; Lovett, C.; Kapoor, A. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. Field and Service Robotics. *arXiv* **2017**, arXiv:1705.05065.
56. Yao, S. Autonomous-driving vehicle test technology based on virtual reality. *J. Eng.* **2018**, *2018*, 1768–1771. [CrossRef]
57. Gavigan, P. AirSim Navigating Car. Available online: https://github.com/NMAI-lab/AirSimNavigatingCar (accessed on 19 February 2021).
58. Gavigan, P. AirSim Car BDI Agent. Available online: https://youtu.be/yX20gJjjbMg (accessed on 19 February 2021).
59. Gavigan, P. Agent in a Box Demo-Car Lane Keep and Obstacle Avoidance. Available online: https://youtu.be/tvqkNnpKIPo (accessed on 5 April 2021).
60. Onyedinma, C.; Gavigan, P.; Esfandiari, B. Toward Campus Mail Delivery Using BDI. In Proceedings of the First Workshop on Agents and Robots for reliable Engineered Autonomy, Virtual Event, 4 September 2020; Electronic Proceedings in Theoretical Computer Science; Cardoso, R.C., Ferrando, A., Briola, D., Menghi, C., Ahlbrecht, T., Eds.; Open Publishing Association: Waterloo, NSW, Australia, 2020; Volume 319, pp. 127–143. [CrossRef]
61. Onyedinma, C.; Gavigan, P.; Esfandiari, B. Toward Campus Mail Delivery Using BDI. *J. Sens. Actuator Netw.* **2020**, *9*, 56. [CrossRef]
62. Perron, J. create_autonomy. Available online: http://wiki.ros.org/create_autonomy (accessed on 8 March 2020).
63. Gavigan, P.; Onyedinma, C. saviRoomba. Available online: https://github.com/NMAI-lab/saviRoomba (accessed on 9 May 2020).
64. Gavigan, P. Mail Agent-Mail Mission. Available online: https://youtu.be/4nVOVI1GJOM (accessed on 19 July 2021).
65. Gavigan, P. Mail Agent-Collision Recovery. Available online: https://youtu.be/bKHR-DaXZq0 (accessed on 19 July 2021).
66. Gavigan, P. Mail Agent-Docking to Recharge Battery. Available online: https://youtu.be/hvq_vduv-OM (accessed on 19 July 2021).
67. Gavigan, P. savi_profiling. Available online: https://github.com/NMAI-lab/savi_profiling (accessed on 1 September 2021).