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

by

Patrick Gavigan

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Carleton Universty
Ottawa, Ontario

Abstract

This thesis investigates the use of Beliefs-Desires-Intentions (BDI) for controlling autonomous mobile robots. Autonomous systems should ideally be designed for flexibility so that they can intelligently react to ever-changing environments and operational conditions. If they are given such flexibility, they can accept goals and set a path to achieve these goals in a self-responsible manner while displaying some form of intelligence. Developers of autonomous mobile robotics are interested in how to program mobile robots while guaranteeing reliability and resilience.

In investigating the use of BDI for controlling autonomous mobile robots, the Agent in a Box framework was developed. This framework includes a general architecture for connecting a Jason agent to the environment using Robot Operating System (ROS). This connection is done in a way that is flexible to a variety of application domains that use different sensors and actuators. The Agent in a Box provides the needed customization to the agent's reasoner to ensure that the agent's behaviours are properly prioritized. Generic plans for behaviours that are common to a variety of mobile robots are also provided. These include plans for resource management and for navigation, which generates a route to a destination in the form of a plan which can be monitored and interrupted by the reasoner if needed. These components allow developers, for specific application domains, to focus on domain-specific code. Agents implemented using the Agent in a Box are rational, mission capable, safety conscious, fuel autonomous, and understandable. The Agent in a Box was used to demonstrate the capability of BDI agents to control robots in a variety of application domains through a set of case studies: a grid environment, a simulated autonomous car, and a prototype mail delivery robot. These case studies demonstrated that the agent was able to successfully control the robots in all of the application domains tested, including when it was run on a Raspberry Pi computer. By implementing the robots with the Agent in a Box, the development burden was reduced because the framework provided generic behaviours that could be used by each of the agents. The Agent in a Box was also found to enforce good software engineering practices, resulting in a noticeable improvement in runtime performance compared to other approaches that did not use the Agent in a Box.

Acknowledgements

I would like to thank my supervisor, Babak Esfandiari, and the members of my evaluation committee for their time, effort, support, and advice.

○ ○ ○

Thank you to my family, especially to my wife Catharina and my parents Jacqueline and Neil for all of the help and support. To my daughters, Gemma and Maeve, who were both born while I was working on this degree, thank you for being small and adorable, even if you did not make my progress on this thesis go any faster.

○ ○ ○

I would also like to thank a number of colleagues who have provided help along the way. They include (in no particular order) Sacha Gunaratne, Chidiebere Onyedinma, Calvin Jary, Michael Vezina, Jason Miller, Alan Davoust, Cristina Ruiz Martin, Guillermo Trabes, Simon Yacoub, Hamna Manzoor, Jaskaran Singh, James Baak, and Maaike Gooderham.

○ ○ ○

I acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and Defence Research and Development Canada (DRDC).

Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) et Recherche et développement pour la défense Canada (RDDC).

○ ○ ○

Lastly, thank you to all of the health care and front line workers who helped us all stay as safe as possible through the COVID-19 pandemic.

Contents

Figures

Tables

Listings

Chapter 1

Introduction

The field of autonomous mobile robotics, and autonomous agents in general, 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 but not limited to defence, security, infrastructure inspection, passenger travel, freight, and mail delivery. Recently, the COVID-19 pandemic has put an enormous burden on logistics due to the need to minimize human interactions. Widespread adoption of autonomous mobile robotics could ease this burden for a variety of domains, including activities as simple as a visit to the store. For this work, autonomous mobile robots are considered to be a robotic vehicle that is able to move in its environment on its own in order to complete some tasks while maintaining the safety of itself and others around it.

An autonomous agent can be defined as a system that pursues its own agenda, affecting what it perceives in the future, by sensing the environment and acting on it over time [3]. Autonomous systems should be designed in such a manner that they can intelligently react to ever-changing environments and operational conditions. If given such flexibility, they can accept goals and set a path to achieve these goals in a self-responsible manner while displaying some form of intelligence. Developers of autonomous mobile robotics are interested in how to program these systems while guaranteeing reliability and resilience.

Agents implemented using *Beliefs-Desires-Intentions (BDI)* are rational, where an individual agent maintains commitments to various goals that it believes are achievable and drops goals that it either believes are no longer possible or no longer required [3]. Defined using a declarative symbolic programming language [9], these agents maintain a set of *beliefs* about themselves and their environment, and have *desires*, or goals, that they need to achieve. To achieve their goals they have a set of plans, in the form of 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*. The plans can

be monitored, executed, suspended, and resumed depending on the agent's changing context. The agent's ability to do this is an attractive property for enabling the agent to be resilient while working in a partially-observable, unpredictable, and dynamic environment. Assuming a correct symbolic representation of the environment and the agent's behaviours, the agent will select plans that move the agent closer to its achievable goals.

The BDI paradigm was first developed in the field of cognitive science by Bratman in the 1980s as a means of modeling human agency [10]. This model of human agency was adapted by Rao and Georgeff for use in computer science using modal logic to underpin rational decision making [11]. Bordini et. al. [12] highlighted that BDI agents have distinct advantages in that they provide the capability of interleaving plan and action selection and can recover gracefully from failure. They 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 it is going to do next. However, they also highlighted several limitations, such as the need for a developer to define the agent's plans because the agent lacked the ability to generate plans on its own, and the needed effort to adapt the default BDI reasoning cycle for advanced applications. There are several implementations of the BDI paradigm. One of the most popular is Jason [3, 13], which uses the AgentSpeak language [14] for defining the agent's beliefs, goals, and plans.

There are many other ways the software for autonomous mobile robotic systems can be developed. Software developers who work more commonly with more popular procedural or object oriented languages, such as C, C++, Python, or Java may suggest that they could develop software using these approaches which could also provide the desired results. In short, they would be correct. Declarative languages such as AgentSpeak, however, come with the strengths which underpin logic-based systems and rational decision making using means-end reasoning. By programming in this way the developer can focus on *what* the agent needs to do rather than *how* the agent needs to do it. Custom made procedural or object oriented software would not provide these properties without additional effort to validate them. Declarative approaches provide the tools for handling rational reasoning behind the scenes. With these advantages of using BDI, and declarative programming in general, one could expect that this would be a popular way of implementing autonomous mobile robots; however, as will be seen in the state of the art section, there has been a limited amount of work in this area.

Another alternative for implementing mobile robotic systems is the use of machine learning. Indeed, 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 an agent. Furthermore, the deep learning

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. Although there are efforts to improve the explainability of black box methods, such as those outlined by Xie et. al. for deep learning [15], these challenges complicate the development of such models. A simpler alternative is proposed using BDI agents, which use a declarative language with transparency into the agent's mental state, what the agent is doing and why, at any point in time. The mental state is defined using high-level concepts, including beliefs, desires, and intentions, which should make the activities of these agents easier to understand, although there are research activities aimed at improving explainability [16, 17].

As mentioned in the last two paragraphs, BDI offers a number of properties that seem well suited for autonomous mobile robots. However, as will be seen in the state of the art, in chapter 3, there has been a limited amount of work in this area. Of the work that has been done, there seems to be a common concern that the agent's reasoner may struggle to keep up with the sensor data and the demands of controlling a robot in real time. Another possible reason for the seemingly limited work in this area could have to do with the complexity of working with BDI agents or the AgentSpeak language. Perhaps working this way results in more complex and less maintainable software. If this is the case, having a framework which provides a number of generic components may help improve some of these properties. In this work, the use of the BDI paradigm for mobile robotics has been explored. This included using the Jason reasoning system and the AgentSpeak language to answer the following research questions:

Research Questions

- Can a BDI agent framework be used for developing and controlling autonomous mobile robots?
- How would such a framework address the limitations identified by Bordini et. al. [12]?
- Can it be expected that agents developed with such a framework would work well in environments representative of the real world?
- Do agents implemented with the Agent in a Box benefit from improved ease of development? What metrics can be used for assessing this?
- Are there performance trade-offs for using the Agent in a Box? If so, what are they?

In response to these research questions, the Agent in a Box was developed and demonstrated. Its name comes from the idea that the Agent in a Box would be a hardware component that could be connected to any mobile robot and configured to use that robot's specific sensors and actuators in order to control it. The Agent in a Box handles several *generic* aspects of

mobile robotic agents. This includes a general architecture for connecting a Jason agent to the environment using Robot Operating System (ROS). It also includes generic behaviour plans, which provide the agent with mission management, navigation, safety, and energy autonomy. The generic navigation plan generates a route for the robot to its destination in the form of a plan. These plans have been designed to be mapped to domain-specific plans, allowing the development of mobile robotic agents for a variety of domains. The Agent in a Box provides customization to the Jason BDI reasoner, which appropriately prioritizes the agent's available plans so that the most appropriate plan is selected in the event that more than one is applicable to the agent's context.

The Agent in a Box has been used for several case studies with different domains, using different robots equipped with different sensors and actuators, highlighting the flexibility and performance of this approach. These domains include a basic grid based environment, a simulated autonomous car, and a prototype mail delivery robot. The components that a developer for a specific robot needs to provide have been highlighted.

1.1 Requirements for an Agent in a Box

In considering how to answer the research questions posed in the previous section, it is first necessary to consider the requirements of agents for controlling mobile robots. This includes how the agent connects to the robots' sensors and actuators, reasoning system performance, and the agent's behaviour. The focus of this thesis is to control mobile robots with BDI agents and to investigate the use of a framework for doing so. This means that the framework needs to work for a variety of mobile robots, providing generic features that can be customized for use in specific domains. Ideally, the Agent in a Box should make use of commonly used tools which should be familiar to developers working in the field of mobile robots.

For the Agent in a Box to be useful for controlling mobile robots, it needs to be able to receive data from the sensors that are provided by that robot and then control the robot using its domain-specific actuators. Mobile robots may also be equipped with different payloads, with even more unique sensors and actuators needed for the robot's specific mission. These sensors can range from those that provide the agent with simple telemetry, such as a compass or position sensor, to something much more complicated, such as a camera providing image data that needs to be processed. Actuators can range in complexity as well, from being controllable with simple commands, such as commanding an agent to stop moving, to more complex types of actuators that need continuous management, such as the accelerator and brakes on a car. Therefore, the Agent in a Box needs to be flexible so that a variety of sensors or actuators can be connected,

ideally without a need for significant changes to the rest of the system. Furthermore, this flexibility should be achieved using familiar tools such as ROS, a popular software framework for robotics which will be discussed in more detail in section 2.5. The flexibility of the framework can be validated by applying the Agent in a Box to a variety of different mobile robots, both real and simulated, which use different types of sensors and actuators.

Once the agent has been connected to the robot's sensors and actuators, it needs to reason about its perceptions and control the actuators in a timely way. As will be seen in more detail in section 2.3, the agent's reasoning cycle starts with it perceiving the environment, in this case using the robot's sensors. The agent then deliberates about these perceptions in the context of its other beliefs, goals, and plans, followed by the agent taking an action before repeating the reasoning cycle. This reasoning, implemented in Jason's reasoner, takes time to execute. There is therefore a risk that the reasoner may not be able to keep up with the demands of the real world. Naturally, in order for a BDI agent to be successful in controlling the robot, the reasoner needs to keep up with the robot's sensor data. Depending on the robot, the reasoner may need to do this on a computationally constrained system. The reasoner therefore should be demonstrated on a computationally constrained system as part of the case studies. Through the case studies, the behaviour of the robot and the agent's reasoning rate has been monitored and observed to verify that it is behaving properly and with sufficient performance. Performance benchmark tests have also been used to measure the performance difference of using the Agent in a Box approach compared to other agent designs. Profile testing was performed to identify performance bottlenecks in the agent's reasoning cycle. Using these testing methods provides both qualitative and quantitative results for assessing the Agent in a Box. This testing found that although an imperative agent could outperform a BDI agent, it was possible for a BDI agent to control mobile robots on constrained systems. It also found that the deliberation of rules was a source of significant processing time. Therefore, there needs to be care taken in how the agent is designed so that the reasoner does not need to needlessly deliberate on rules that are not relevant to a specific situation.

With the connection between the agent and the robot established, and the need for the reasoner to keep up with the demands of the mission having been considered, it is necessary to consider what the agent should do. This includes how the agent's behaviour is defined and how the plans that implement that behaviour are selected as the agent's intentions. There are a number of concerns that need to be addressed, tested, and studied. For example, the agent must adapt and act appropriately in the presence of conflicting and changing mission objectives. These changes in objectives could include short term issues such as the avoidance of unexpected obstacles or longer term issues such as seeking maintenance when needed. Those interruptions

need to be managed all while navigating through the environment to achieve the agent's primary mission. Some of these behaviours are generic, useful for a variety of applications and domains. The Agent in a Box should handle these generic behaviours, allowing the developer to focus their work on the aspects that are unique to their domain. Among these generic behaviours is a plan for navigation which generates a route to the robot's destination in the form of a plan that can be monitored and interrupted by the reasoner as needed. This separation of concerns between the generic and domain specific plans has been validated by using the Agent in a Box to control mobile robots in different domains during various case studies. Using these case studies, the robots were tested to verify that they properly handle obstacles, low battery, map errors, etc., while trying to complete their mission. There are performance trade-offs with respect to where the agent's behaviour should be implemented. For example, should they be implemented as plans within the agent in AgentSpeak, as internal actions (Java functions that are called within the agent's mind), or externally to the agent as separate modules in the environment. To assess this trade-off a comparative test was conducted to identify if there was a significant difference in the performance of the agent depending on the way that the behaviour was implemented. Lastly, the software engineering properties, including coupling, cohesion, and cyclomatic complexity, of the Agent in a Box relative to agents implemented without the Agent in a Box were assessed.

Table 1.1: Summary of Agent in a Box Requirements.

Connecting to the Environment	Agent Behaviour
• Flexible for different sensors and actuators • Multiple platforms • Can control a real robot • Can control a simulated robot • ROS compatible	• Uses popular BDI reasoner (ex: Jason) • Behaviour framework • Behaviour prioritization • Obstacle avoidance • Resource management • Navigation

Table 1.1 provides a summary of the requirements for the Agent in a Box that have been discussed in this section. This table is organized in two categories: how the agent connects to the environment and the behaviour of the agent itself. Considering the connection to the environment, there needs to be flexibility for a variety of different sensors and actuators that may be used by a variety of different robots, both real and simulated.

1.2 Contributions and Publications

This research has made several contributions and has resulted in a number of peer-reviewed publications. The first, published at the International Workshop on Engineering Multi-Agent Systems (EMAS) 2019, was a joint project on modelling and simulation and Multi Agent System (MAS) using Jason BDI agents. It provided the Simulated Autonomous Vehicle Infrastructure

(SAVI), an open source architecture for integrating BDI agents with a simulation platform. This allows for separation of concerns between the development of complex multi-agent behaviours and simulated environments in which to to test them [7]. Building on the work with SAVI, an integration of this concept with ROS was developed and integrated with an iRobot Create2 robot as a prototype for mail delivery in the Carleton University tunnels. This work was presented to Agents and Robots for reliable Engineered Autonomy (AREA) 2020 [18] and then extended in a special issue journal [19]. For EMAS 2021, the work focused on the navigation aspects of mobile robots using BDI [8]. It provided a generic navigation plan which generates a route to the robot's destination in the form of a plan that can be monitored, suspended, and resumed by the reasoner as needed. A journal paper introducing the Agent in a Box framework, which included the navigation sub-framework, the customized selection of plans, and a profile assessment, was published in a special issue journal on "Intelligent Control of Mobile Robotics" [20]. The most recent publication, accepted at EMAS 2022, provided a performance comparison against different agent designs, including the Agent in a Box and several alternatives, to measure the performance cost and benefits of using the Agent in a Box [21]. These alternative agents were also compared in terms of their software engineering properties, specifically coupling, cohesion, and cyclomatic complexity. This assessment provided insight on the maintainability of the different agents as well as how difficult it would be for a developer to work with the Agent in a Box.

1.3 Organization of this Thesis

The rest of this thesis is organized as follows. Chapter 2 provides a background on several relevant topics including the concept of software frameworks, other relevant agent architectures which provide useful properties, BDI agents and the AgentSpeak language used for programming them, and a popular framework for developing robotic systems – ROS. This is followed by the state of the art, in chapter 3, which presents several projects that have worked in the area of BDI for robotic agent systems. Areas of commonality between these projects as well as limitations in the current state of the art in this area are discussed. Chapter 4 presents the methodology for validating the Agent in a Box, followed by chapter 5, which details its design by building on recommendations and identified limitations from the state of the art. Chapter 6 details how the Agent in a Box was used to control robots in a variety of application domains. These include grid-based environments, a simulated autonomous car, and a prototype mail delivery robot. This chapter also details how the Agent in a Box was used in each of these domains, including how the agents were connected to their specific robots and how the behaviour was customized, so that the agent could complete its mission. Chapter 7 discusses the validation of the Agent in a

Box. Results of the case studies used for validating the Agent in a Box are provided. The results show that the use of BDI was successful for controlling the mobile robots in all of the tested application domains. Finally, chapter 8 includes a summary of the key accomplishments of this work, its limitations, and a recommendation with respect to the use of BDI and the Agent in a Box.

1.4 Summary

This thesis explores the use of BDI to control autonomous mobile robots. To do this, the Agent in a Box was developed with the goal of easing the burden of implementing robotic agents for various application domains. This has been demonstrated in several application domains, such as a grid-based agent, a simulated autonomous car, and a prototype mail delivery robot. This thesis provides the design methodology and implementation details for both the Agent in a Box and the specific application domain agents mentioned above. The benefits and limitations of this approach will be identified, focusing on how the agent can be connected to various robots to sense their sensors and control their actuators in a timely way. As well, the behaviour of the agent will be considered, specifically how the agent's behaviour is defined and generalized for use by different applications, how it can be customized for specific applications and how the behaviour is selected by the agent.

Chapter 2

Background

Background information on the key tools and methods that have been used for creating the Agent in a Box will be provided in this chapter. As the Agent in a Box provides a framework for the development of autonomous mobile robots with BDI, it is important to understand the definition of software frameworks. This is provided in section 2.1. Examples of other architectures used for agent implementations, including the Subsumption Architecture, TouringMachines, and InteRRaP, are discussed in section 2.2. The Subsumption Architecture's method of behaviour prioritization provided the inspiration for the agent behaviour prioritization used by the Agent in a Box. The TouringMachines and InteRRaP architectures emphasised both the need for, and the difference between, goal-directed behaviour and reactive behaviour, as well as the relative prioritization between these behaviours. Understanding the difference between goal-directed behaviour and reactive behaviour and making effective use of them is important for the Agent in a Box, as a mobile robot is expected to react to changes, such as previously unknown obstacles that need to be avoided, while working toward its goals. With these concepts defined, the focus can then move to understanding BDI agents, in section 2.3, including a high-level overview of the BDI paradigm and the Jason reasoner. Section 2.4 provides an explanation of the AgentSpeak language, which is used for defining a Jason agent's behaviour. The explanation includes details on how the AgentSpeak language can be used for defining an agent's behaviour using both goal-directed and reactive behaviour. This is illustrated using an example environment. An introduction to ROS, a popular system for implementing distributed robots which was used for connecting the agent's reasoner to the robot's sensors and actuators, is provided in section 2.5. All of this background information is summarized in section 2.6.

2.1 Software Frameworks

The Agent in a Box provides a software framework for autonomous mobile robots using BDI. It is therefore important to understand what a software framework is. A software framework can be defined as being a "reusable design of a program or part of a program" [22]. It is a mixture of both concrete and abstract software, promising their users "higher productivity and shorter time-to-market through design and code reuse" [23]. The defining characteristic of a framework is the concept of "inversion of control" [24]. Typically in software development, without the use of a framework, the program's thread of control is specified by the developer, with any functions provided by external libraries. By contrast, frameworks serve as "extensible skeletons" [24]. To use a framework, a developer must provide functions to the framework to call. By providing functions, the framework's generic algorithms are customized to specific domains. With the framework being in control of when these functions are called, the control of the application has been inverted. 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 framework imposes a structure on the developer. By following the framework's required structure, the developer can take advantage of the framework's features. The use of the framework should eliminate any duplication of effort between different applications, as the framework handles the generic aspects of the application.

In object-oriented software development, a framework will typically provide a set of classes with default behaviour to be overridden as needed or abstract classes to be completed by a developer. The developer's job is to provide an application-specific implementation of any methods required, enabling the framework to use these methods when needed. In languages that are not object-oriented, this can be accomplished through the use of specified function interface definitions. Ultimately, the framework needs a way to call the developer-provided software. As will be shown later in this chapter, BDI agents often use a language called AgentSpeak. Therefore, a method is needed to provide generic algorithms in AgentSpeak as well as a way for providing custom AgentSpeak code so that the framework can be applied to specific domains. This issue is discussed in the approach chapter, where the details of the Agent in a Box's behaviour framework are provided.

The developers of JUnit, a popular testing framework for Java, highlighted several key design features that help make frameworks better in practice [25]. For example, a framework should use tools that are familiar to developers, thus making use of the framework is a natural extension of the way the developers are already working. As well, a framework should allow for components provided by different developers to be combined without concern for interference between the components. For the Agent in a Box, this motivated the choice of using both a popular BDI reasoner, Jason, and a popular framework for connecting the agent the robot, ROS. Jason and

ROS are both discussed later in this chapter.

2.2 Other Agent Architectures

In addition to the BDI approach discussed in section 2.3, there are alternative agent architectures. These offer different means for an agent to select which action is appropriate to take for a given circumstance. The Subsumption Architecture, which inspired the method used with the Agent in a Box for prioritizing behaviour, is discussed in section 2.2.1. This is followed by two other agent architectures: TouringMachines and InteRRaP, which separate the concerns of goal oriented behaviour and reactive behaviour in their architectures. This separation of concerns provided a useful design principle for defining the behaviour of the Agent in a Box. These architectures are discussed in section 2.2.2 and section 2.2.3 respectively.

2.2.1 Subsumption Architecture

Brooks proposed the reactive Subsumption Architecture [1,2], claiming that it was not necessary to have the explicit representations or abstract reasoning required by other types of symbolic Artificial Intelligence (AI). Instead, the intelligence of the agent can be an emergent property of the agent's behaviour. The Subsumption Architecture has two defining characteristics. The first of these characteristics is that the agent is made up of a set of task-accomplishing behaviours implemented as situation-action pairs without the use of symbolic logic. The second characteristic is that multiple task-accomplishing behaviours can fire simultaneously. These behaviours are organized in a layered hierarchy, where lower layers have higher priority and the ability to inhibit higher layers if needed.

Figure 2.1 shows how action selection functions in the Subsumption Architecture. Here, 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 can *subsume* the actions provided by the higher level layers and take 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 provided the inspiration for the behaviour prioritization method used by the Agent in a Box.

2.2.2 TouringMachines

TouringMachines, shown in figure 2.2, are a type of hybrid agent architecture which uses a combination of horizontal and vertical layering. Sensor inputs are processed by a perceptual subsystem and then provided to three activity-producing layers: a reactive layer, a planning

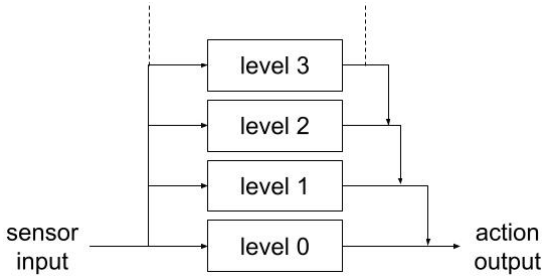


Figure 2.1: Action Selection in the Subsumption Architecture [1].

layer, and a modelling layer. These are implemented as part of a control subsystem which coordinates which layers have access to the inputs and which layers provide actions for the agent to follow. The modelling layer is used for modelling various aspects of the environment, the agent itself and other agents. The modelling layer then predicts conflicts and generates goals for the agent to achieve. In the planning layer, the agent uses a plan library in an attempt to achieve goals. The reactive layer provides reactive, stateless behaviour in the form of "situation-action rules" for the agent to follow. [2]

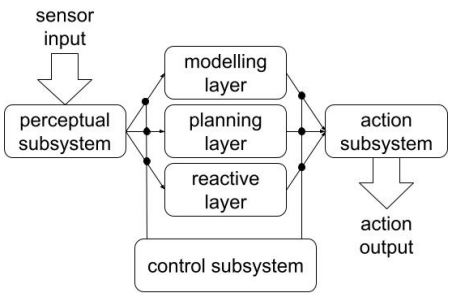


Figure 2.2: Action Selection in the TouringMachines Architecture [2].

2.2.3 InteRRaP

Another example of a hybrid agent architecture is InteRRaP, shown in figure 2.3. Similar to TouringMachines, this architecture has several layers, each capable of receiving sensor input, maintaining a knowledge base, and generating actions. The lowest layers have the rawest representation of data and the highest levels have more processed symbolic representations. The higher level layers act by utilizing the tools provided by the lower levels. The lowest layers are also more *reactive* than the higher level layers. [2]

2.2.4 Summary and Discussion of Other Agent Architectures

This section provided a brief overview of three other agent architectures. All three of these architectures provide useful features that address challenges that the Agent in a Box needs to

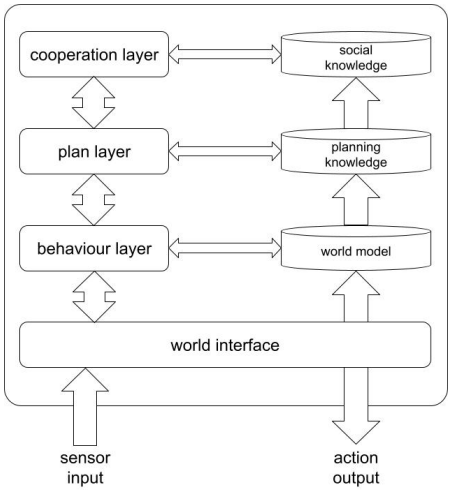


Figure 2.3: Action Selection in the InteRRaP Architecture [2].

overcome, namely relative priorities of different behaviour and the need to mesh interruptions with mission behaviour. Indeed, the first architecture discussed was the Subsumption Architecture, which provided the inspiration for the Agent in a Box's method for prioritizing behaviour in the reasoning cycle. Further discussed in the approach chapter, this prioritization ensured that the reasoner selects behaviour that addresses its highest priority. The second architecture discussed was the TouringMachines architecture, which uses modeling, planning, and reactive layers for controlling the agent. The separation of reactive behaviour from the goal-based behaviour provides an approach for addressing mission related goals and emergent issues, such as obstacle avoidance. The third architecture discussed, InteRRap also has this feature, using a combination of horizontal and vertical layering, including cooperation, plan, and behaviour layers as well as knowledge associated with each of these layers. The ability to provide reactive behaviour to a goal-oriented agent is an attractive feature of BDI that will be used in the design of the plans for the Agent in a Box behaviour framework, discussed in the approach chapter.

2.3 Beliefs-Desires-Intentions

The Beliefs-Desires-Intentions (BDI) approach, originally called the Desire-Belief model, was first introduced by Bratman as a means of modelling the cognitive processes which give rise to agency [10]. Shoham provided a logical foundation for rational reasoning using Agent Oriented Programming (AOP) [26]. Shoham defined the agent as having a mental state consisting of beliefs, decisions, capabilities, and obligations. A complete system using the Agent Oriented Programming (AOP) paradigm includes a restricted formal language with syntax and semantics for describing the agent's mental state using modalities including belief and commitment. It should also have an interpreted programming language in which to define and program agents,

and an "Agentifier" for converting neutral devices into programmable agents. Building from AOP, it is understood that a rational agent should only commit to the goals it believes that it can achieve. It should also give up a goal if it has become impossible to achieve, or if the motivation for the goal no longer exists [3]. Rao and Georgeff [11] built on foundation provided by both Bratman and Shoham by developing BDI as a type of symbolic AI. In BDI systems, a software agent performs reasoning based upon internally held beliefs, stored in a belief base, about itself and its 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 on these foundations, BDI agents can enforce 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 its task environment and receiving messages. From this information, the agent can then decide on a course of action, in the form of a plan, suitable to the context provided by those perceptions, the agent's own beliefs, messages received, and desires. Once a course of action has been selected, the agent sets it as an intention for itself. These plans can include updating the belief base, sending messages to other agents, and taking an 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 [3, 13]. There have been several implementations of BDI, the most popular being Jason [3, 4, 13]. Other implementations include JACK [27] and LightJason [28, 29]. Gwendolen is another BDI framework which emphasises the use of model checking for verifiable agents [30]. Jason has also been included in the JaCaMo framework, which includes Jason for AOP in BDI, Cartago for programming environment artifacts, and Moise for setting up multi-agent organisations [31]. This work uses the popular Jason implementation of BDI, which was built off the formal foundations of AOP and BDI, and provides rational decision making to its agents. Jason also provides its reasoning cycle as a framework, which provides default functions which can be overridden as needed.

As mentioned in the previous paragraph, the reasoning cycle provides the means through which the agent makes decisions; from perceiving its environment, deliberating, and selecting actions to perform. This is important, as one of the requirements of the Agent in a Box is that the agent must appropriately select which behaviour should be run from several possible options. The Jason reasoning cycle is shown in figure 2.4 [3, 4]. Starting in the upper left corner, the agent receives perceptions and messages, which leads to an update of its internal *belief base* and also triggers an *event*. As only a single event can be handled per reasoning cycle, an *event selection* function selects the applicable event. The parameters of that event are unified with the plans in the *plan library*, which contains the plans that are available for the agent to execute.

The contexts are verified and applicable plans are selected and added to a queue of *intentions*, which is a queue of plans that have been selected to run. An *intention selection function* is used to select the plan, which is then executed. A plan can involve performing actions, sending messages, updating beliefs, or updating goals. [3]

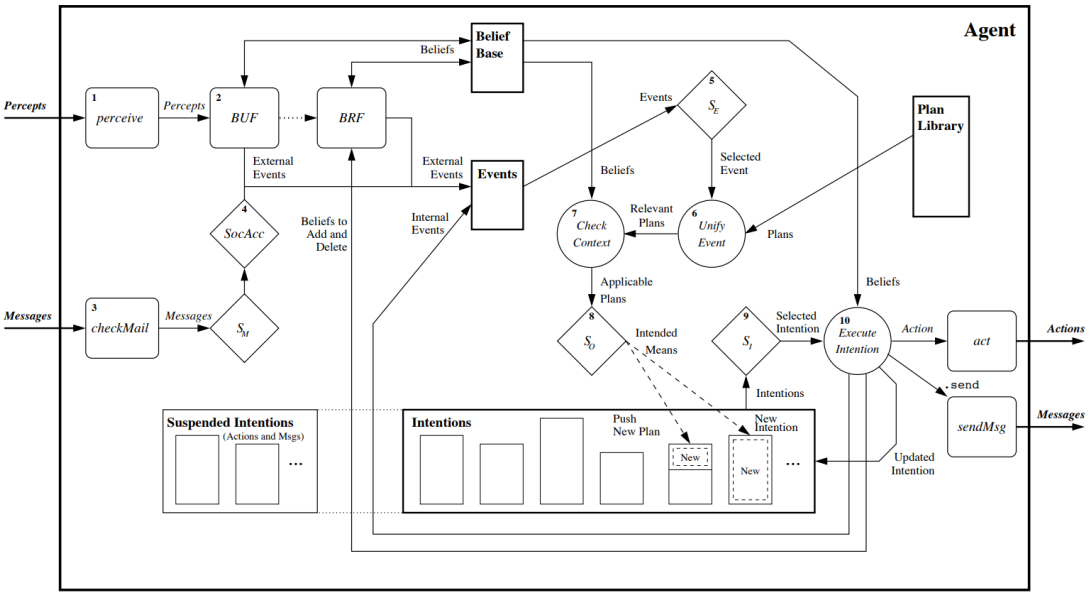


Figure 2.4: BDI Reasoning Cycle in Jason [3, 4].

2.4 Overview of the AgentSpeak Language

Agents developed for BDI systems using Jason are programmed using a language called AgentSpeak [3, 14]. AgentSpeak is a Prolog-like language that has been described as *post-declarative*, as it is designed to be interpreted as part of an agent's reasoning cycle [3]. 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 [3]. Below, a brief overview of AgentSpeak is provided starting with the basics of the AgentSpeak syntax for specifying plans and rules in section 2.4.1. With the basics established, the focus will then shift to discussing the use of AgentSpeak to design and specify agent behaviours. In section 2.4.2, examples of how AgentSpeak can be used to implement both goal-directed agent behaviour and reactive agent behaviour are discussed. Those examples are provided in the form of a sample implementation of an agent for a simple linear environment. These examples demonstrate how a BDI agent can provide both of these types of behaviours, as was also the case with both the TouringMachines and InteRRaP architectures, discussed previously in section 2.2. These types of behaviours can

be combined using sub-goals, discussed in section 2.4.3.

2.4.1 AgentSpeak Syntax

In general, AgentSpeak plans are declared using the syntax shown below in listing 2.1. Each of the terms in the listing are logical literals forming beliefs, goals, etc. The components of the plan syntax include a *trigger*, a *context*, and a *body*. The properties of these components are discussed in the following paragraphs. Examples of their use are provided in the next section. [3]

Listing 2.1: AgentSpeak Plan Syntax.

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

A *trigger* is the addition or deletion of a belief, an achievement goal, or a test goal. Triggers that are based on the addition or deletion of a belief or a goal begin with a positive (+) or negative (-) sign respectively. To differentiate goals from beliefs, achievement goals begin with an exclamation mark (!) and test goals begin with a question mark (?). An achievement goal is used for providing the agent with an objective related to the state of the environment, whereas a test goal is generally used for querying the state of the environment. [3]

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 a logical sentence that can use both beliefs as well as separately defined rules. Rules, which are useful for reducing code duplication, are structured as logical implications as shown in listing 2.2. In this listing, the `conclusion` is implied by the `condition` [3].

Listing 2.2: AgentSpeak Rule Syntax.

```
1 conclusion :- condition.
```

The body of the plan includes the instructions for the agent to follow to execute the plan. The plan's 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 is an atomic plan, which must be run to completion, it cannot be interrupted. Because the AgentSpeak language follows the Agent Oriented Programming (AOP) paradigm, agents written in this language will select plans

that move them toward their goals. This will work as long as the symbolic representation of the environment is correct, the plan's context is correct, and the plan's body is correct. [3]

With the syntax of AgentSpeak now understood, the next section provides examples of how this language can be used to define goal-directed and reactive behaviours for BDI agents.

2.4.2 Example AgentSpeak Programs

In understanding the AgentSpeak syntax for plans and rules, the method for how an AgentSpeak program can be written has been provided. Using these components, AgentSpeak can be used to provide an agent with a combination of goal-directed and reactive behaviour. The need for agents to have both types of behaviour was highlighted by both the TouringMachines and InteRRaP architectures, discussed in section 2.2.

Consider an agent in a simple linear environment shown in figure 2.5. The agent is situated in a one-dimensional path where it can move between adjacent locations. The agent perceives its current position with the `position(Current)` predicate, which provides the agent's location index. The agent has a goal of moving from its start position at `position(1)` and moving to `position(4)`. A goal-directed implementation of the desired agent behaviour is provided as an example in section 2.4.2.1. This is followed by an example using reactive behaviour in section 2.4.2.2.

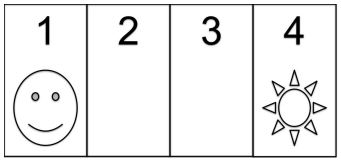


Figure 2.5: Linear Agent Environment.

2.4.2.1 Example Goal-Directed Behaviour Implemented in AgentSpeak

To understand the use of AgentSpeak, it is beneficial to consider how a goal-directed behaviour can be written. This program is provided in listing 2.3. A simple achievement goal of `!goTo(4)` specifies that the agent needs to move to the fourth location in the environment. Next are two plans for `!goTo(Location)`. The first plan is applicable when the `Location` variable in the achievement goal and the context unify, meaning that the agent has arrived at the location. The body of this plan is the execution of the `drive(stop)` action. The second plan is a default plan, in that there is no context specified. Using Jason's intention selection method of selecting the first applicable plan for execution, the second plan will only be selected when the first plan was not applicable, meaning that the agent had not yet arrived. The plan body in this case has the

Listing 2.3: Example Goal-Directed AgentSpeak Program.

```
1 !goTo(4).
2 +!goTo(Location)
3     :   position(Location)
4     <-  drive(stop).
5 +!goTo(Location)
6     <-  drive(forward);
7         !goTo(Location).
```

Listing 2.4: Example Reactive AgentSpeak Program.

```
1 +position(4)
2     <-  drive(stop).
3 +position(Location)
4     :   Location < 4
5     <-  drive(forward).
```

`drive(forward)` action, moving the agent forward, followed by `!goTo(Location)` so that the agent readopts the achievement goal of moving to the destination. This use of recursion is a common approach for maintaining goals in AgentSpeak.

2.4.2.2 Example Reactive Behaviour Implemented in AgentSpeak

An alternative method for providing the agent with the desired behaviour is to use a reactive behaviour, without a specific goal being provided. This can be achieved using belief-triggered plans, as shown in listing 2.4. In the program, there are two plans which are triggered by the agent's knowledge of its position. The first plan triggers on the specific knowledge that the agent is located at position four, the goal position, therefore the agent should stop moving. The second plan is applicable when the agent's position is smaller than four, therefore the agent needs to drive forward to reach the destination.

Although both the goal-directed and reactive programs resulted in very similar results, with the agent moving to location four and stopping, the method of achieving this was different. The goal-directed agent worked on the achievement of a goal, whereas the reactive agent had no goals, it simply reacted to perceptions and executed the available plans that were triggered by the associated beliefs. There was an advantage to implementing this behaviour as goal-directed behaviour, as the agent could more naturally drop this goal, or adopt a different goal if one was provided. This was not the case with the reactive agent, where the goal was embedded in the program in the triggers and contexts of several plans. These plans would keep triggering every time the agent receives a position update, even if it had other goals, or even had arrived at the destination.

In the example scenario provided there was a clear advantage to implementing this agent's behaviour using achievement goals, however this may not be the case for every desired behaviour. In fact, as was discussed with the TouringMachines and InteRRaP architectures, there may be a

need for an agent to have both goal-directed and reactive behaviour. AgentSpeak provides BDI agents with a means of implementing both types of behaviour.

2.4.3 Sub-Goals in AgentSpeak

Building on the basic goal-directed behaviour that was discussed in section 2.4.2.1, AgentSpeak provides a mechanism for specifying sub-goals: Goals which are adopted as part of executing a plan. This is shown in listing 2.5. In this listing, the agent needs to achieve the goals `!subGoal1(A)` and `!subGoal2(B)` in order to achieve `!goal(A,B)`.

Listing 2.5: Example of an AgentSpeak SubGoal.

```
1 +!goal(A,B)
2     <-  !subGoal1(A);
3         !subGoal2(B).
```

Sub-goals can also be used to combine reactive behaviour with goal-directed behaviour. Consider listing 2.6. In this case, the agent adopts `!goal(A)` whenever the agent gains the belief of `observation(A)`.

Listing 2.6: Combining Reactive and Goal-Directed Behaviour.

```
1 +observation(A) <- !goal(A).
```

The AgentSpeak language provides the syntax and support necessary to implement both goal-directed and reactive behaviours using goal-triggered and belief-triggered plans. It also provides the opportunity to combine reactive and goal-directed behaviour, using a sub-goal, by having a belief-triggered plan adopt an achievement goal. The Agent in a Box uses all of these constructs as part of the behaviour framework, presented in the approach chapter.

2.4.4 AgentSpeak Summary

The AgentSpeak language is used to provide beliefs, goals, plans, and rules to Jason. After gaining an understanding of AgentSpeak syntax, including the components of plans and rules, the focus shifted to understanding the design of agent behaviour using this language. AgentSpeak plans are triggered by an event, which can include a change in goals or beliefs. This allows agents implemented with AgentSpeak the ability to perform goal-directed and reactive behaviours, as was seen with TouringMachines and InteRRaP agents. Goal-directed and reactive behaviours were demonstrated using an example agent in a simple linear environment. The agent was implemented using achievement goal-triggered recursive plans, and using reactive behaviour using belief-triggered plans. AgentSpeak can be used to combine the features of reactive behaviour

with goal-directed plans, by including the adoption of a goal in the body of a belief-triggered plan. All of these features are used by the Agent in a Box behaviour framework.

Now that BDI agents have been discussed, it is necessary to consider how an agent can be connected to a robot's sensors and actuators. The Agent in a Box uses ROS for providing this connection. ROS is discussed in the next section.

2.5 Robot Operating System (ROS)

Robot Operating System (ROS) is a package for developing distributed software for robotic applications [32]. For the Agent in a Box, ROS was used for connecting the BDI agent to the environment, which will be discussed in more detail in the approach chapter. ROS operates using a tuple-space architecture where different aspects of the software are written in different software nodes. These software nodes publish and subscribe to various *topics* using socket-based communications rather than communicating with other nodes directly. By publishing and subscribing to these topics, the developers of individual nodes do not necessarily need to concern themselves with which nodes they are interacting with. The connections between the nodes are 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 and the Point Cloud Library [33, 34] for 3D data from sensors such as a laser imaging, detection, and ranging (LIDAR) sensor. There is also integration with the MoveIt! library [35, 36] for planning algorithms as well as tools for industrial robotics with ROS-Industrial. Also the 'move_base' node is a useful part of the ROS navigation stack, useful for providing Simultaneous Localization and Mapping (SLAM) based navigation support [5, 37, 38]. By building robotic applications that are compatible with ROS, developers enable their applications to be used by other devices and software nodes supported by the ROS community. Therefore, developers are able to focus on the implementation of individual nodes and at the same time are able to connect and use other available ROS nodes without concern for how those nodes are implemented. These features help address the need for the Agent in a Box to be flexible for use with a variety of different sensors and actuators found on different robots.

2.5.1 ROS Navigation Stack

Among the variety of nodes available for ROS is the navigation stack. The navigation stack is available for terrestrial robots [5, 37, 38]. Specifically, it provides mobile robots with guidance

for moving to a destination while avoiding obstacles. The main components of the navigation stack are shown in figure 2.6. The architecture and function of the navigation stack provides valuable insight into how a popular and useful navigation tool for mobile robotics works, and what is expected of it.

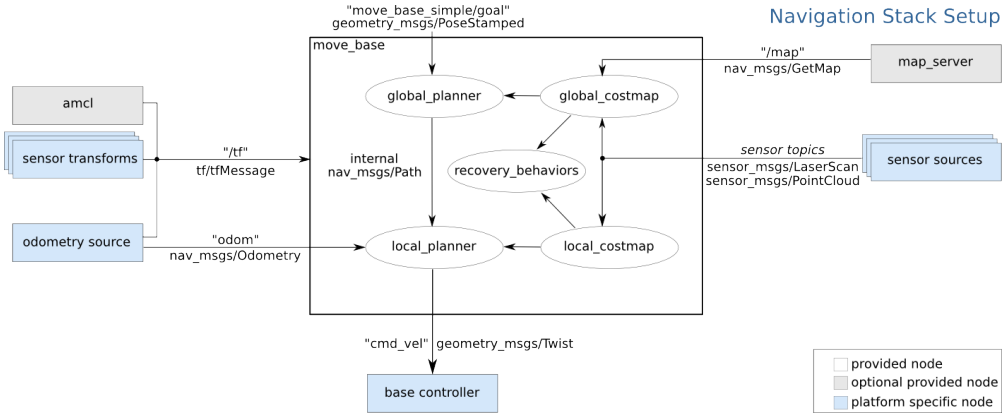


Figure 2.6: ROS Navigation Stack Architecture [5].

The navigation stack senses the environment using point cloud data, generated by a LIDAR sensor, and the robot's odometer. Using point cloud data, a three-dimensional Voxel Grid representing the obstacles that the robot needs to avoid in the environment is generated. The Voxel Grid is then projected onto a two-dimensional cost map which provides the cost for the robot's movement. A global planner, implemented using A*, generates a route to the destination through the Costmap. The global planner does not include any handling of the dynamics of the robot or obstacles, which are both the responsibility of the local planner, which controls the speed and steering of the robot. In the event that the planners are unable to continue, perhaps because the global planner has generated a route which the robot cannot complete, a new plan is generated through the recovery behaviours.

Although the ROS navigation stack does provide a useful solution for mobile robot navigation, it does have several published requirements that need to be satisfied to be useful [5]. It requires the use of a planar laser sensor, such as a LIDAR for generating the point cloud used for generating the Voxel Grid and cost map that is used by the planners to control the robot's movement. Furthermore, the navigation stack was designed for use with robots that move using either differential drive or holonomic wheels, ideally robots that have a circular or square base, although there is some precedent for the navigation stack having been used to control robots with rack-and-pinion steering [39].

By developing an agent for mobile robots that works with ROS, the Agent in a Box has the opportunity to use complementary modules like the navigation stack. This module can be used to provide the lower level aspects of navigation, such as the management of point cloud sensor

data, and the control for maneuvering the robot while the Agent in a Box's BDI agent can focus on higher level issues.

2.6 Background Summary

Key elements that were used for this research and the development of the Agent in a Box were discussed in this chapter. Specifically, software frameworks, other agent architectures, details of how BDI agents work and how their behaviour is defined in AgentSpeak, and ROS.

Software frameworks provide generic functionality that can be customized for use in a specific application. A software library provides functionality that can be called by a custom application. A key difference between a software framework and a software library is the concept of *Inversion of Control*, where the framework is primarily in control of the software execution, calling the customized methods as needed. A developer using a framework can specify what these functions do, but not when they are called. In the case of the Agent in a Box, it was necessary to design it as a framework for several reasons. One of the reasons being that for the Agent in a Box to control a mobile robot's movement it needs to be provided custom plans for it to use for moving that specific mobile robot. The design decisions for the Agent in a Box are discussed in chapter 5, the approach.

Among the desired features of the Agent in a Box are the need for the agent to appropriately select plans to run based on their relative priority as well as balance the need for performing goal-oriented behaviour for achieving its mission and reactive behaviour, such as for obstacle avoidance. For inspiration for these features, several other relevant agent architectures were considered. The Subsumption Architecture provides a simple but effective way to define agent behaviour using perception-action pairs in a relative priority layering structure. This relative prioritization inspired the behaviour selection method that was used for the Agent in a Box. The TouringMachines and InteRRaP architectures both identify the need to have a separation of concerns between goal-oriented behaviour and reactive behaviour, a concept that is needed for the Agent in a Box.

Central to the Agent in a Box is the use of a BDI agent for rational decision making. BDI agents can commit to goals that they believe that they can achieve and dropping them if they believe that the goal is no longer possible. Jason is one of the most popular BDI reasoners available, providing an AgentSpeak interpreter and BDI reasoner. Jason implements the semantics of the AgentSpeak language, used for providing the agent with a set of initial beliefs, rules, and plans which define its behaviour. Jason's reasoning cycle provides a set of default functions used for selecting which plans get executed by the agent. These default functions can be customized,

allowing developers to override how the agent selects which plan to run. The Agent in a Box provides customization for this purpose. It also provides plans and rules that are useful for a variety of mobile robot applications.

ROS, a popular framework for distributed robotics, was also used by the Agent in a Box. As ROS has a vibrant community of developers and projects, using it makes it easier to leverage these third-party nodes. ROS uses a publish and subscribe architecture, where various nodes communicate with relevant topics rather than to other nodes directly. This architecture removes the need for the developers of any specific node to concern themselves with what other nodes are part of the system and focus on the specific issues concerning their specific node. Thanks to these features and popularity among robotics developers, ROS was selected as the means by which the Agent in a Box was connected to the environment, using the robot's sensors and actuators.

Now that the key methods and tools used by this thesis have been established, attention can shift to the state of the art, looking for examples of BDI being used for controlling autonomous mobile robots in relation to the research question of this thesis.

Chapter 3

State of the Art

This chapter discusses the state of the art in the application of BDI for autonomous mobile robots. BDI, or Agent Oriented Programming (AOP) in general, is most commonly used for simulated problems, often using grid based environments, as can be seen in several iterations of the Multi-Agent Programming Contest [40]. Although grid based environments produce valuable insight into the properties of software agents and MAS, the interest here is with applications where the focus was 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. Therefore, the state of the art focuses on projects that are closely related to the goal of this thesis: to investigate the use of BDI agents with mobile robots.

First, to establish the context of what work has been done in this area, projects which have used BDI agents for controlling real or simulated mobile robots are introduced in section 3.1. Specifically, the challenges that the projects address, the questions that the projects answer, and the characteristics of the agents that the projects have implemented will be discussed. In addition, what BDI implementation was used, the target platform, and the architecture are considered. Whether the agent worked on one or several different simulated or real robots is identified. Since this thesis focuses on the use of the Agent in a Box, each of these projects is assessed against the notional requirements for the Agent in a Box that were presented in the introduction.

With the relevant projects introduced, these projects have been assessed against the requirements for the Agent in a Box that have been outlined in the introduction chapter in section 1.1. This includes how the agents were connected to the environment, the flexibility for different mobile robots, the performance of the system, and the behaviour of the agents themselves, including the activities that these agents demonstrated and any generalized framework that was used. The evaluation of these projects is not an assessment of the project or the work that was done. All

of these projects have made useful contributions in their own right. This evaluation assesses the work in terms of suitability for the purpose of achieving the goals of the Agent in a Box, which was not necessarily the goal of these projects. This evaluation is provided in section 3.2. Finally, a summary of the state of the art, along with the strengths and gaps, is provided in section 3.3.

3.1 Overview of Related Work

There have been a number of projects that have, for various reasons, investigated the use of BDI for controlling autonomous mobile robots. The first of these examples is a set of projects developed by the Australian military's research agency into BDI for controlling a Unmanned Aerial Vehicle (UAV). These projects are examined in section 3.1.1. Next, more recent works which integrate BDI with the ROS, the JaCaMo UAV and JaCaROS are discussed in sections 3.1.2 and 3.1.3 respectfully. A Python based implementation, called Python RObotic Framework for dEsigning sTrAtegies (PROFETA), for implementing robotics with BDI is discussed in section 3.1.4. The ARGO project is discussed in section 3.1.5. The concept of "Abstraction Engines" is discussed in section 3.1.6. In addition to the projects that are evaluated, a variety of other uses of BDI in robotic applications are discussed in section 3.1.7. These include a variety of smaller projects, or projects where less detail was available, making a full assessment less practical.

3.1.1 Codarra Avatar

The Australian defence research activities into autonomous systems investigated the use of BDI using JACK [27]. Their focus was to provide a simple way for pilots or other domain experts who are not programmers to set missions for UAVs. Wallis et. al. developed the "Automated Wingman", a graphical programming environment where pilots could provide mission specific programming for a UAV in a way that was more familiar to them [41]. The authors' goal was to reduce the requirement for a BDI specialist and an application domain specialist to work together to develop behaviours.

Continuing on this theme, Karim and Heinze trialed the use of BDI, again using JACK, for controlling a UAV [42]. Their implementation focused on implementing Boyd's Observe Orient Decide Act (OODA) Loop [43], a military decision-making model, in BDI with a goal of developing an agent that would solve problems similarly to how pilots perform the tasks. Karim and Heinze used JACK to implement a software agent to fly a Codarra Avatar UAV, a type of fixed wing aircraft. The BDI agent was loaded on an HP iPaq connected using a serial link to an autopilot that controlled the aircraft's flight controls. They successfully performed flight tests of a navigation focused mission with their UAV in July of 2004. The mission management was

performed using BDI, however the lower level trajectory and attitude management was handled by the autopilot mentioned earlier. The architecture was designed to use a "single, linear thread of control", meaning that the agent checked for sensor data at the beginning of each reasoning cycle. With respect to the OODA loop, the agent was implemented to first *observe*. This was done by the agent updating its "situational awareness" through perception of itself and the environment, for example detecting its position. Next, the agent was to *orient* by doing a "situational assessment" where the agent would, among other activities, check its proximity to a waypoint. This is followed by *decide*, or "tactics selection", where the agent could select waypoints. The *act* step of the loop, also referred to as "tactics implementation" in the paper, is where the agent commands its autopilot to fly to a destination. The authors pointed out that there was additional complexity of building an agent to fly the UAV that may be viewed as an added burden of development. However, they also saw an advantage to making the development of drone behaviours more "intuitively accessible" to domain experts, such as pilots, who already naturally think in terms of OODA. [42]

3.1.2 JaCaMo UAV

A more recent example of BDI being used for controlling a drone was provided by Menegol [6]. The experiments used for their project are available as open source software [44]. Their implementation used the JaCaMo framework [31], which includes Jason. A video of their UAV flying is available online [45]. For their implementation, the agent was installed on a BeagleBone Black single board computer. This was connected to a pixhawk flight controller using MAVLINK over a serial bus. Although the implementation was developed to fly on a real SK450 quadcopter UAV, they performed their experiments using the ArduPilot Software in the Loop (SITL) simulation environment. The sensors available to the agent provided the agent with the position of the UAV. The actions available to the agent allowed it to command a flight controller to fly the UAV to particular waypoint locations.

Menegol's architecture uses the *head* and *body* approach shown in figure 3.1 [6]. The head is a JaCaMo agent which receives perceptions and messages, and provides actions and messages to and from its body, which is a hardware interface implemented in Python. The mission management activities are done with BDI, and trajectory and attitude management are handled by an autopilot controller. The autopilot controller provides all sensor inputs to the agent as well as providing all available actions to the agent – primarily locations that the UAV should visit. MAVLINK is used for communication between modules. Coordination between multiple agents is implemented using Moise, which is part of JaCaMo. In multiagent tests, a manager agent was used to help coordinate the activities of individual agents. In their paper they demonstrated a

search and rescue mission. As the UAV was not implemented with sensors that were able to perceive the locations of the persons in distress, these locations were hard-coded into the agent's belief base [6].

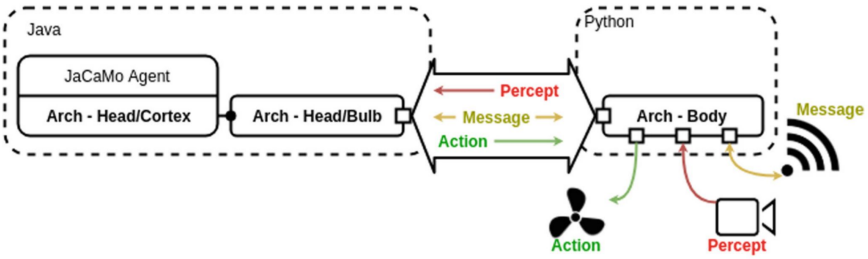


Figure 3.1: Implementation of the JaCaMo UAV [6].

Menegol's publication claims that their work was being extended to use ROS as the core of the architecture [46, 47]. Their approach was to build a linkage between ROS and Jason, where Jason agents can run actions by passing messages to modules in ROS and receive perceptions by receiving messages from other modules. The perceptions and actions were defined using manifest files that specify the properties and parameters of the messages. This is similar to other efforts to link ROS to Jason, such as Rason [48], JaCaROS [49], and JROS [50], although it is unclear if these efforts are related to the JaCaMo UAV project. [6]

3.1.3 JaCaROS

Related to the JaCaMo UAV discussed in section 3.1.2, Wesz's Master's thesis sought to investigate the advantages of using Agent Oriented Programming (AOP) using Jason with JaCaMo for controlling agents that were setup in a ROS based architecture [51]. They compared the source code of Agent Oriented Programming (AOP) implementations with similar behaviours implemented in Python – comparing the number of identifiers as well as the number of lines of code and the size of the source code files. They found that the Jason agents generally had fewer lines of code than the Python equivalents, interpreting this to mean that the Jason code was less complex than the Python equivalent. Unfortunately, as this conclusion was drawn from the number of lines of code, which is not necessarily an indicator of the level of coupling or cohesion in the software, this is not a convincing assessment.

The comparison was done for five separate simulated TurtleBot scenarios using the Gazebo simulator's tiled floor environment. These scenarios were:

1. Forward and back experiment – the robot moves forward for a set time and then returns

2. Navigating a square – the robot must navigate in a square by moving between the corners

3. Fake battery – the robot was sent fake battery state of charge information. Based on this data, the robot was required to log a status of "full battery", "medium battery", or "low battery" based on this data.

4. Patrol – An aggregate of the square navigation and battery scenarios; the robot was to navigate the perimeter of the square until a low battery, then return to a charging location to recharge before continuing the patrol.

5. House cleaning – The robot was required to perform a mock cleaning action in various rooms in a grid environment. The rooms were modelled as corners of squares on the grid.

3.1.4 PROFETA

The Python RObotic Framework for dEsigning sTrAtegies (PROFETA) library is a Python implementation of BDI and the AgentSpeak language designed for use with autonomous robots [52]. The developers of this library were interested in determining if Agent Oriented Programming (AOP) can be implemented with Python for simpler robotic implementations, although they did not provide any metrics for assessing if this has been achieved. In their paper, the authors used the Eurobot challenge as well as a simulated warehouse logistics robot scenario as case studies. In the Eurobot challenge, the robot needed to sort objects in the environment while also working in the presence of other uncooperative robots and people, adding a safety requirement to the agent [53,54]. For the warehouse logistics case study, the robotic agent was a forklift that moved crates in a warehouse. The primary activity of the agent behaviour was path planning using Dijkstra's algorithm, which was available to the agent as an action. Otherwise, the agent used a scanner to identify crates that needed to be moved and moved them between different locations of the warehouse as needed, avoiding collisions and path planning as needed. [52]

3.1.5 ARGO

The ARGO project interfaced Jason agents with Arduino computers using a library called Javino, a Java library built for this purpose. The authors of the ARGO paper claim to not be tied to specific hardware or a specific Agent Oriented Programming (AOP) language, such as AgentSpeak. [55, 56]

The developers of ARGO developed a small robotic car with a set of sensors, including a distance sensor, a light sensor, and a temperature sensor. As expected, it also included a motor for driving the car, however it did not include steering. The tests discussed in their paper focused on the task of driving the car and stopping before hitting a wall. The authors claimed to be concerned about the computational burden of the agent reasoning cycle, citing Stabile and

Sichman [57], who found that depending on the number of perceptions that an agent needs to handle, belief update and unification accounted for up to 99% of Jason's execution time. The proposed solution was to use perception filters so that the agent was not required to update as many beliefs or unify as many variables. In ARGO the perception filters were written in Extensible Markup Language (XML) files that were available to the agent. These perception filters were built directly into the Jason reasoning cycle using an overridden Jason class. The Jason perception process was also tied directly to the implementation of the sensors, which were perceived at a predefined time interval. All actions available to the robot were implemented as internal actions in Jason. This included the motor for driving the robot as well as an internal action for changing which perception filter was being used, if any. The Javino library, responsible for the connections to the sensors and actuators connected to the Arduino, could use a single serial port at a time, either for listening or acting, which limits the agent to a single sensor or actuator at a time. [55]

The performance evaluation of the agents focused on the use of perception filters. The author's goal was to determine the effectiveness of the perception filters at reducing the burden of the reasoning cycle on the various sensors. The test used a collision avoidance scenario where the car was driven towards a wall and given the task of stopping once the wall was detected by the agent. It was found that without perception filters, the robot always collided with the wall and with the use of perception filters the performance improved. The authors concluded that perception filters are essential to having useful performance from embedded BDI agents. The authors proposed several activities in their future work, which included having a MAS for controlling the robot. They also discussed an interest in expanding their robots to work with other robots as part of a MAS. [55]

3.1.6 Abstraction Engines

Dennis et al. explored using rational agents, implemented with GWENDOLEN, for several robotic applications [58]. A key feature of their implementation was the use of an "Abstraction Engine", which translated between the agent and the robot's sensors and actuators. These sensors and actuators were encapsulated in a "physical engine", the robot's control software and its "continuous engine", a calculation and simulation tool for path planning. This method was used to address the challenge of using an agent reasoning system, which operates using discrete first-order logic predicates, to control a robot in the real world which can include continuous sensor signals.

Cardoso et al. interfaced BDI agents, implemented in GWENDOLEN, with ROS as described in their paper [59]. Their implementation used the rosbridge protocol [60,61] to connect

their agent reasoner with ROS. The authors also experimented with connecting Jason to ROS using the protocol approach. However, their choice of GWENDOLEN was motivated by their desire to use the Agent Java PathFinder (AJPF) model checking tool. Cardoso et. al. also highlighted two interesting issues related to linking agents with robots. First, the concern that the sensors may overwhelm the reasoner, as the sensors may generate data faster than the agent can handle it. They addressed this issue by proposing the use of filters, similar to the ARGO project in the previous section, on the sensor data to prevent the agent from being overwhelmed. Second, the concern with implementing actions using synchronous service routines in ROS could cause the agent to wait for the action to be completed before continuing the reasoning cycle. Their proposed approach for addressing this second concern was to use an external handler for executing longer term actions. This handler would provide updates to the agent, which could in turn command the handler to stop or continue the longer-term action, as necessary.

3.1.7 Other Examples of BDI Applied to Robotics

In addition to the projects discussed in the previous sections, there are other examples of work with BDI and robotic applications that can be considered. Here, a few others are discussed in less detail, either because they were less relevant to the specific focus of this thesis or because additional details were not found.

The UAVAS Platform [62] and RoboticCPS [63] projects were both found on GitHub. UAVAS is an integration of a Jason implemented BDI into a simulation engine. RoboticCPS is a graduate project where BDI agents implemented with Jason have been integrated into Lego NX. Although both projects used a fairly simplistic hardware implementation, they demonstrated multiagent behaviour in a tabletop search and rescue environment. Another project that was considered proposed using GOAL for implementing agent behaviour on a humanoid Nao robot [64].

Han, Wang, and Yi explored the control of UAVs using MAS [65]. They discussed the use of BDI as well as the use of auctions and various planning algorithms for coordinating the objectives of UAV in a military combat context. They focused on the use of artificial potential fields for both evaluating the performance of these algorithms, primarily from the perspective of path planning to various target locations. Although the focus of their work was on the implementation of an agent-based UAV, or a group of UAVs working as a MAS, the implementation of such a system was not discussed or evaluated. Dominguez, Nesmachnow, and Hernández-Vega have, as part of a broader project aimed at controlling a group of UAVs as a MAS, investigated the use of BDI with an integrated Genetic Algorithm (GA) for path planning [66].

Another popular theme in projects is the use of BDI for autonomous driving. An example of connected and autonomous vehicles was discussed by Rüb and Dunin-Kęplicz [67]. The paper

discussed an architecture for connected and autonomous cars using BDI in a simulated traffic model environment. Ehlert's Master's thesis provided a good background on agents and agent technologies [68]. It included information on the requirements and design of an agent-based autonomous driver and simulation. The results section of the thesis focused on the performance of the agent in various driving specific tests and assessed to what degree the behaviour was "human-like". For the domain of autonomous driving, Vinitsky et. al. provided several benchmark tests targeted to reinforcement learning systems[1] [69]. The benchmark tests included a figure eight intersection capacity test, merge test for controlling traffic shockwaves, traffic signal timing schedule tests, and a test for measuring traffic performance though a traffic bottleneck. A demonstration of LightJason, used for an autonomous car lab project for graduate students was presented by Aschermann et. al. [70]. In this case, the environment was a web browser-based game for the basic functions of a car.

Somewhat related to BDI is the concept of applying goal reasoning to robotic agents. Recent work by Gillespie et. al [71] and Floyd et. al. [72] used goal reasoning in military scenarios, first for a reconnaissance squad and second, for a beyond-visual-range air combat [72].

Pěchouček and Mařík provided a survey of agent technologies being deployed in industrial applications [73]. They highlight BDI as an important agent technology and also mention the Australian UAV project discussed in section 3.1.1. Also mentioned is a training tool for military leaders which used BDI for modelling the behaviour of soldiers. Lastly, Hofmann et. al. demonstrated the how a BDI agent can be used for image analysis in remote sensing [74].

3.2 Assessment of the State of the Art

Now that the relevant projects to this thesis have been introduced it is time to assess them against the requirements for the Agent in a Box that have been outlined in the introduction chapter in section 1.1. As stated at the outset of this chapter, the evaluation of these projects is not an assessment of the project or the work that was done. All of these projects have made useful contributions in their own right. This evaluation assesses the work in terms of suitability for the purpose of achieving the goals of the Agent in a Box, which was not necessarily the goal of these projects.

The assessment begins with how the agents were connected to the environment in section 3.2.1, focusing on flexibility for different mobile robots, sensors, and actuators; and the performance of the system. Section 3.2.2 discusses the agents themselves, including the activities that these agents demonstrated and any generalized framework that was used.

[1] Although the benchmarks were proposed for reinforcement learning they may also be useful for other agent implementation methods.

3.2.1 Connecting to the Environment

In order for an agent to control a mobile robot, it must connect to that robot's sensors and actuators, enabling the agent to perceive its environment and take action. As stated earlier, projects were assessed on the requirements of the Agent in a Box, which may not necessarily align with the goals of the projects. The requirements include a desire for a method that is flexible to be used to control different robots in a variety of different domains using a variety of sensors and actuators without significant new development; the use of ROS for the opportunity to leverage nodes developed in the ROS community; and a need for the agent's reasoner to have sufficient performance to keep up with the demands placed on it by controlling these robots. Establishing flexibility for the Agent in a Box is discussed first in section 3.2.1.1, followed by a discussion of the system performance when controlling real robots in section 3.2.1.2.

3.2.1.1 Flexibility

The goal of the Agent in a Box is to provide an agent that can be used for controlling a variety of mobile robots with different sensors and actuators without significant redevelopment effort. Although all projects demonstrated an ability to interact with various sensors and actuators for at least one robotic platform, the interest is with respect to the flexibility of their approach. Was their project meant for a specific application or was it easily adaptable to other mobile robotic applications? One way for a project to demonstrate its flexibility was the project's use of ROS, which comes with the potential of taking advantage of other external nodes for connecting to various sensors and actuators. Whether the state of the art has provided this type of flexibility is discussed here.

The Abstraction Engine project provides a framework and design guidance for how an agent connects to sensors and actuators. This is beyond simply providing a connection to ROS, it provides guidance on how to connect to the hardware, and was demonstrated on multiple simulated and real robotic platforms [58, 59]. The JaCaROS project used ROS for connecting an agent to sensors and actuators, though guidance on how to setup those sensors and actuators was not apparent [51]. They defined the details of the different perceptions in a manifest file, which specified the ROS topics and formats of different data sources and how the data sources should be formatted as perceptions. The experiments in the JaCaROS project used a turtle bot simulation in Gazebo with simple scenarios. The sensors and actuators included those already provided by the simulation in addition to a battery sensor that was simulated outside of Gazebo. However, the battery sensor does not appear to have been demonstrated on multiple platforms but has apparent flexibility for other sensors and actuators. Meanwhile, PROFETA was demonstrated on two different simulated platforms, although it was unclear what level of

abstraction was provided by the framework [52].

Other projects which did not provide examples of multiple platforms, or much flexibility for different types of sensors and actuators include: the Codarra Avatar UAV project [42], the JaCaMo UAV [6], and ARGO [55, 56]. In the case of the Codarra Avatar UAV the mission management was performed using a BDI agent with the flight control on an autopilot which handled the lower level trajectory and altitude management. The architecture was designed to use a "single, linear thread of control", meaning that the agent checked for sensor data at the beginning of each reasoning cycle. The project's design suggests that the approach is likely not flexible to other robots without new development burden. The Codarra Avatar UAV design was similar to the JaCaMo UAV which used Python implemented controllers for managing the hardware. These controllers monitored a MAVLINK communication channel for commands from the BDI agent. In the JaCaMo project, all the sensors and actuators used were provided by an autopilot controller. The implementation of this agent used a significant amount of hard coding rather than code that may be more flexible for other types of mobile robots. Lastly, ARGO connected the Jason agent to the sensors and actuators using overridden classes in the Jason framework. This means that any new sensors or actuators would need to have drivers written into the Jason framework in Java, rather than using any external software for example, that may have been developed for using ROS.

3.2.1.2 Performance

In order for the agent to be effective at controlling a mobile robot the reasoning system needs to keep up with the demands of its sensors and actuators. Although many of the related works made claims that they were developed for embedded applications, not all of these projects provided evidence that they were tested and demonstrated on embedded platforms. The projects which did provide explicit evidence of their use on embedded computers included: the Codarra Avatar UAV [42], the JaCamo UAV [6], ARGO [55, 56], and Abstraction Engines [58, 59].

The Codarra Avatar project integrated a BDI agent onto an HP iPaq [42]. Flight tests of a navigation focused mission with the UAV were successful in July of 2004. Similarly, the JaCaMo UAV used a BDI agent on an embedded computer [6]. Although the JaCaMo UAV paper does not specify the specific computer that was used, it does provide some performance results with respect to the Central Processing Unit (CPU) and memory usage, which was 85% of the available memory and 45% of the available CPU capacity. The ARGO project claimed to not be tied to a specific hardware or Agent Oriented Programming (AOP) language, used Javino, a Java library for linking Java to Arduino computers, specifically intended for use with Jason [55, 56]. Citing Stabile and Sichman [57], who found that the agent's belief update and unification

accounted for up to 99% of Jason's execution time, the JaCaMo UAV authors expressed concern about the computational burden of the agent reasoning cycle. The proposed solution for the JaCaMo UAV was to use perception filters so that the agent was not required to update as many beliefs or unify as many variables. Lastly, the Abstraction Engines project was developed out of practical concerns for implementing GWENDOLEN and Jason BDI agents for real robots [58,59]. The Abstraction Engines were designed to address the concern that continuous signals could overwhelm the reasoner, which has been demonstrated using simulated platforms, turtle bots, and LEGO NXT robots.

While the JaCaROS project was intended for embedded applications, literature providing details of any demonstration of this capability was not found [51]. This is also the case for PROFETA, which used a Python implementation of BDI and the AgentSpeak language designed for use with autonomous robots [52]. The PROFETA authors were interested in determining if Agent Oriented Programming (AOP) can be implemented with Python for simpler robotic implementations, implying an intent for their agent to be used in embedded applications. However, they did not provide any metrics for assessing if this has been achieved.

3.2.2 Agent Behaviour

In addition to the connection between the agent and the environment, the Agent in a Box aims to provide a generalized approach for implementing agent behaviour. This should reduce the development burden for specific application domains, allowing developers to focus on the unique aspects of their specific domain. This should also result in a measurable improvement of the software in terms of its software engineering properties, such as software maintainability (measured with coupling and cohesion) or cyclomatic complexity. In assessing the agent behaviour in the related works the interest was whether the work provided a generalization for the agent's behaviour. In other words, could the project's implementation be easily adapted to another type of mobile robot?

To answer this question, the behaviour that the various works demonstrated, focusing on several general activities that have been identified in the Agent in a Box requirements, is discussed. These include the avoidance of obstacles, management of resources, and navigation. This is discussed in section 3.2.2.1. Whether or not a framework was used for developing the agent is also examined. This includes a concern for whether any behaviour was generalized so that it could be used for agents to control a variety of robots and how behaviours or plans were prioritized and selected. This is discussed in section 3.2.2.2.

3.2.2.1 Agent Activities

It is understood that an agent must adapt and act appropriately in the presence of conflicting and changing mission objectives. This includes the need to navigate through the agent's environment as part of achieving its mission, but the agent must also be able to handle any mission interruptions. These interruptions could be for obstacle avoidance or resource management, such as seeking maintenance, recharging a battery, or refueling. Here, the state of the art is examined to see if these projects have demonstrated these behaviours.

The Codarra Avatar agent, implemented in JACK, was used to fly a UAV. Although the aircraft needed to navigate through its environment, this was performed through the use of an autopilot module [42]. The aircraft had a limited ability to handle safety-related issues by declaring an emergency as needed. Although the agent was successful at flying the aircraft, its role seemed to be limited. Similar to the Codarra Avatar agent, the JaCaMo UAV was flown using an autopilot module which was responsible for the navigation and movement of the aircraft [6]. Although this agent was designed for a search and rescue mission, all the locations of the persons in distress were hard-coded in the agent's program, meaning that the agent was simply passing a list of locations to the autopilot.

Shifting to ground-based agents, the JaCaRos demonstrated an agent managing a simulated battery while working on its mission [51]. The agent did perform some navigation, although this was simplistic due to the nature of the environment; the agent only needed to move between corners of a square. The PROFETA agent used a more sophisticated form of navigation, using Dijkstra's algorithm to generate a path using an action call [52]. It also avoided obstacles, non-cooperative agents that were situated in the same environment.

The ARGO project primarily focused on the agent's reasoning rate, testing if the car that the agent controlled could stop before colliding with a wall, a type of obstacle avoidance behaviour [55, 56]. In the case of the Abstraction Engines project, mission interruptions, such as obstacle avoidance, were demonstrated [58, 59]. However, the navigation aspects of the behaviours were generally a hard-coded list of waypoints.

Although the state of the art has demonstrated the ability for an agent to control a mobile robot, none of them demonstrated the complete combination of navigation, resource management, and obstacle avoidance. Furthermore, many of these works seemed to be agents that were simply providing waypoints to some other autopilot controller which was ultimately responsible for controlling the robot.

3.2.2.2 Agent Framework

Now that the types of activities that the agents could carry out, such as obstacle avoidance, navigation, and resource management, have been considered attention can shift to how the agents were implemented. Specific focus will be placed on aspects that have been generalized, as is needed for the Agent in a Box.

Considered first was which BDI implementation was used. Familiar tools, which implement the formal foundations of BDI, such as Jason and the AgentSpeak language, should be used for defining and selecting the agent's behaviour. Needed here is a reasoner that follows the strict properties of Agent Oriented Programming (AOP), and enforces the semantics specified by the AgentSpeak language. The JaCaMo UAV [6], JaCaROS [51], and ARGO [55, 56] all used Jason for their agents. The Abstraction Engines project was primarily focused on the use of GWENDOLEN, making use of the model checking functionality, however they have also demonstrated their approach with Jason [58, 59]. The Codarra Avatar [42] and PROFETA [52] projects each used different agent implementations, using JACK and a custom made Python reasoner, respectively.

Considered next was whether the agent's behaviour, written in AgentSpeak or some other similar language, was developed with the use of reusable or generic elements, such as a framework. The use of a framework would allow for the reuse or refinement of the mobile robot behaviours by the developer for mission-specific needs. In addition how the agent selected which plans to set as intentions for the agent at any given context, especially in situations where there could be several applicable plans available to the agent was of interest. For example, ensuring that plans for obstacle avoidance were considered at the highest priority over mission management related plans. If the agent does not provide this, there may be a need for the developer to provide the agent's plans in a relative order, for example, as Jason's default selection function selects the first applicable plan in the plan base. Another alternative could be that the plans have mutually exclusive context checks so that only one plan is applicable for any given context. Both of these options present significant challenges for development and refactoring the agent's software as any change to one plan may require modifying several others, increasing the likelihood of error. This could result in less maintainable and more complex software with poor coupling, cohesion, and cyclomatic complexity. Ideally, the developer should not need to concern themselves with the order of the plans that they provide, nor should the addition or modification of new plans require the modification of other possibly unrelated plans.

The Codarra Avatar UAV performed a navigation focused mission with their behaviour focused on emulating the OODA loop [42]. Although the authors pointed out that there was additional complexity of building an agent to fly the UAV that may be viewed as an added burden

of development, they also saw an advantage to making the development of drone behaviours more "intuitively accessible" to domain experts, such as pilots, who already naturally think in terms of OODA. Unfortunately there was no discussion of generalizing this approach for other types of missions or applications.

The agent framework aspects of the Agent in a Box requirements, which speak to providing generic behaviour which can be used for implementing specific mobile robots, were lacking in the state of the art. Lastly, although a number of projects demonstrated obstacle avoidance behaviour, implying that the agent was able to properly select the highest priority plan for the given context, there was no discussion on how the agent selected the appropriate plan.

3.3 Summary, Strengths, and Gaps in the State of the Art

This chapter has outlined the state of the art in the applications of BDI for autonomous mobile robots with an eye for robotic applications. Table 3.1 provides a summary of the features of the various projects, assessed against the requirements established for the Agent in a Box in section 1.1. Methods that fully provided these features were given a ✓. If the project did not fully address the feature they were given a ✗. As these projects were not necessarily developed with these requirements in mind, this evaluation should not be a judgement on the quality of the work that was done, but rather how well that work aligns with the requirements of the Agent in a Box.

Table 3.1: 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 real robot	✓	✓	✗	✗	✓	✓
	Controlled simulated robot	✗	✗	✓	✓	✗	✓
	ROS compatible	✗	✗	✓	✗	✗	✓
Agent Behaviour	Uses popular BDI reasoner (ex: Jason)	✗	✓	✓	✗	✓	✓
	Behaviour framework	✗	✗	✗	✗	✗	✗
	Behaviour prioritization	✗	✗	✗	✗	✗	✗
	Obstacle avoidance	✗	✗	✗	✓	✓	✓
	Resource management	✗	✗	✓	✗	✗	✗
	Navigation	✗	✗	✗	✓	✗	✗

As mentioned before, the projects surveyed in the state of the art were 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, using either simulation or real robots. Unfortunately, this was the only requirement of the Agent in a Box that was provided by all surveyed projects. None of them demonstrated all Agent in a Box requirements, although they all made valuable contributions in their own right. The literature for several of the projects expressed a concern of computational performance of the agents being used to control the robots, making use of perception filtering to ensure that the agent was not overburdened by sensor data.

In general, many of the surveyed projects used ad hoc approaches for their implementation. Additionally observed in the works 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 could provide attitude, position, and velocity data. There was less emphasis, however, on payload sensors which would be used for mission activities. The stand-out project in the state of the art was the Abstraction Engines project, which demonstrated the feasibility of all requirements established for how the Agent in a Box connects the agent to the environment, although this project did not address many of the agent behaviour requirements. This project provided a valuable insight with respect to how to effectively connect a BDI agent to the sensors and actuators found on real mobile robots. This connection is the beginnings of an infrastructure for autonomous mobile robots using BDI reasoning.

Missing from all surveyed projects was any discussion of a framework for an 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. This would allow developers to leverage these generic elements for different robots, focusing only the plans that are unique to that specific robot. There was also very little discussion of the properties of the software that was used to implement the agents.

Also missing from all works was any discussion of how the agent selected its behaviour, either through prioritization or some other method. The default selection method in Jason, for example, is to select the first applicable plan for the first event in the event queue. This plan selection method ties the agent's plan selection to the order in which the plans were loaded in the plan base rather than having the plans deliberately selected based on what the most appropriate or highest priority plan for any given circumstance should be. The Agent in a Box provides a method for selecting plans for execution based on their relative priority, and is discussed in the next chapter.

Chapter 4

Methodology

The validation of a framework such as the Agent in a Box comes with a number of challenges. To validate a framework it must demonstrate that the framework provides useful generic functionality to developers, an ability to map domain specific functionality to the framework, and that the use of the framework does not introduce some undue performance burden. The following paragraphs provide a description of the experiments used to assess these properties and validate the Agent in a Box. The details and results of these validation experiments are provided in chapter 7.

The first part of the validation is to assess that the Agent in a Box does indeed provide useful functionality for a variety of different mobile robots. In the case of the Agent in a Box, it provides a set of generic behaviour plans which are useful for a variety of different mobile robots as well as how those plans should be selected to run. Therefore, the first step in validating a framework should be to first assess if the Agent in a Box is indeed providing useful generic functionality for various different mobile robots. Simply put, the Agent in a Box needs to work. This can be demonstrated by using the Agent in a Box to control a variety of autonomous mobile robots, in this the case a grid agent, a mail delivery robot, and a simulated autonomous car, which are described in chapter 6. These case studies make use of the generic navigation plans, which generates routes as plans, using domain specific movement plans and map information. They also demonstrate customised movement, collision avoidance, and energy autonomy. This was validated by assessing the run logs of the different agents to ensure that the agents were behaving appropriately.

The next question asks how well the the Agent in a Box works. This relates to whether or not there is a runtime performance benefit or cost to using it. The quantitative assessments focused on the processing time needed for performing key computations such as: the length of the reasoning cycle, the time needed for the agent to make key decisions or take key actions

given specific inputs. These tests were used to assess how responsive the agents were compared to various alternative designs. Among these alternatives were different approaches to how the agent performed navigation – either using routes generated internally to the agent using logic implemented in AgentSpeak, using an internal action, or using a support module implemented in the environment. Another dimension of the comparison looked at alternative agent implementations including agents implemented using AgentSpeak programming without the use of the Agent in a Box and agents programmed purely using imperative programming (using Python).

The second set of quantitative assessments focused on the software engineering properties of the Agent in a Box. The goal here was to assess the level of difficulty for a developer who may be working with the Agent in a Box. This was measured through assessing the maintainability of the software, using coupling and cohesion, as well as the overall complexity of the software, using cyclomatic complexity. These assessments were applied to the Agent in a Box, alternative AgentSpeak agents, and to the imperative programming version.

Chapter 5

Approach

Looking back to the research questions presented in the introduction, this thesis is aimed at determining if BDI agents are capable of controlling mobile robots and if it is possible to generalize aspects of how those robots are controlled using a framework, hopefully simplifying the development of new robots using BDI. As was apparent from the state of the art, although there are a limited number of examples of BDI being used to control autonomous mobile robots, it clearly is possible to do so. That said, the examples discussed made use of ad hoc implementations with hard coding and there did not seem to be much concern for flexibility for working with different types of robots, beyond how the agent was connected to the hardware, nor was there much emphasis on the design of the agent's behaviour itself. This means that the examination of how BDI agents can be generalized for use with different types of robots, such as with a behaviour framework, has not been examined in the state of the art. The use of a framework may impact the runtime performance of the agents by possibly adding additional overhead. It may also affect the agent's software engineering properties, such as coupling, cohesion, and cyclomatic complexity. Hopefully, the use of a framework should improve the development experience by removing the need to develop parts of the agent's behaviour that can be provided by the framework, ideally reducing the likelihood of development errors.

In order to perform the validation experiments, the Agent in a Box is needed. Building from the established requirements discussed in the introduction chapter and the strengths of existing work discussed in the state of the art, the Agent in a Box was developed in three main themes. These themes include the development of the agent's behaviour, the customization of the agent's reasoning cycle to ensure that the appropriate behaviour is selected, and the connection of the agent to the environment, more specifically the robot's sensors and actuators. The goal of the Agent in a Box is to simplify the development of mobile robots using BDI by providing features that are common to a variety of mobile robots. The Agent in a Box handles the common aspects

for mobile robotic agent's in an extensible skeleton, while the developer provides the needed customization for their specific application for the framework's generic algorithms to use as needed. Recommended design approaches for performing this mapping, which have been found to work well with the Agent in a Box, are provided. The Agent in a Box also uses tools that should be familiar to developers. The Agent in a Box uses Jason, a popular BDI reasoning system which uses AgentSpeak. The Agent in a Box 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, however, use different types of sensors and actuators. Thus, there is a need for flexibility in the design of how an agent connects to the sensors and actuators so that different devices can be connected to the agent relatively easily. Furthermore, the development of these connections using tools that are familiar to the robotics community is desired. 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 Agent in a Box architecture, inspired by the Abstraction Engines approach discussed in section 3.1.6, is discussed in section 5.1.

With the connection between the agent and the environment established, it is necessary to consider the agent's reasoning system and how it selects which of the agent's behaviour plans will be executed at any given time. In considering the desired behaviour of a mobile robot, the relative priority of desired behaviours is always the same: safety related behaviour (such as for obstacle avoidance) is always the highest priority, followed by energy autonomy, navigation, and then mission. This provides an opportunity for the Agent in a Box to handle this prioritization on behalf of robotics developers, providing a generic and reusable method for the agent to select the most appropriate plan at any given time. The method for plan selection is discussed in more detail in section 5.2.

The next 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 Agent in a Box provides generic plans which handle several aspects of controlling a mobile robot. To use these generic plans, a developer for a specific application domain must provide the needed mapping of these generic plans, and any needed domain-specific plans, for the agent to control this specific robot. The robot's mission was considered to generally include some combination of moving to one or more locations, using a navigation sub-framework which generates a path in the form of a plan, and performing some actions at those locations. A mobile robot should perform its mission

while avoiding obstacles while also ensuring that it has sufficient resources, such as a charged battery, while working. If it encounters a blocked path on its route 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 maneuver 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 prioritized by the reasoning system. The details of the behaviour framework are provided in section 5.3.

5.1 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. An *impedance mismatch* problem [75] exists between the fields of mobile robots and BDI agents. The term impedance mismatch refers to the conceptual and technical issues faced when integrating components defined using different methodologies, formalisms or tools. In this section, the method of addressing the impedance mismatch is provided. Jason provides tools for connecting the agent to relatively simple environments. This method ties the agent's reasoning cycle to the updating of the environment, meaning that the agent waits for the environment to provide an update before reasoning about what it perceives and taking an action. It also means that the environment waits for the agent to act before updating. As real world environments are continuously changing, and do not wait for agents to make decisions, there is a need to demonstrate that the agent's reasoning cycle and the environment update could be decoupled. This was demonstrated by decoupling the agent's reasoning cycle from a simulated environment using a project called SAVI, discussed in section 5.1.1. The results from this test, which shows that the reasoning cycle can be decoupled from the environment update successfully, are provided in the validation chapter.

Building on this early work on SAVI, the SAVI project was reworked to provide a Jason agent with a connection to ROS. Discussed in section 5.1.2, this includes the SAVI ROS BDI node as well as the other nodes that are needed for providing the agent with sensory perception, control of the robot's actuators, and the ability to communicate. These nodes take inspiration from the Abstraction Engines project that was discussed in the state of the art chapter.

5.1.1 Decoupled Reasoning Cycle

As discussed in the previous paragraphs, there is an impedance mismatch between how existing BDI frameworks connect to their environments and the capabilities of modelling and simulation architectures. The SAVI project provided an architecture for simulating autonomous mobile robots [7, 76]. SAVI decouples the agents from the simulation environment, making it easy to develop them independently, and allowing them to run as separate processes interacting in an asynchronous manner. This avoids linking the environment's simulation updates to the agent's responses – if the agent is slow for some reason, perhaps due to expensive computation, the simulation will continue unaffected, making the transition to a natural environment more realistic.

The agents were connected to the environment using SAVI, shown in figure 5.1. In this architecture, the simulated environment and each BDI agent were run as separate threads of execution. The flow of perceptions and actions between the agents and their simulated presence in the environment was accomplished with the state synchronization module, which ensured thread safe data flow between these components. The simulation time step could be configured by setting the frame rate in the simulator. In doing this, the agents were tested at different simulation rates to confirm if they can complete their missions. This also highlighted that the reasoning cycle was indeed independent from the simulation time step, showing that the simulation did not wait for reasoning cycles to be completed before moving on, and that the reasoning system could keep up with the simulation rate. To demonstrate the separation of these time steps, the simulation was run at various frame rates. The results of this experiment are provided in the validation chapter in section 7.2.1.

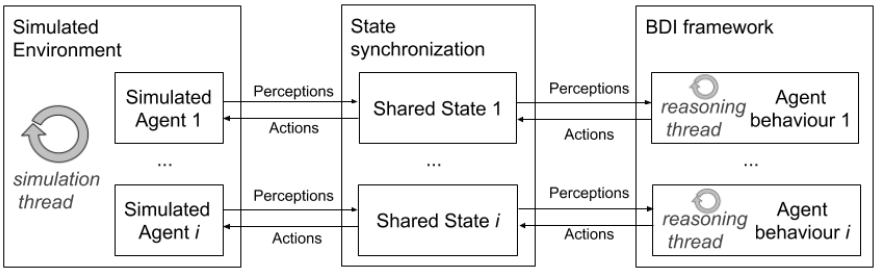


Figure 5.1: SAVI Architecture [7].

For the Agent in a Box, the architecture for connecting the agent to the environment was built upon the successes of the SAVI. This architecture is discussed in the next section.

5.1.2 Agent in a Box Architecture

Similar to the field of modelling and simulation, an impedance mismatch also exists between the BDI agents and the software used for controlling robots. A popular method for controlling robots is the use of ROS, a distributed architecture which provides a publish and subscribe interface for a variety of nodes responsible for different aspects of controlling robots. As discussed in the background chapter, using ROS enables the use of a variety of third party nodes that are available in the ROS community. Given its popularity, and the availability of a variety of modules that may be useful to developers, it was opted to use ROS for connecting the reasoner to the environment.

The Agent in a Box architecture for connecting the agent to the environment builds on the success of the SAVI project discussed in the previous section. The main difference is that this architecture uses the design approach from the SAVI project to connect the BDI agent to ROS rather than to a specific simulator. The connections to the various sensors and actuators for the robot are provided as different ROS nodes as shown in figure 5.2. At a high level this architecture includes the agent reasoner, connected using SAVI ROS BDI and discussed in section 5.1.2.1, and a set of application nodes and environment interface nodes, discussed in section 5.1.2.2.

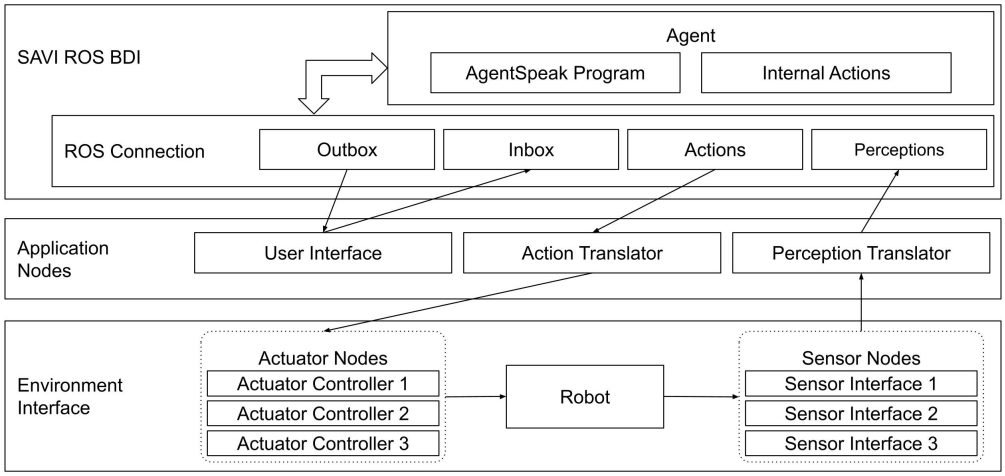


Figure 5.2: Connecting the Agent to the Environment.

5.1.2.1 SAVI ROS BDI

SAVI ROS BDI, available open source on GitHub [77], provides the necessary abstraction that the agent requires to be suitable for connecting a variety of sensors and actuators, thus making the Agent in a Box available for multiple platforms. This node decouples the agent's reasoning cycle from the implementation of the sensor and actuator interfaces.

The internal architecture of SAVI ROS BDI is shown in figure 5.3. This node consists of a set

of ROS connectors, which are responsible for publishing and subscribing to several ROS topics, such as `perceptions`, `actions`, the agent's `inbox` and `outbox`. By separating these components, they can each operate as separate threads, reducing the risk of bottlenecks, meaning that the node can be receiving perception data while also publishing the actions that were generated earlier. In parallel with all of this, the reasoner operates on its own thread of control, running a reasoning cycle whenever there is a set of perceptions available to it. These components pass data between each other using the thread safe state synchronization module, as was also the case with the SAVI project. The two listener nodes pass perceptions and messages received from their respective ROS topics to the perception manager and inbox located in the state synchronization module. The two publishing nodes, which publish actions and outbox messages, monitor the data stored in the state synchronization module.

SAVI ROS BDI's agent reasoner is housed in the agent core. This component triggers Jason's reasoning cycle when there is a set of perceptions available to it in the state synchronization module. Through the reasoning cycle, the agent deliberates on what to do next, such as taking an action, sending a message, etc. The behaviour of the agent is defined by an AgentSpeak program which contains beliefs, rules, and plans that are provided by the Agent in a Box, discussed later in this chapter, and domain-specific beliefs, rules, and plans that have been provided by a developer. Lastly, the agent core passes any actions and messages generated by the agent to the state synchronization module so that they can be published to ROS to the appropriate topics.

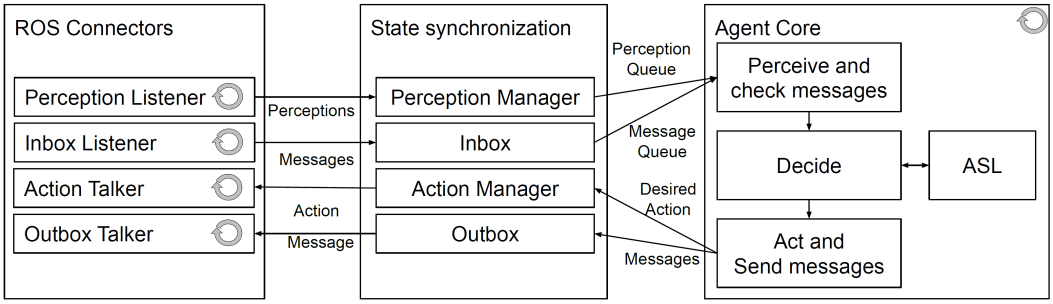


Figure 5.3: SAVI ROS BDI Internal Architecture.

With the BDI reasoner connected to ROS, it is now possible to connect the agent to the robot's sensors and actuators. It is not enough for SAVI ROS BDI to simply publish and subscribe to ROS topics, there need to be other nodes that communicate with the reasoner using these topics. These other nodes are discussed in the next section.

5.1.2.2 Environment Interface and Application Nodes

With the reasoner connected, it is necessary to consider how to pass useful data from the robot's sensors to the agent, and vice versa. This is the role of the environment interface and application

nodes. There are several challenges to doing this. The BDI agent requires data to be passed in the form of discrete predicates, however, many sensors generate continuous time signals. These signals need to be translated into discrete predicates for the reasoner to use. The reverse of this needs to happen for the agent's actions. The Agent in a Box approach to resolving this impedance mismatch takes inspiration from the approach provided by the Abstraction Engines project discussed in the state of the art section [58, 59]. The two main components of this are the environment interface nodes and the application nodes.

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 can be custom built nodes. The sensors are encapsulated in a set of individual nodes which publish their data to relevant ROS topics. The reverse of this was developed 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.

Application nodes are domain-specific nodes which translate between raw data published by sensors, or expected by actuator controllers, and the predicate structure used by the BDI agent. These 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 to 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. These perceptions for all the sensors are provided in a single message. This is important as the agent may have plans which uses data from multiple sensors in their context checks. If these perceptions are not provided together, there is a possibility that the plans which require data from multiple sensors may never be applicable. Lastly, the action translator subscribes to the `actions` topic. Similar to the perceptions, the agent's actions are provided in the form of 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 topic so that the relevant actuator node can react.

5.1.3 Summary of Environment Connection

The connection between the agent reasoning system and the environment is critical to the Agent in a Box. Needed was a means for providing the agent with a generic method of connecting it to

a variety of sensors and actuators, making it compatible with a variety of robots. The Agent in a Box bridges the impedance mismatch between Jason and ROS. Part of the challenge was the need to decouple the environment update from the agent's reasoning cycle using the SAVI ROS BDI node. With the connection between the agent reasoning system and the connection to the environment complete, attention can now turn to the agent itself. Specifically, this includes the prioritization of the agent's behaviour and the agent's behaviour framework. These are discussed in the following sections.

5.2 Prioritization 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 to set as an intention. As was discussed at the beginning of this chapter, it is expected that mobile robotic agents will have a need 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 it finds an inconsistency between its map and its observations of the environment, it needs to update its map.

In considering these types of activities, it is proposed that they 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 2.2.1, the Agent in a Box's agent architecture uses a prioritization scheme with respect to the selection of which applicable plan will be added to the agent's intentions. As AgentSpeak gives us a full programming language, it is possible to implement subsumption behaviours, but also other behaviours if needed. The prioritization scheme for the framework is provided in figure 5.4. The prioritization 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.

In considering how to incorporate behaviour prioritization into Jason's reasoning framework, it is first necessary to revisit Jason's reasoning cycle in more detail. Shown in figure 5.5, the focus is on the how the reasoner determines what plans should be set as intentions. When an event occurs, which can be a change in goals or beliefs, they are passed to an event selection function S_E, shown in diamond 5 in the figure, highlighted with a red marking. This function

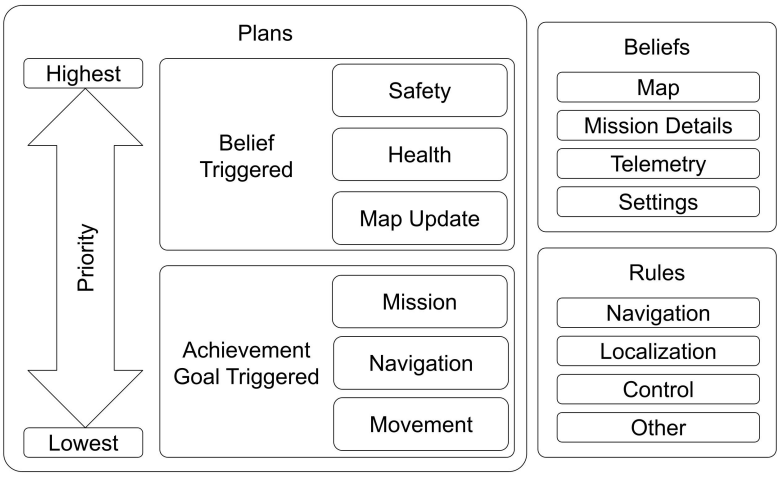


Figure 5.4: Behaviour Prioritization.

selects which event will receive the attention of the reasoner for that cycle. 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. The applicable plan is then loaded to the top of the intention stack.

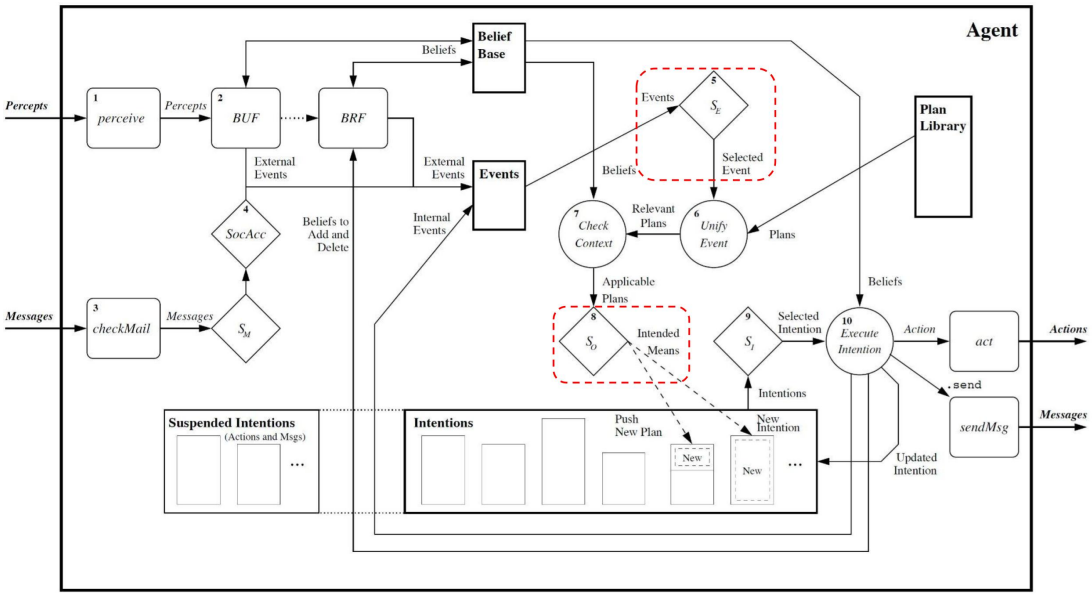


Figure 5.5: BDI Reasoning Cycle in Jason with Event and Option Selection Highlighted [3].

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. For the Agent in a Box, however, this is complicated by the fact that there are plans provided by a framework

as well as by the domain-specific developer. Therefore, 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. There is also 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 prioritize the agent's behaviour.

The simplest way for the reasoner to select an event and an option, would be for the reasoning system to use a total ordering of the plan priorities for each event and option. Such an ordering would make the selection of events and options a trivial matter for the reasoning system. It would simply need to select the highest priority event and option. The problem with using a total ordering is that it could make it challenging to refactor the code as any change in the plan could affect the ordering of all of the other plans. Another approach could be to categorize the plans based on the types of behaviours that they implement. Indeed, it is expected that any mobile robot should prioritize safety over its mission, for example. If the reasoner were to have knowledge of the types of behaviours that each plan implements it could use that as part of the plan selection process.

One approach for providing the plan categorization to the reasoner could be to provide annotated event triggers, which could be used by the event selection function for prioritizing 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. The need to annotate the trigger at the source could be particularly challenging with respect to the belief-triggered plans which use perception-generated beliefs. The perception translator, which was discussed in section 5.1, would need to generate perceptions with annotations which reflected how the agent would use them. It also means that all other achievement goals also need to be annotated when the goal is adopted, which is often in the plan body of other plans. This leads to the potential of some event triggers being missed in the event of refactoring. Ultimately, the use of trigger annotations was rejected for these reasons.

Having rejected the use of priority annotated event triggers, another means of specifying and selecting the highest priority event is needed. The solution is to have a set of beliefs that the agent has with respect to which event triggers are used for what 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 is shown in listing 5.1. Here,

Listing 5.1: Behaviour Prioritization Beliefs.

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

the trigger names for different types of behaviours are identified. The event triggers can then be prioritized by the event selection function so that the appropriate event trigger is selected. The terms have been colour coded to illustrate which terms are associated with components provided by the Agent in a Box or components that are not provided but are expected by the Agent in a Box. The terms coloured in blue are Agent in a Box provided beliefs or event triggers. The terms coloured in green are for components that are expected or used by the Agent in a Box that need to be provided by the developer. This framework only works if the developer provides these expected beliefs for the reasoner to use for its prioritization. In section 5.3, the definition of the Agent in a Box behaviour framework is discussed, including what behaviour is provided by the framework and what needs to be provided by the developer of an agent for a specific domain.

With this prioritization knowledge, the reasoner's event selection function can determine the type of behaviour associated with each triggering event. This can then be used to select the highest priority event. The implementation of this prioritization is illustrated in figure 5.6. 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.

With the highest priority event having been chosen, it is now necessary to select the best option for the intention queue. Prior to calling the option selection function, the reasoner checks the context of all plans triggered by the selected event. This means that all of the possible options are plans applicable to the context of the highest priority event, as chosen by the event

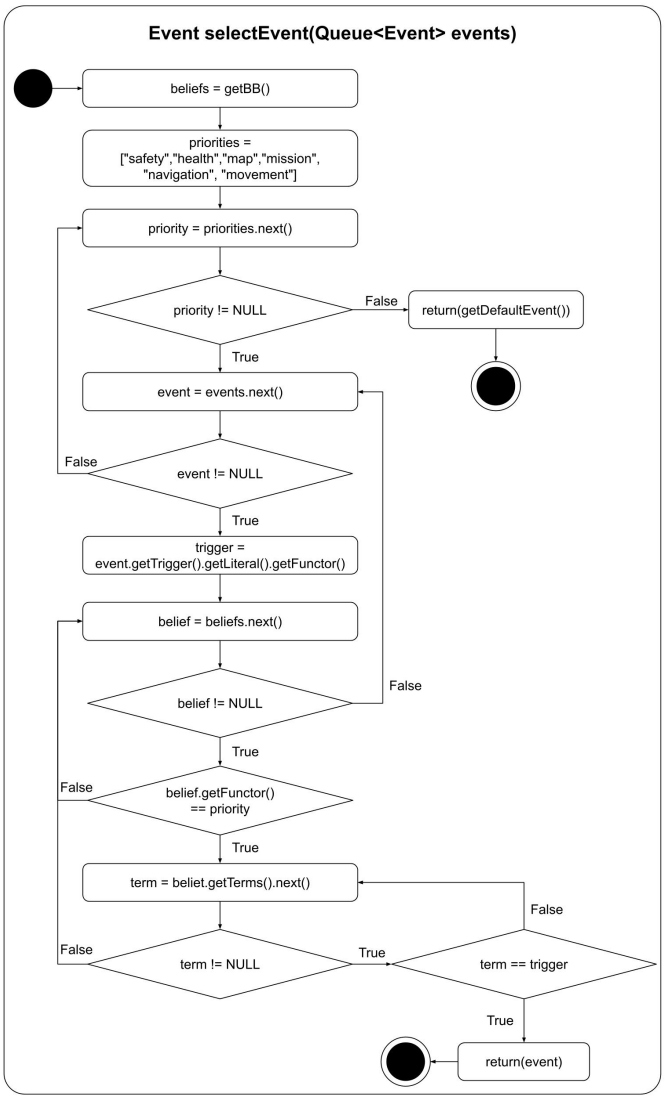


Figure 5.6: Event Selection Function.

selection function. The agent reasoner must now select which option it should add to the agent's intention queue. 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 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 plan contexts 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 5.7. 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. This software is available in the behaviour framework repository [78] as well as in SAVI ROS BDI [77].

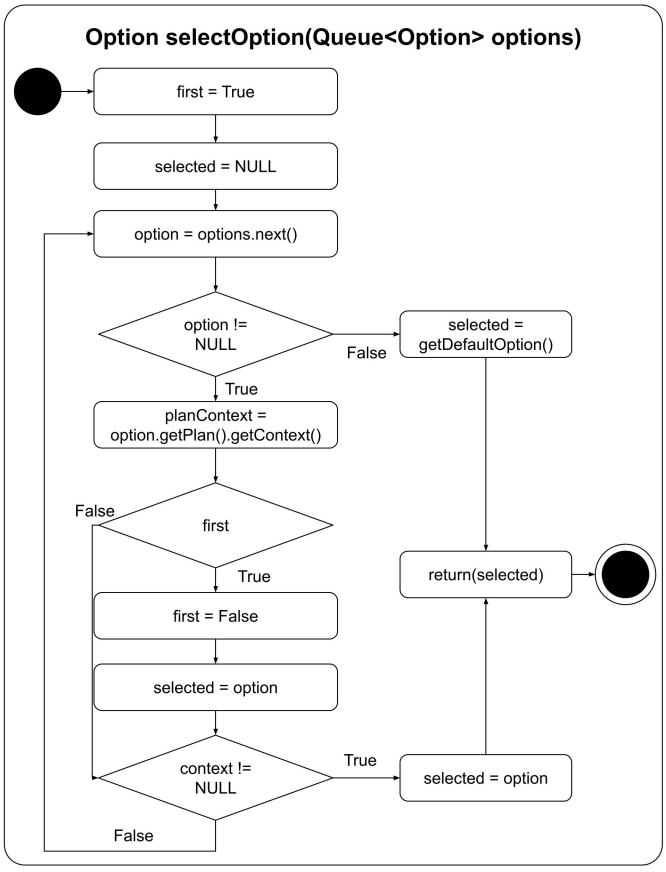


Figure 5.7: Option Selection Function.

With the overridden event and option selection functions complete, the Agent in a Box has a method for selecting the highest priority event and plan 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.

5.3 Behaviour Framework

The Agent in a Box provides generic behaviour that is useful for a variety of mobile robots. Represented visually in figure 5.8, this includes a set of plans which provide the agent with the ability to navigate by generating route plans using the navigation sub-framework, 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 prioritization to select the highest priority plan for any given context. Components that are provided by the Agent in a Box are shown in blue; Dependencies that the Agent in a Box expects, or needs in order to function, are shown in green. This software is available in the behaviour framework repository [78].

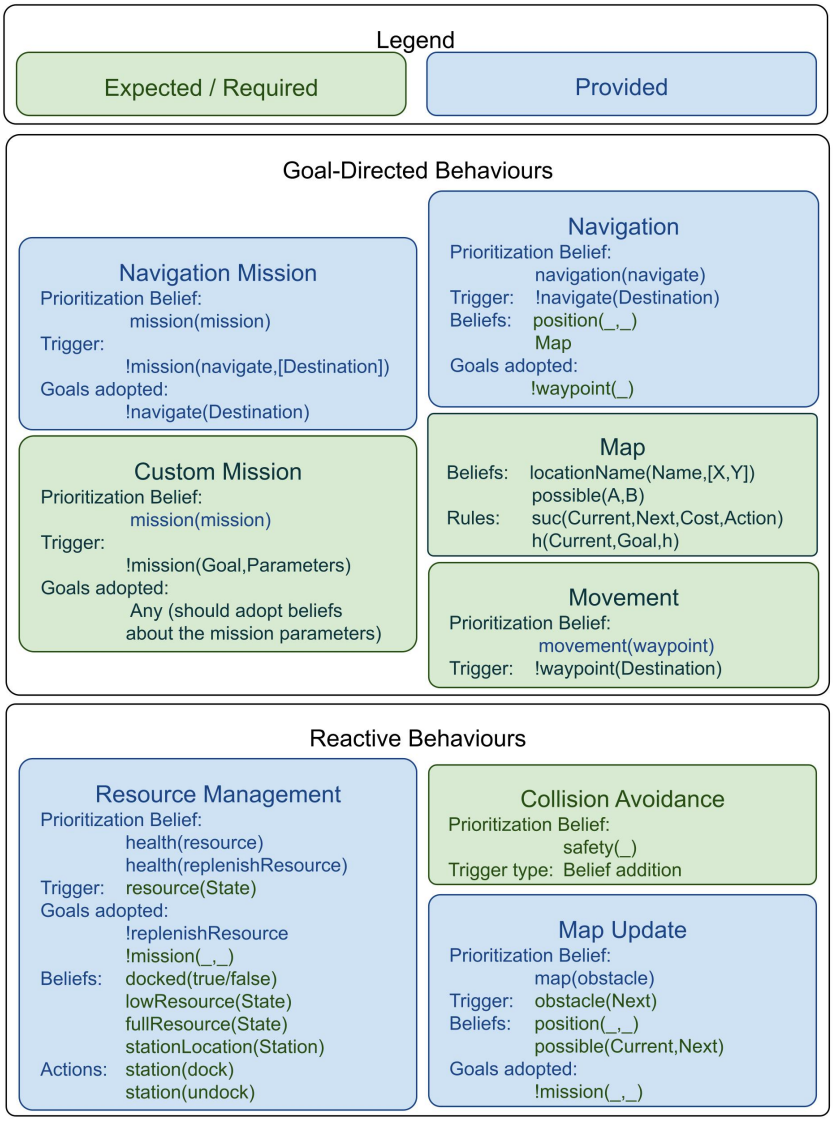


Figure 5.8: Behaviour Framework.

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, specialize, or customize 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. This section provides a variety of AgentSpeak listings which provide the implementation of various agent behaviours. These listings are colour coded in the same way as figure 5.8, showing the framework provided beliefs and event triggers in blue, and the beliefs and event triggers that a developer must provide when developing for their specific domain in green.

5.3.1 Navigation

No matter the application domain, all mobile robotic agents need an ability to plan a route to a destination. The Agent in a Box provides this behaviour in a navigation framework-in-a-framework. This navigation sub-framework includes the generation of the route as a plan as well as how the map is defined [8].

The agent's navigation behaviour is triggered by the adoption of the `!navigate(_)` achievement goal, which generates a route in the form of a plan for the agent based on a set of map beliefs. This automatically generated plan provides a route that consists of a set of `!waypoint(_)` achievement goals. By having the route provided in this way it can be easily monitored, interrupted, and suspended as needed by the reasoner. The developer for specific domains must provide an implementation of the `!waypoint(_)` achievement goal for their specific domain. Examples of `!waypoint(_)` implementations are provided with each case study in chapter 6. These two achievement goals are analogous to the roles of the global and local planners from the ROS navigation stack respectively.

In developing this navigation framework several design options were considered and tested to see if there was a performance difference. One approach could be to have the route generation be provided naively in AgentSpeak. Another option could use an internal action to generate the route. Lastly, there is also the option to remove the route generation from the agent and have it generated by a module in the environment. Was there a performance advantage for any of these approaches? Each of these options was developed and tested to assess its performance. These different versions are discussed in the following sections.

Listing 5.2: Navigation Plan [8].

```
{ 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);}.
```

5.3.1.1 AgentSpeak Navigation Module

The first version of the navigation module considered was fully implemented in AgentSpeak. This module provides the implementation of the `!navigate(Destination)` achievement goal, which is shown in listing 5.2, using Jason's A* implementation [79] 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 prioritization methods. The generated path is a list of location waypoints for the agent to follow. These are adopted as a 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 achievement goal are provided with each of the case studies in chapter 6.

The navigation plan shown in listing 5.2 requires the developer to provide several domain-specific parameters to work. The first of these parameters is a map that is written in AgentSpeak and includes predicates that specify the names and coordinates of the points of interest on the map, 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 second parameter is a set of successor state predicates, of the form `suc(CurrentState, NewState, Cost, Operation)`, which provides A* the cost of moving between any two nodes. The third parameter is a heuristic predicate, which provides the estimated cost for moving from any given node to the destination. This take the form of `h(CurrentState,Goal,H)` [8]. The fourth and final parameter is a set of 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 the case studies chapter. 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.

Listing 5.3: Internal Action Navigation Plan.

```
+!navigate(Destination)
    :   position(X,Y)
        & locationName(Current,[X,Y])
    <-  +destination(Destination);
        navigation.getPath(Current,Destination,Path);
        for (.member(NextPosition, Path)) {!waypoint(NextPosition);}
        !navigate(Destination).
```

5.3.1.2 Internal Action Navigation Module

As mentioned previously, an alternative implementation of the navigation module, which uses an internal action for route generation, was considered. This section discusses the use of an internal action for generating the agent's route instead of using the AgentSpeak implementation of A*. The plan version which uses an internal action for navigation is shown in listing 5.3. In this scenario, the `navigation.getPath(Current, Destination, Path)` internal action, which uses the AIMA3e library implementation of A* search [80, 81], provides the path for the agent to follow. As was the case with the AgentSpeak design, the path is a list of locations that the agent needs to visit. The path is loaded as a set of achievement goals for each position in the route. An additional internal action for updating the map was also provided.

5.3.1.3 Environment Support Module for Navigation

In addition to the internal action version of the navigation routine, an environmentally supported version of the navigation framework was also considered. In this case, the agent commands an environmentally situated module to calculate a route. The agent receives the generated route via a perception. The environmentally supported approach for providing a navigation route is provided in listing 5.4. These plans use an action in the environment that the agent can use to generate a path for it to perceive and apply. A version of this method was implemented directly in Jason using the AIMA3e library, as was also the case with the internal action. This module was also implemented as a ROS module using the python-astar library [80–82] for agents that connected to their environment using ROS.

The first plan in the listing deals with the perception of a path which is added to the knowledge base as a mental note. The next plan, tied to the `!navigate(Destination)` achievement goal, adopts achievement goals for all the locations in the route that was loaded to the belief base. Lastly is the plan which uses the `getPath(Current, Destination)` action, which commands the navigation module to generate a path to the destination. An additional action was provided so that the agent could update the map as needed.

Listing 5.4: Environment-Supported Navigation Plans.

```
+path(Path)
    <-  -route(_);
        +route(Path).
+!navigate(Destination)
    :   route(Path)
    <-  for (.member(NextPosition, Path)) {!waypoint(NextPosition);}
        -route(Path);
        !navigate(Destination).
+!navigate(Destination)
    :   position(X,Y)
        & locationName(Current,[X,Y])
    <-  +destination(Destination);
        getPath(Current,Destination);
        !navigate(Destination).
```

5.3.1.4 Navigation Summary

This section has detailed three versions of the navigation module which were developed for the Agent in a Box. All three of these modules provided plans which implemented the navigation achievement goal `!navigate(_)` to send the agent to a location on the map. These different modules each generated a route for the agent to follow in the form of a plan that the reasoner can monitor and interrupt as needed. This route plan was a sequence of `waypoint(_)` achievement goals. By specifying an application domain-specific map, the implementation of the plans triggered by the `waypoint(_)` goal, and in the case of the AgentSpeak version, successor and heuristic predicate definitions, a developer can provide navigation functionality to their robot. The comparison of the performance of each of these methods is provided in the validation chapter, in section 7.2.4.

Now that the agent has the ability to navigate to a destination, it is necessary to look at how the agent's mission should be defined.

5.3.2 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 Agent in a Box as needed. To illustrate this point, let us assume that the agent is at location `A` in the environment shown in figure 5.9 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)`. This will result in the agent adopting achievement goals for `!waypoint(B)`, followed by `!waypoint(C)`, and `!waypoint(D)`. Let us complicate this mission by adding an additional challenge to this scenario: what would happen if, after the agent adopts its waypoint goals, it realises that it needs to go charge its battery at a station located at `E`? 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

in section 5.3.4, it can be presumed 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.

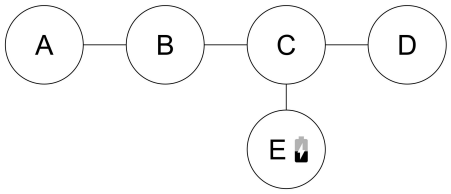


Figure 5.9: Mission Management Scenario.

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 with that is that the Agent in a Box needs to support many different types of domain-specific missions specified using achievement goals and parameters not known by the Agent in a Box. This means that the Agent in a Box's battery recharging plan needs to readopt an unknown, domain-specific achievement goal from its generic plan. Therefore, a generic mission-level achievement goal is needed so that other plans provided by the Agent in a Box can adopt it when needed.

Now that the need for a generic mission-level achievement goal has been established, let us consider what this goal needs to accomplish. It needs to specify both a specific mission, possibly among many possible missions that the agent is capable of, and it needs to specify any needed parameters that are relevant to that mission. Both of these need to be accomplished using a generic achievement goal that can be adopted by other plans provided by the Agent in a Box. The mission also needs to be implemented so that it can be interrupted, suspended, and readopted. This means that the mission 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. These beliefs ensure that the agent doesn't needlessly repeat any parts of the mission that have already been completed.

Listing 5.5 shows an example implementation of the mission management behaviour for a navigation mission using the generic `!mission(Goal,Parameters)` goal. Additional domain-specific missions should be specified using a similar format, enabling the agent to complete other types of missions. The achievement goal in listing 5.5 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 plan context checks for the mission type, allowing the plan body to be used for adopting the appropriate achievement goal that implemented the needed mission

Listing 5.5: Example Mission Management Plan for a Navigation Mission.

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

plan. The plan body also sets a mission-level mental note with the mission parameters, useful for readopting the mission if necessary. After adopting the mission-specific achievement goal, the agent can drop this mental note, as the mission has been completed. The navigation mission shown in this listing is included with the Agent in a Box, as any mobile robot 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 as missions are implemented using this type of plans they can be interrupted and resumed seamlessly by the framework provided plans.

Now that the agent has the ability to navigate through the environment and can be provided with domain-specific missions, it is necessary to consider what the agent should do if it encounters an obstacle.

5.3.3 Obstacle Avoidance and Map Update

As the agent moves through the environment, there is a possibility that the agent may need to avoid obstacles. It has been found that 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. The second type of obstacle avoidance behaviour is where the obstacle causes a significant change in the agent's mission context and the presence of this obstacle invalidates the agent's 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 an obstacle, like the one in the example, will require a change to the agent's goals.

Let us start with the simpler obstacle avoidance. Listing 5.6 provides a sample of this type of obstacle avoidance. Assume that the agent has a perception associated with this type of obstacle, in this case `pedestrian(blocking)`. Let us further assume 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 the plan is executed to its completion, without

Listing 5.6: Example of Simple Obstacle Avoidance.

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

Listing 5.7: 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);
        }.
```

any interruption, the plan is implemented with the `[atomic]` annotation. The event trigger must also be provided using `safety(pedestrian)` in order for the event to be prioritized 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.

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, the second type of obstacle avoidance is when there are obstacles which interrupt the mission. For example, if the agent is moving between waypoints that were generated by the navigation plan and then finds that the planned route is blocked, perhaps by a closed road, the agent can't continue. In this scenario, the agent needs to update its map to reflect that the route is blocked and regenerate the waypoint achievement goals that are in its intention queue.

The implementation of a map update is provided in listing 5.7. In listing 5.7, the agent is provided a belief that identifies that a map-related behaviour is triggered by `obstacle`. In this case, the plan is triggered by the agent gaining the `obstacle(Next)` belief, which needs to be generated by the perception translator based on the robots sensor data. With the obstacle observed, the agent may need to update the map. The context check verifies if this obstacle belief contradicts any belief that the agent has about moving from its current location to the location of the obstacle. If it does, the agent needs to update the map by dropping the inaccurate belief. The plan body then checks if the agent was on a mission, which would require the agent to drop any invalidated intentions before readopting them. This is done with a simple conditional statement in the plan body.

The need to avoid obstacles is not the only type of issue that can interrupt the agent's progress on a mission. The agent also needs to manage its resources, such as the charge of a battery, while working on its mission. The Agent in a Box's behaviour framework provides resource management behaviour, discussed in the next section.

5.3.4 Management of Resources

Another common aspect for mobile robots is the need for resource management. This could be to replenish a consumable resource, such as fuel, battery state, or even to seek a maintenance facility for more significant maintenance. Such behaviour requires the agent to stop whatever it was doing and then navigate to a maintenance station to either refuel, recharge or seek some other type of maintenance. Once the maintenance is complete, the agent can continue whatever mission the agent was pursuing. 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 the agent to be able to drop out-of-date intentions and then readopt mission goals once the maintenance has been completed.

Listing 5.8 provides the Agent in a Box's mechanism for replenishing a depleted resource. This resource could be anything, such as requiring fuel of some sort, recharging a battery, requiring an oil change, etc. The plan is triggered by the addition of the `resource(State)` belief. This belief needs to be generated by the perception translator, based on the data from a resource monitoring sensor. In order for the agent to properly prioritize the 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 context, using the `lowResource(State)` rule, also provided in the listing, limits the plan to only be applicable when the resource has been depleted and 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 a mission running, 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 plan is a default plan which ensures that the goal of replenishing the resource is readopted while the agent works on replenishing the resource.

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 `resource(State)`,

Listing 5.8: Resource Management Plans and Rules.

```
health(resource).
health(replenishResource).
+resource(State)
    :   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).
```

and whether or not the agent is docked at the station. 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)`. Lastly, the action for docking and undocking the agent with the station, `station(dock)` and `station(undock)` is needed.

5.3.5 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 Agent in a Box and what needs to be done to use the Agent in a Box 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. 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 included perceptions of the agent's position, resource state, docking station status, and an action for docking the robot at the maintenance station.

5.4 Summary

As stated earlier, the goal of the Agent in a Box is to simplify the development of mobile robots using BDI by providing features that are common to mobile robots in a variety of domains. This section has provided the details of three main aspects of how this goal has been achieved. The Agent in a Box goes beyond the discussed limitations of the state of the art, namely, the frequently used ad hoc or hard-coded implementations. By contrast, the Agent in a Box provides a comprehensive solution, developed by identifying the general components and behaviours that are common in all mobile robotic domains. This included several main components: the connection of the agent to the environment, the appropriate prioritization of behaviour, and a behaviour framework. Also discussed was a trade study on how the agent manages its perceptions.

The first challenge for the Agent in a Box was to connect the reasoner to the sensors and actuators in a flexible way. The Agent in a Box connects the agent reasoner to the robot using ROS, a popular system in the robotics community. This provides a familiar method for application domain developers to integrate the agent's reasoner with other ROS nodes that are already available. The BDI reasoner, implemented with Jason, was connected to ROS using the SAVI ROS BDI node, which subscribes to sensor perceptions and an inbox while publishing actions and messages as needed. Also described was the needed perception and action translators and the sensor and actuator nodes, which encapsulate the sensors and actuators components.

The second component of the Agent in a Box provides reasoner support for prioritizing agent behaviour. Inspired by the Subsumption Architecture, Jason's default event and option selection functions were overridden so that it selects events based on the relative priority of the behaviour that these events trigger. This was accomplished by adding a set of beliefs to the AgentSpeak program which identify what types of behaviours are triggered by different events. This type of behaviour prioritization was missing from every surveyed work in the state of the art.

Thirdly, the Agent in a Box includes a behaviour framework which provides generic behaviours that are useful for a variety of application domains. This includes behaviour for navigation which generates paths as plans, movement, mission management, map update, obstacle avoidance, and resource management. The behaviour framework provides the needed mechanisms for the Agent in a Box to be applied to various application domains. This can be done, for example, by providing the implementation of expected achievement goals for defining the agent's mission and movement. An overview of the behaviour framework with the prioritization scheme are shown in figure 5.10.

In order to validate the Agent in a Box is has been used to control a variety of mobile robots. The implementation of these case studies, which served as the basis of qualitative and quantitative validation for the proposed framework, are discussed in the next chapter.

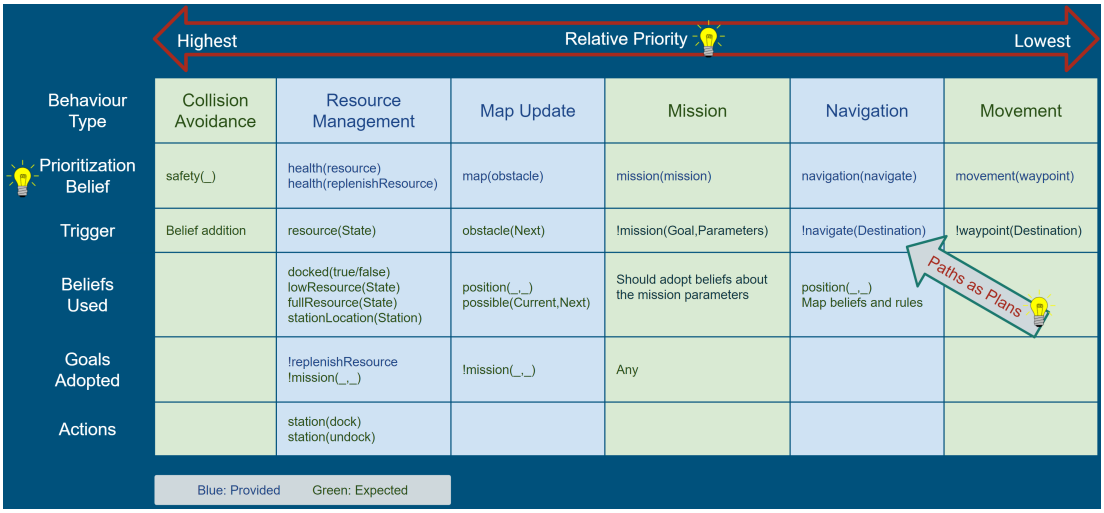


Figure 5.10: Behaviour Framework Summary.

Chapter 6

Case Studies

With the Agent in a Box established, including how it connects to the environment, how the behaviour is prioritized, and the generic behaviour defined, it is necessary to investigate how the framework can be used by mapping it to application domains. In this chapter, the details of how the Agent in a Box was used for controlling a combination of real and simulated mobile robots in three different environments are provided. The domains, listed in table 6.1, were chosen to investigate how the agent works in different environments so that the results observed in the experiments were not tied to the properties of any specific environment. These case studies were also chosen to exercise all the different aspects of the Agent in a Box's generic behaviour plans. Additionally, the case studies needed to 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. Provided for each case study is an outline of the environment, the sensors and actuators that are available to the agent, the agent's mission parameters, and how the framework is mapped to the domain.

Table 6.1: Case Studies Use of Agent in a Box.

Agent in a Box Feature	Case Study		
	Grid	Autonomous Car	Mail Delivery Robot
Navigation	✓	✓	✓
Custom Map	✓	✓	✓
Movement	✓	✓	✓
Map Update	✓	✗	✗
Collisoin Avoidance	✓	✓	✓
Resource Management	✓	✗	✓

The goal of the first cases study was to get a simple introduction to how the Agent in a Box can be used without getting distracted by complexity of the environment. For this reason a grid-based domain, discussed in section 6.1 was used. To investigate how the Agent in a Box can be

used to control a more realistic vehicle in a more realistic environment, a simulated autonomous car was driven through a neighbourhood environment. The details specific to that case study are discussed in section 6.2. Finally, to investigate the use of the Agent in a Box with real robotic hardware, providing an opportunity to see how it performed on an embedded computer, a prototype mail delivery robot was used for the third case study. This robot was built using the iRobot Create hardware – a Roomba vacuum cleaning robot without the vacuum, and is discussed in section 6.3.

6.1 Grid Environment

To investigate the the basic principles of the Agent in a Box a grid environment was used, as shown in figure 6.1. In this environment, the agent was situated on the grid and given the task of moving to a destination. As stated previously, this provided the opportunity to explore all the aspects of the Agent in a Box without added complexity from the environment. This means that this *should* be the simplest case study environment for the Agent in a Box.

The agent was given a map showing the possible paths between the locations on the map. In order to investigate the map update behaviour, the the map did not include the locations of all of the obstacles. This meant that the agent would need to update its map as it moved through the environment. The agent was also equipped with a simulated battery which reduced in charge over time, forcing the agent to manage its battery resource. The agent would need to recharge the battery at a charging station located on the map. To test the obstacle avoidance behaviour there were pedestrians on the map, obstacles which would block the agent's path until the agent honked a horn at them signalling them to move out of the way. The agent was not given any warning that these pedestrians would be present. The agent's perceptions of the environment included 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.

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 using the Agent in a Box connection scheme discussed in the approach chapter and shown in figure 5.2. 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 agent is available on GitHub [83,84]. The architecture for this environment is shown in figure 6.2.

With the connection to the environment handled, the grid agent needed to be customized to

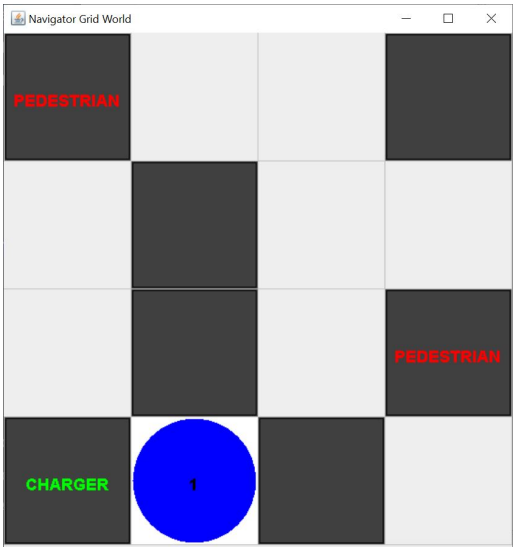


Figure 6.1: Grid Environment.

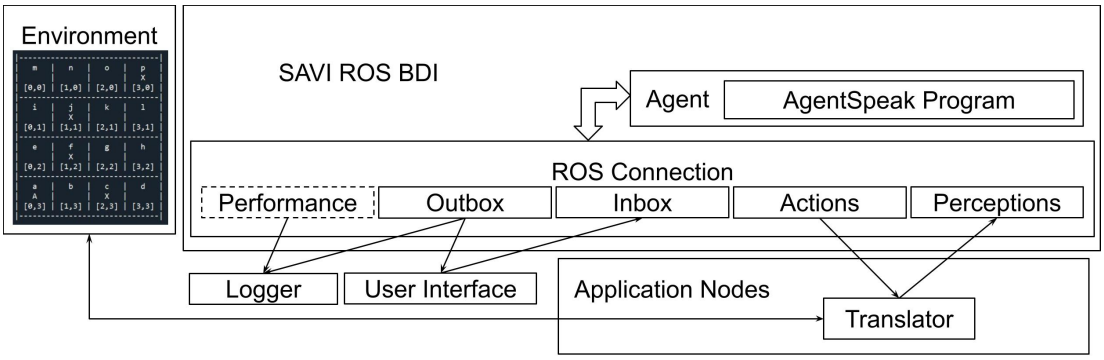


Figure 6.2: Grid Environment Architecture.

its domain. Let us first discuss the domain-specific development that was needed to map the Agent in a Box to this domain, starting with the agent's navigation. The navigation behaviour provided by the Agent in a Box requires a map written in AgentSpeak as well as successor state predicates and a heuristic predicate. An excerpt of the map is provided in listing 6.1, specifying the coordinates of each grid location as well as the possible paths between these locations.

A sample successor state predicate, which uses a supporting rule, and the heuristic predicate are provided in listing 6.2. The successor state predicate provides the cost to the agent for moving to an adjacent grid square. The example shown in listing 6.2 provides the rule definition for the successor state predicate for grid squares in the `up` direction. This uses a rule for getting the

Listing 6.1: Partial Grid Agent Map.

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

Listing 6.2: 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]).
```

location coordinates from the belief base. Last in listing 6.2 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 used by the navigation plan.

With the map defined, the agent needs plans for achieving the `!waypoint(NextPosition)` goal. The plans triggered by this goal, adopted by the Agent in a Box frameworks's navigation plans, are responsible for moving the agent to the specified locations along the generated route using the domain-specific actions available to the agent. The grid agent's movement was designed as shown in listing 6.3. The first line in the listing provides the behaviour type for the reasoner to properly prioritize the `!waypoint(_)` event as a triggering event for movement behaviour. This is followed by the plan for moving the agent. The agent perceives its position with the `position(X,Y)` predicate and uses 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. It should be noted 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.

With the robot able to navigate and move through the environment, the last needed piece of the implementation is to provide the pedestrian avoidance behaviour. The Agent in a Box provides the mechanism for prioritizing safety-related behaviour. The developer does this by providing the agent a belief that tells the reasoner that safety behaviour is triggered by `pedestrian`, a belief that the agent perceives when a pedestrian is in an adjacent location. The desired avoidance action in this environment is for the agent to honk a horn to signal them to get out of the way. This can be easily implemented as a belief-triggered plan, providing the agent reactive behaviour for avoiding pedestrians.

The implementation of the pedestrian avoidance behaviour is provided in listing 6.4. First,

Listing 6.3: 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]).
```

Listing 6.4: Grid Agent Obstacle Avoidance.

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

the behaviour belief, identifying to the reasoner that a safety related behaviour will be triggered by pedestrian events, is provided. Next, the plan name is specified as `pedestrianAvoidance` and that it must be executed atomically, meaning that it must be run to completion, without interruption. Lastly, 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 prioritization method will select this plan as the highest priority.

A video of the grid behaviour is available on YouTube [85]. With the grid agent complete, it is time to apply the Agent in a Box to a more realistic environment: a simulated autonomous car.

6.2 Simulated Autonomous Car Environment

Although the grid agent provided an opportunity to demonstrate all aspects 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. To test this, this case study investigates the use of the Agent in a Box for controlling a simulated autonomous car.

AirSim is a high fidelity simulator maintained by Microsoft and available open source [86,87]. 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 [88]. The development focus was to

drive the car in AirSim's neighbourhood environment, shown in figure 6.3. This environment provides an enclosed suburban neighbourhood, complete with roads and parked cars that need to be avoided. There are also 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. Our software for the agent in this environment is available on GitHub [89].

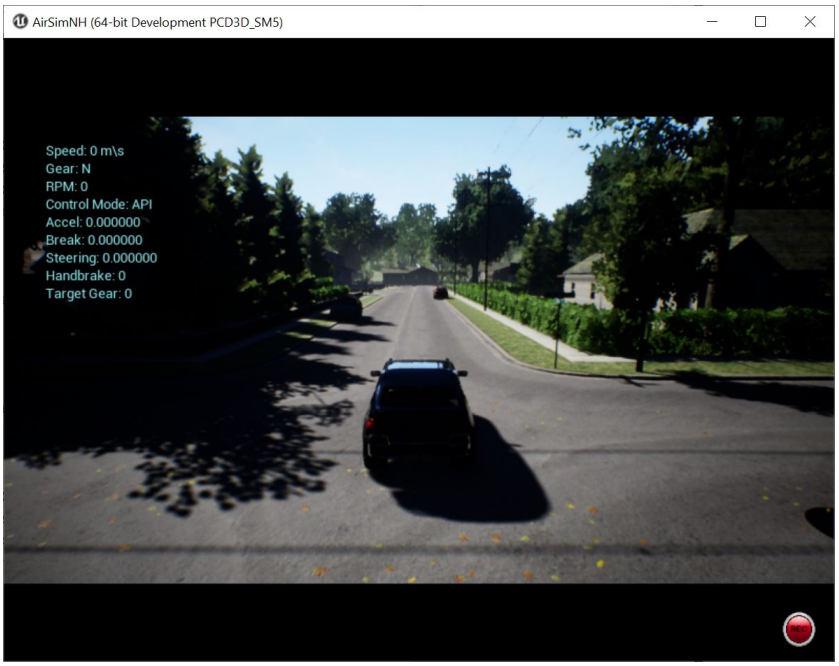


Figure 6.3: AirSim Neighbourhood Environment.

The architecture of the car agent using the Agent in a Box, is shown in figure 6.4. For this environment the agent needed access to a variety of different sensors and actuators. These included a GPS sensor, a magnetometer, a speedometer, a LIDAR, and a camera. To connect each of these sensors, they were 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 following the road. The topics published by these sensor nodes were subscribed to by the perception translator, which assembled the data into predicates and published it 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

Listing 6.5: Excerpt of the AirSim Car Map Definition.

```
locationName(post1,[47.6414824,-122.1403650]).
locationName(post2,[47.6426243,-122.1403545]).
possible(post1,post2).
```

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.

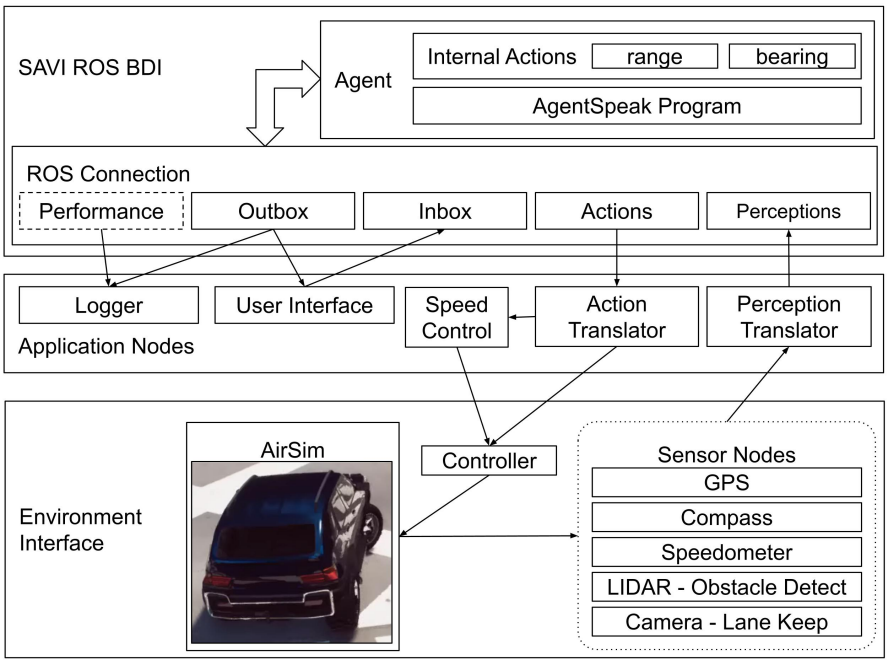


Figure 6.4: AirSim Autonomous Car Architecture.

The Agent in a Box provided the car agent the mission behaviour for its navigation mission. For the framework-provided navigation plans to work, the car needed a map of latitude and longitude coordinates, plans for achieving the `waypoint` goal by controlling the car's steering and speed, and collision avoidance behaviour for avoiding parked cars.

An excerpt from the agent's map definition, using latitude and longitude, is provided in listing 6.5. The map defines the locations of interest in terms of their latitude and longitude with seven decimal places to enable sub-meter measurement precision. The successor state rule and the heuristic rule needed to be specialized to work with latitude and longitude, as shown in listing 6.6. The successor state rule was simplified, only specifying that it was possible for the car to drive between the specified locations. The heuristic required an internal action for calculating the range between the coordinate locations.

Listing 6.6: 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).
```

Listing 6.7: AirSim Car Obstacle Avoidance.

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

The agent had 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 6.7. First, the behaviour triggered by the `obstacle` was identified as being for safety behaviour, signalling to the Agent in a Box's behaviour prioritization functions that this event trigger must be prioritized as being safety related. This means that the reasoner will select this plan, when it is applicable, as an intention over any other plans that may be applicable to the agent's context. Next, the plan is 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 until the distance to the obstacle is less than 5 m.

With navigation and obstacle avoidance handled, it is time to provide plans for driving the car. These are provided in listing 6.8. The first plans in this listing are for the `!waypoint(_)` achievement goal. These plans 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. Second – the plan responsible for slowing the car as it approaches a location is provided. 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 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

Listing 6.8: AirSim Car Driving Plans.

```
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).
```

Listing 6.9: 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).
```

keeps the car driving toward the location using recursion.

These plans are supported by a set of rules shown in listing 6.9, 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 6.10 provides the implementation of the speed controlling behaviours. First, the mental note of the speed setting for when the agent starts and the car is not moving is provided. This mental note allows the agent to remember the speed setting so that the cruise controller does not need to be reset needlessly. Also provided is the behaviour belief, identifying 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. There is also a plan for setting a speed setting in the

Listing 6.10: AirSim Car Speed Control.

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

event that a speed setting 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 the listing is the default plan, which is only applicable if the speed setting does not need to be changed.

Listing 6.11 provides the rules and plans for controlling the car's steering. Listing 6.11 begins with 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 the listing. The listing begins by providing a behaviour belief identifying that these plans are movement related, enabling the reasoner to properly prioritize this behaviour. The first 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.

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 validation of these behaviours is discussed in the validation chapter in section 7.1. The car agent made good use of the Agent in a Box, which enabled the development of the sensor and

Listing 6.11: AirSim Car Steering Control.

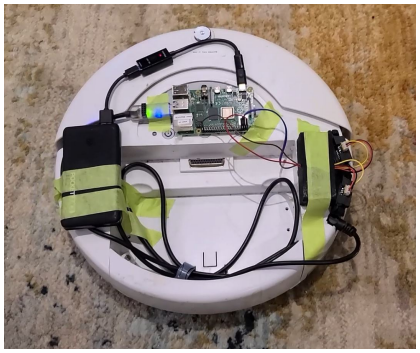
```
declination(7.5).
courseCorrection(TargetBearing, Correction)
    :- compass(CurrentBearing)
       & declanation(Declanation)
       & (Correction = TargetBearing - (CurrentBearing + Declanation)).
steeringSetting(TargetBearing, 1)
    :- courseCorrection(TargetBearing, Correction)
       & (Correction >= 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(_,_).
```

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 [90] along with another video focusing on the lane-keep and collision avoidance behaviour [91].

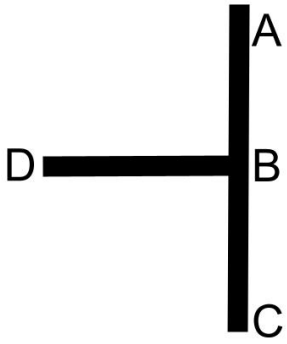
6.3 Mail Delivery Agent

Both of the case studies discussed so far have involved simulation on a desktop computer. If the Agent in a Box is to be useful for controlling autonomous robots in the real world there needs to be a case study which uses the Agent in a Box on a real robot using an embedded computer. With increasing reliance on mail delivery and logistics, especially true during the COVID-19 pandemic, the mail delivery domain was explored. In this case, the mail delivery robot is a developmental prototype for delivering mail between points of interest in the Carleton University tunnel system. This provides the robot with an indoor space which connects to almost every building on campus with no weather to deal with and smooth floors to drive on. Although these are attractive features of the tunnel system, there are some drawbacks. First, the tunnels do not have consistent wireless internet coverage, although there are locations where there is reliable network access. The tunnels also have lower lighting levels than typical office environments, providing a potential challenge to the design. Finally, in the tunnel there is no access to Global Navigation Satellite System (GNSS) signals, such as GPS, meaning that the robot will need to determine its location through another method. At this stage, development and testing efforts have focused on the use of an analogue environment and the testing of the agent reasoning system.

The current implementation of the mail delivery robot uses an iRobot Create, shown in figure 6.5a, which is the development version of the Roomba vacuum cleaning robot, without the vacuum-cleaning components. This robot can be controlled using a command protocol over a serial interface [92]. A Raspberry Pi 4 computer was attached to the robot and connected via a serial cable, providing an embedded computer for the Agent in a Box to run on. The map used for the prototype development is shown in figure 6.5b, where the letters in the figure represent locations that the robot can visit and the lines represent possible paths between these locations.



(a) Mail Delivery Robot.



(b) Mail Delivery Map.

Figure 6.5: Mail Delivery Robot and Map.

The architecture of the mail delivery robot nodes is shown in figure 6.6. The robot was controlled using the `create_autonomy` node [93], which acts as a connector for the robot to ROS, providing a variety of publishers of the robots various sensors, and subscribers 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 was provided to the reasoner in the form of perception predicates, formatted by the perception translator. The action translator generates 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 [94].

The mission for the agent controlling the mail robot is provided in listing 6.12. 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

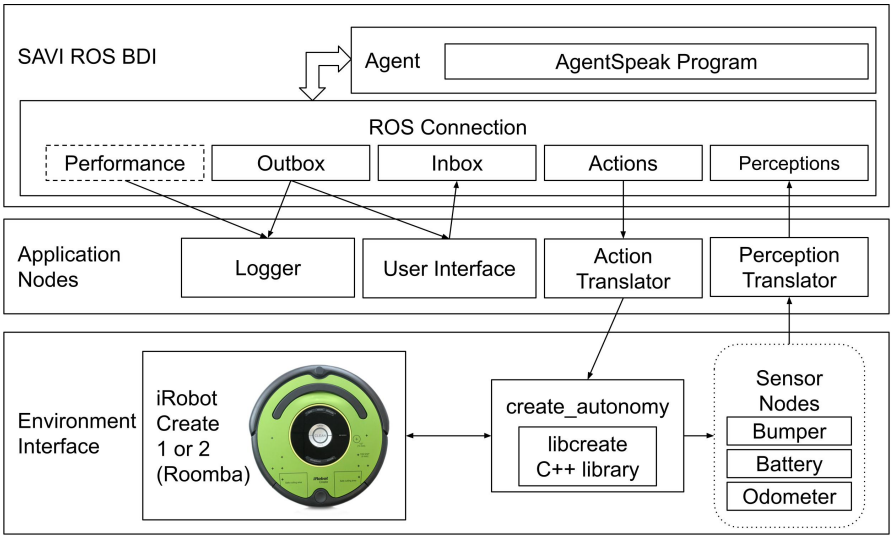


Figure 6.6: Mail Delivery Robot Architecture.

Listing 6.12: Mail Robot Mission Definition.

```
1  mission(mission).
2  +!mission(Goal,Parameters)
3      :   Goal = 'collectAndDeliverMail'
4          & Parameters = [Sender,Receiver]
5      <-  +mission(Goal,Parameters);
6          !collectAndDeliverMail(Sender,Receiver);
7          -mission(Goal,Parameters).
8  mission(collectAndDeliverMail).
9  +!collectAndDeliverMail(Sender,Receiver)
10     <-  !collectMail(Sender);
11         !deliverMail(Receiver).
12 mission(collectMail).
13 +!collectMail(Sender)
14     :   not haveMail
15     <-  !navigate(Sender);
16         +haveMail.
17 +!collectMail(Sender).
18 mission(deliverMail).
19 +!deliverMail(Receiver)
20     :   haveMail
21     <-  !navigate(Receiver);
22         -haveMail.
23 +!deliverMail(Receiver).
```

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.

The mail robot's map 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 6.1. 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

Listing 6.13: 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).
```

Listing 6.14: 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).
```

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 6.13.

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 using a route obtained with A* search. The plans that achieve this goal, provided in listing 6.14, move the agent between the locations using the domain-specific `!waypoint(_)` goal. 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.

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 6.15. The listing starts by providing 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. The `!faceNext(_)` goal uses the `direction(_)` belief, which is the direction the robot is currently facing, and the `directionToNext(_)` rule, which provides the direction that the agent needs to face to move to the next destination. A sample of the `directionToNext(_)` 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 [94]. The `!faceNext(_)` triggered plan which also uses a rule to

Listing 6.15: 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).
```

Listing 6.16: 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).
```

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.

There are two subgoals used for actually moving the robot: `!turn(_)` and `!drive(_)`. The implementations of these are provided in listing 6.16. 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 the `!move(_,_,N)` goal is implemented using recursion, using the movement action to control the robot and repeating the goal with a reduced counter after each iteration.

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 prioritizing it over the agent's

Listing 6.17: Mail Robot Obstacle Avoidance.

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

other activities. The obstacle avoidance for this agent is provided in listing 6.17. The obstacle avoidance is a reactive behaviour implemented using a belief-triggered plan. This is triggered when the robot's bumper sensor is activated. The resulting actions are to turn the robot to the left and attempt to 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.

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 [95–97].

6.4 Summary

To demonstrate the use of the Agent in a Box for controlling autonomous mobile robots, it was applied to various robots in different domains through case studies. These case studies were selected to investigate how the Agent in a Box could be used and also to expose the different aspects of the Agent in a Box. The first of these environments, a grid-based environment, was selected to test the Agent in a Box without additional complexity from the environment. The Agent in a Box also needed to be investigated in a more realistic environment using more realistic sensors and actuators. To accomplish this, the Agent in a Box was used for controlling an autonomous car simulated using AirSim. Lastly, to investigate the Agent in a Box on real hardware with a real robot, the final case study was to have the Agent in a Box control a prototype mail delivery robot. Although each of these domains provide the agent with different sensors and actuators, and have different mission demands, the Agent in a Box was was successfully used for developing the agent needed for controlling these robots and completing their missions, while avoiding collisions and maintaining the robot's resources as needed.

Detailed in this chapter were the connections between each agent and their respective robot's sensors and actuators. Also discussed was how the behaviour framework was applied to each

domain. This included the details of the needed plans, rules, and beliefs that were necessary to complete the implementation of the behaviours for each agent. Each of these case studies were evaluated in their respective environments, allowing for the validation the Agent in a Box for controlling autonomous mobile robots. The results of these case studies are provided in the next chapter.

Chapter 7

Validation

This chapter provides the experimental results from validation experiments conducted with robotic agents implemented with the Agent in a Box. The validation demonstrated that the Agent in a Box provides useful properties to robotic systems and assess the burden of using it in terms of runtime performance and software engineering properties. There are three main themes to the validation experiments discussed in this chapter, outlined in table 7.1, and the next few paragraphs.

Table 7.1: Overview of Validation Experiments

Theme	Experiment	Result Metric/ Success Criteria	Section
Agent Behaviour	Mission Behaviour	Qualitative – behaviour in runtime logs	
	Grid Agent		7.1.1.1
	Simulated Autonomous Car		7.1.1.2
	Mail Delivery Robot		7.1.1.3
	Reactive Behaviour		
	Map Update		7.1.2.1
	Collision Avoidance		7.1.2.2
	Resource Management		7.1.2.3
Runtime Performance	Decoupled Reasoning Cycle	Reasoning cycle not coupled to environment refresh	7.2.1
	Responsiveness	Response time for collision avoidance compared to alternatives	7.2.2
	Profiling	Reasoning time relative to perception and action time	7.2.3
	Navigation sub-framework	Route generation time compared to alternatives	7.2.4
Software Engineering Properties	Assessment of development burden	Coupling, Cohesion, and Cyclomatic Complexity	7.3

The first set of validation experiments focused on the properties of the Agent in a Box's behaviour framework and the behaviour prioritization added to the reasoner. Discussed in section 7.1, these results came from runtime logs generated by the case study agents discussed

in the previous chapter. These experiments demonstrated how the agents were all able to complete domain-specific missions, navigate to destinations as part of those missions, handle collisions, and maintain resources. The experiment results highlighted how these missions could be implemented and executed in a goal-directed manner without concern for collision avoidance and resource management, while still having the confidence that these concerns (implemented with reactive behaviour) would be properly handled by the Agent in a Box.

The second set of validation experiments focused on the performance of the Agent in a Box. These results are provided in section 7.2. The focus of these experiments was to assess if there was a performance bottleneck resulting from the use of the Agent in a Box for controlling mobile robots. The first of these experiments came from the SAVI project, which demonstrated the feasibility of decoupling the reasoning cycle from a simulator's environment update. The SAVI architecture formed the basis of how the Agent in a Box connects the agent to ROS. The second experiment compared the performance of the Agent in a Box several alternatives, specifically comparing the reaction times of the different agents in an obstacle avoidance scenario. Another experiment provided profile measurements which focused on assessing if the agent's reasoning imposed a performance bottleneck. Also examined was the performance of the navigation framework, where the performance of BDI agents, were compared using algorithms implemented in three different programming languages. This assessed if there was a performance cost to using AgentSpeak to implement aspects of the Agent in a Box's navigation framework. This assessment observed the agent's reasoning cycle, identifying if the agent was more or less responsive based on which implementation was used.

The third focus of the validation was an assessment of the software engineering properties of the Agent in a Box. Detailed in section 7.3, this experiment examined the implementations of the car agents used in the stop trial performance tests to examine the experience a developer might have when working with the Agent in a Box. This assessment focused on assessing the coupling, cohesion, and cyclomatic complexity of the different agents.

In addition to the validation experiments discussed in this chapter, Appendix A examines what parallels exist between the development of BDI agents with Jason and the development of Finite State Machines (FSMs). The goal was to see if any parallels could provide guidance for designing agents using BDI which have favourable software properties.

7.1 Agent Behaviour

The focus of this section is on the validation of the Agent in a Box behaviour framework and behaviour prioritization. This validation was accomplished by both observing the behaviour of

the agents implemented in the case studies discussed in the previous chapter, and examining the logs generated by these agents. How the Agent in a Box's generic behaviours were used successfully to complete the missions in each of the case studies is a highlight of this section. Also shown is 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 7.1.1. This section highlights how the agents missions were achieved using a combination of generic and domain-specific software. The generic code provided by the Agent in a Box included the navigation plans which generated the robot's path to its destination as a plan plan for the agent to run, monitor, 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 which is discussed in section 7.1.2. This includes how the agents handled unexpected situations, such as incorrect maps, collision avoidance, and resource management. The handling of 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 behaviour plans interrupted whatever else the agent may have been doing and then allowed the agent to return to whatever mission was underway. The interrupting plans needed to do this without any specific knowledge of what the interrupted mission may have been.

7.1.1 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. The routes took the form of plans: sequences of waypoint achievement goals for the agent to follow. The first case study discussed is the grid agent in section 7.1.1.1, followed by the autonomous car in section 7.1.1.2, and the mail

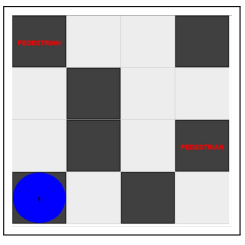
robot in section 7.1.1.3. Each of these agents used generic mission level plans which could be interrupted, suspended, and resumed as needed, but without concern for what the interruptions may be. The interruptions are discussed in section 7.1.2.

7.1.1.1 Grid Agent Mission

The first scenario considered was the simplest: the grid agent. This scenario provided the opportunity to test the Agent in a Box without much concern for complexity from the environment. The grid agent started its run at the location in the bottom left corner of the map, called location `a`, and given the task of navigating to the grid square at the bottom right, location `d`. This was a navigation mission that was provided by the Agent in a Box. This meant that for the agent to achieve the mission, all that was needed to be provided for the Agent in a Box to begin its mission was a map of the environment and the domain-specific plans for achieving the `waypoint(_)` goal. The following paragraph and figure show how the agent, using the Agent in a Box, generated a route to the destination and began moving to its destination.

At the start of the grid agent's run we expect to see the agent perceive its position at the start location and then adopt the generic mission of navigating to location `d`, defined in listing 5.5. This involved getting a route around the obstacles on the map using the plan in listing 5.2 and then moving along that route to the destination using the plans in listing 6.3.

The start of the grid agent's run is shown in figure 7.1. The first line of the run provides the agent perception, followed by several lines of the agent deliberating internally with respect to navigating to the destination. The agent started by setting a mental note about the nature of the mission, in case of interruption later. The Agent in a Box's generic navigation plan calculated a route to the destination, which was loaded as a sequence of achievement goals. Then the agent began working to achieve the first waypoint goal. The agent sought to achieve the movement to location `e`, in the square above its starting location. At this point, the generic navigation plan used the domain-specific implementation of the `waypoint(_)` goal, which resulted in the agent taking the `move(up)` action.

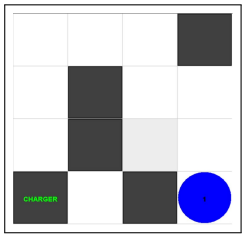


```
1 Perception: [position(0,3), obstacle(down), obstacle(left),
2     map(up), map(right), battery(9), charging(false)]
3 Agent: +missionTo(d)
4 Agent: gettingRoute(a,d)
5 Agent: route([op(up,e),op(up,i),op(up,m),op(right,n),
6     op(right,o),op(down,k),op(right,l),op(down,h),
7     op(down,d)]))
8 Agent: waypoint(move(e))
9 Action: move(up)
```

Figure 7.1: Grid Agent Run: Getting Route.

The Agent in a Box continued achieving its waypoint goals until the agent reached its destination, as shown in figure 7.2. At this point, the agent dropped the mission level mental note

as it had successfully completed its mission.




```
1 Perception: [position(3,3), obstacle(down), obstacle(left),
2     obstacle(right), map(up), map(left), battery(9),
3     charging(false)]
4 Agent: arrived(d)
5 Agent: -missionTo(_)
```

Figure 7.2: Grid Agent Run: Arrived at the Destination.

The Agent in a Box provided a useful means for the grid agent to generate a route to a destination using its generic navigation plan. The generic navigation plans used the domain-specific map and movement plan, which implemented the `waypoint(_)` achievement goal. 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 7.1.2.

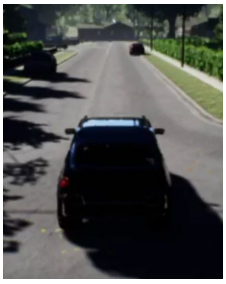
The Agent in a Box was successful at moving the grid agent through its mission using its generic plans. Although this result is encouraging, it is a relatively simple test scenario. To further validate that the Agent in a Box is useful for controlling mobile robots a more realistic scenario was needed. The next section outlines the validation of the Agent in a Box for driving a simulated autonomous car.

7.1.1.2 Autonomous Car Mission

The previous section provides confidence that the Agent in a Box's generic route generation plan could be used to control a relatively simple grid agent. This section demonstrates that the same plans in the Agent in a Box were also successful in a more complicated environment by driving simulated autonomous car. In this case the agent needed to first perceive its position, compass direction, and speed. Then, the agent needed to generate a route to its destination from its current location based on its perceived position using the plan in listing 5.2. Once the route plan had been generated, the agent could begin driving using its `waypoint(_)` achievement goals, defined in listing 6.8, to drive to the different waypoints on its route. This involved adopting achievement goals for both steering the car and controlling its speed. The goals for speed control and steering are provided in listings 6.10 and 6.11.

Images and logs that focus on different aspects of the Agent in a Box's navigation behaviour for the car are provided in figures 7.3, 7.4, and 7.5. In this case study, the agent was provided the car's speed, compass bearing, and GPS coordinates. The agent controlled the car by setting a steering setting and specifying the desired speed. The agent began its run with the goal of

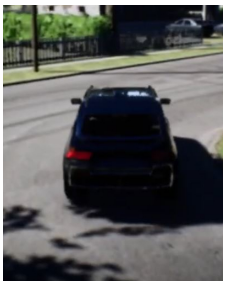
moving to `post3`. Moving to this location required the car to drive to the end of the street, turn right, then drive to the end of the next street, and then stop. The Agent in a Box began by generating a route, using the domain-specific map, and then adopting a sequence of `waypoint(_)` achievement goals. The Agent in a Box then called the domain-specific implementations of the `waypoint(_)` goal to drive the car down the street. This began with the car setting its desired speed.



```
Perception: gps(47.641482371260814, -122.14036499101078)
    compass(-7.32937565846)
    speed(0.0)
Agent:  navigate(gettingRoute(post1,post3))
Agent:  route([op(initial,post1),op(drive,post2),
    op(drive,post3)]
Agent:  driveToward(post2)
Agent:  steer(steering(0.005834349870812172),
    bearing(0.3540858823961912))
Action: steering(0.005834349870812172)
Agent:  controlSpeed(speedSetting(8),oldSpeedSetting(0))
Action: setSpeed(8)
```

Figure 7.3: Autonomous Car Navigation: Getting Route.

As the car approached the waypoint at the end of the first street it slowed and then stopped, as specified in the domain-specific plans for achieving the `waypoint(_)` goal, defined in listings 6.8 and 6.9. The Agent in a Box then adopted the next achievement goal to drive to the next location. Once again, the domain-specific plans for achieving the `waypoint(_)` began generating actions for driving the car. This time, the car's steering was adjusted to turn the car to the right and the car was commanded to start driving.



Near Turn:

```
Perception: gps(47.6424528723379, -122.14035419820956)
    compass(-6.77715559995)
    speed(7.26216220856)
Agent:  controlSpeed(speedSetting(3),oldSpeedSetting(8))
Agent:  setSpeed(3)
```

At Turn:

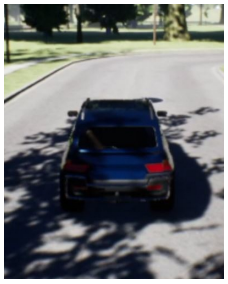
```
Perception: gps(47.64256169095842, -122.14035388536924)
    compass(-7.24600712745)
    speed(2.84764552116)
Agent:  driveToward(arrived,post2)
Agent:  controlSpeed(speedSetting(0),oldSpeedSetting(3))
Action: setSpeed(0)
```

Turn:

```
Agent:  driveToward(post3)
Agent:  steering(1)
Action: steering(1)
Agent:  controlSpeed(speedSetting(8),oldSpeedSetting(0))
Action: setSpeed(8)
```

Figure 7.4: Autonomous Car Navigation: Turning the Corner.

As the car approached the destination, it again slowed before coming to a stop. Upon stopping, the Agent in a Box had demonstrated its ability to successfully drive the car to its destination.



Near Destination:

```
Perception: gps(47.642633645055454, -122.13916613536006)
    compass(82.8087568444)
    speed(7.45147848129)
Agent:  driveToward(near,post3)
Agent:  controlSpeed(speedSetting(3),oldSpeedSetting(8))
Action: setSpeed(3)
```

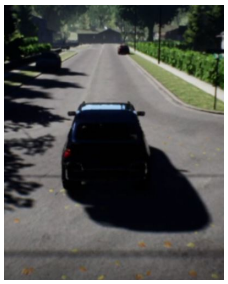
At Destination:

```
Perception: gps(47.64263260053664, -122.13901366745064)
    compass(82.8899635556)
    speed(2.92947816849)
Agent:  driveToward(arrived,post3)
Agent:  controlSpeed(speedSetting(0),oldSpeedSetting(3))
Action: setSpeed(0)
Agent:  navigate(arrived(post3))
```

Figure 7.5: Autonomous Car Navigation: Arriving at the Destination.

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. A second test was performed to expose how the Agent in a Box works with more complex sensor inputs. This was completed by adding 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.

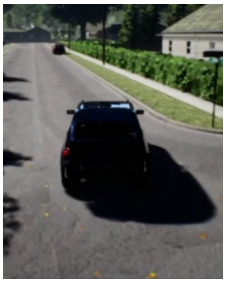
Images and logs of lane-keep behaviour are provided in figures 7.6, 7.7, and 7.8. To start, the car was seen at the scenario's starting location, in the middle of an intersection. As before, the Agent in a Box generated a route to the destination and started to drive using its compass for navigation as the lane had not yet been detected.



```
Perception: gps(47.641482371456824, -122.14036499104333)
    compass(-7.88536198326) speed(-0.000102847225207)
    lane(0.0,0.0,0.0,0.0,0.0) obstacle(56.546998024)
Agent:  compassSteer(0.007829464965595702)
Action: steering(0.007829464965595702)
```

Figure 7.6: Autonomous Car Lane-Keep: Start.

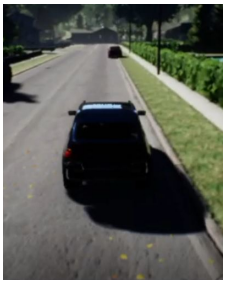
As the car continued to drive, the lane was detected. At this point, the car used the recommended lane-keep steering setting and turned toward the side of the road to line up with the lane, as provided in listing 6.11.



```
Perception: gps(47.641495174820086, -122.14036475278931)
    compass(-1.74476203377) speed(2.54457998276)
    lane(0.5105520651261017,0.0,0.0,0.3062200956937799,
    92.92822966507185)
    obstacle(63.1855273247)
Agent:  lkaSteer(0.5105520651261017)
Action: steering(0.5105520651261017)
```

Figure 7.7: Autonomous Car Lane-Keep: Moving to the Lane.

Once the car had lined up with the lane, the car used the lane-keep recommendations for steering the car.



```
Perception: gps(47.64150653149476, -122.14035987604326)
    compass(11.0515023328) speed(2.18603897095)
    lane(-0.059311016769297856,0.0,0.0,0.38065239086305397,
    97.94559447466612)
    obstacle(34.3281619549)
Agent:  lkaSteer(-0.059311016769297856)
Action: steering(-0.059311016769297856)
```

Figure 7.8: Autonomous Car Lane-Keep: Using lane-keep.

This section has highlighted how, as with the grid agent, the Agent in a Box's generic navigation plans were successful for controlling the car. All that was needed to be provided was a domain-specific map and domain-specific plans for driving the car between the waypoints, using the available sensors and actuators. 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 7.1.2.2.

So far, the Agent in a Box has controlled both a grid agent and a car agent by navigating them both to their destinations using its generic behaviour. That said, in both of these cases, the Agent in a Box was using the generic navigation mission that it provided. 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. The next section provides an example where the Agent in a Box was used to control a prototype mail delivery robot.

7.1.1.3 Mail Delivery Mission

A mail delivery robot was used for demonstrating both the use of the Agent in a Box on a real robot with an embedded computer and also a custom mission. 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 was a domain-specific mission which used the Agent in a

Box's navigation plans, defined in listing 5.2. For this to work, the developer provided plans for implementing the mission (listing 6.12), the map (listing 6.13), and waypoint goals (listings 6.14, 6.15, and 6.16) needed by the Agent in a Box. The implementation of the mail agent, using the Agent in a Box, was discussed in section 6.3. The mission began with the robot receiving a mail mission, which commanded the agent to collect mail from a location and deliver it to another. The agent should then have begun this mission by collecting the mail by using the Agent in a Box's navigation plans to generate a route to the sender's location. The agent should then follow this route using the `waypoint(_)` triggered plans provided for moving the robot. It should then collect the mail and adopt a belief that this portion of the mission was completed. The agent should then begin the delivery portion of its mission by first generating a route to the destination and then following the route, again using its `waypoint(_)` triggered plans. Once at the destination the agent should deliver the mail, completing the mission.

The following three figures provide logs for the agent at different stages of its mission. The robot started at location `D` and was assigned a mail mission where it needed to visit location `A` to collect mail and then deliver it to location `C`. Figure 7.9 provides the logs for the Agent in a Box as it received its mail mission tasking. To complete this mission the agent needed to collect mail by moving to the sender's location and then deliver mail by moving to the receiver's location. The Agent in a Box began achieving the mail collection goal by generating a route to location `A`, the sender's location, as as a sequence of `waypoint(_)` goals. The robot then began to move using the domain-specific `waypoint(_)` implementation.


```
1  Inbox:  mission(collectAndDeliverMail,[a,c])
2  Agent:  mailMission(a,c)
3          collectMail(goToSender,a)
4          navigate(gettingRoute(a))
5          navigate(current(d))
6          navigate(route([op(initial,d),op(drive,b),
7              op(drive,a)],2),a,0)
8          waypoint(b)
9          faceNext(b)
10         drive(forwad)
11 Action: move(0.5,0,5)
```

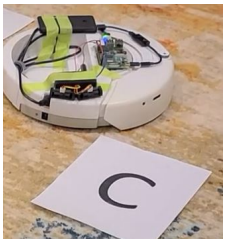
Figure 7.9: Mail Agent Mission: Starting.

Once the robot arrived at location `A` it completed the mail collection part of its mission. The robot then moved to location `C` to deliver it. Figure 7.10 provides a log of the reasoning as the Agent in a Box generated a new route, this time to the receiver, and began to move.

```
Agent:  waypoint(atLocation(a))
        navigate(arrived(a))
        collectMail(atSender,a)
        collectMail(haveMail,a)
        mailUpdate(gotMail,a,c)
        deliverMail(goToReceiver,c)
        navigate(gettingRoute(c))
        navigate(current(a))
        navigate(route([op(initial,a),op(drive,b),
            op(drive,c)],2),c,0)
        waypoint(b)
        faceNext(b)
        turn(around)
Action: move(0,4.1,4)
```

Figure 7.10: Mail Agent Mission: Collect Mail.

Once the robot arrived at the receiver's location, the mission was complete. A log of this portion of the mission is provided in Figure 7.11.




```
Agent:  waypoint(atLocation(c))
        navigate(arrived(c,0))
        deliverMail(arrivedAtReceiver,c)
        deliverMail(delivered,c)
        mailUpdate(delivered,c)>
```

Figure 7.11: Mail Agent Mission: Deliver Mail.

7.1.1.4 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 plan consisting of 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 involved more than simply moving to some destination, 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' 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.

7.1.2 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

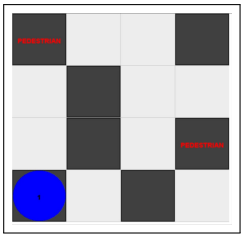
a prototype mail delivery robot. As before, this provided confidence that the Agent in a Box can successfully control a variety of mobile robots in a variety of environments. Although this is encouraging, these tests focused purely on how the Agent in a Box used goal-directed reasoning, using plans that 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 should able to successfully handle these types of interruptions. This section provides a validation of this claim.

First, in section 7.1.2.1, 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 route achievement goals given its updated context in order to complete the mission. Next, in section 7.1.2.2, 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. Lastly, in section 7.1.2.3, 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.

7.1.2.1 Map Update

To test the map update behaviour the Agent in a Box was provided an inaccurate map of the grid environment. This scenario was similar to the scenario discussed in section 7.1.1.1. 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 should be able to detect the inaccuracy in the map, update the map (using the generic map update plan provided in listing 5.7), generate a new route, and move to the destination successfully.

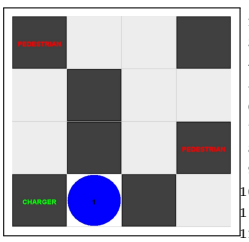
The starting state and reasoning log for this scenario are provided in figure 7.12. The first line of the run shows that the agent perceiving the environment followed by several lines of the Agent in a Box deliberating internally with respect to navigating to the destination. The reasoner started by setting a mental note about the nature of the mission in case of interruption later. The agent calculated a route, which involved moving across the bottom of the map, as a set of waypoint achievement goals. This was the expected route, taking it across the bottom of the map through the obstacle that was not on its map. The Agent in a Box began working to achieve the first waypoint goal by moving to the right.




```
1 Perception: [position(0,3), obstacle(down), obstacle(left),
2     map(up), map(right), battery(9), charging(false)]
3 Agent: +missionTo(d)
4 Agent: gettingRoute(a,d)
5 Agent: route([op(right,b),op(right,c),op(right,d)]))
6 Agent: waypoint(move(b))
7 Action: move(right)
```

Figure 7.12: Grid Agent Run: First Navigation Attempt.

An image and the reasoning log for when the obstacle was encountered are provided in figure 7.13. In this case, the Agent in a Box perceived `obstacle(right)`, which triggered the map update behaviour discussed in the approach chapter, specifically in Listing 5.7. The obstacle perception contradicted a map belief that let the Agent in a Box to believe that it was possible to move between locations `b` and `c` when it was not actually the case, making this plan applicable to the agent's current context. However, as the domain-specific waypoint plans were also still applicable, the reasoner prioritized the obstacle plan over the waypoint plan, so that the Agent in a Box did not run into the obstacle. The Agent in a Box then dropped the inaccurate map belief (which had shown that it was possible to move from location `b` to location `c`) and then repeated that navigation plan to generate a new route based on its updated map knowledge. The new route was again generated as a sequence of waypoint achievement goals to the destination.




```
1  Perception: [position(1,3), obstacle(up), obstacle(down),
2      obstacle(right), map(up), map(left), map(right),
3      battery(8), charging(false)]
4  Agent: -possible(b,c)
5  Agent: navigate(gettingRoute(b,d)
6  Agent: route([op(initial,b),op(left,a),op(up,e),op(up,i),
7      op(up,m),op(right,n),op(right,o),op(down,k),
8      op(right,l),op(down,h),op(down,d)]))
9  Agent: waypoint(move(b))
10 Agent: waypoint(default)
11 Agent: waypoint(move(a))
12 Action: move(left)
```

Figure 7.13: Grid Agent Run: Map Update.

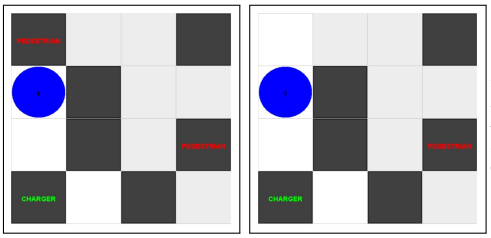
This test validated that the Agent in a Box's provided generic behaviour and behaviour prioritization could appropriately handle a situation where the provided map did not reflect all obstacles between the starting and destination. In this case, the Agent in a Box properly prioritized the map update 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 the Agent in a Box handling this type of behaviour.

7.1.2.2 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, however, there were pedestrians that could be blocking the way. As was 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 6.4 in the case studies chapter, this simple plan forces the Agent in a Box to honk its horn if pedestrians are perceived. As this was a belief-triggered plan, where the triggering event was identified as triggering a safety behaviour, the reasoner selected this plan to run instead of the waypoint plan, which was also applicable.

An image of this scenario, before and after the horn was used, are provided with the reasoning log in figure 7.14. Here, the Agent in a Box perceived the pedestrian, which triggered the pedestrian plan at the highest priority. The agent responded with honking the horn instead of continuing to the location where the pedestrian was located.

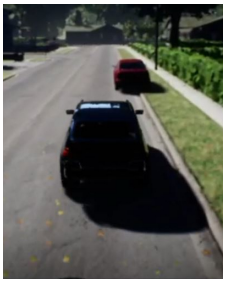



```
1 Perception: [position(0,1),
2     obstacle(left), obstacle(right),
3     map(up), map(down), map(right),
4     battery(5), charging(false),
5     pedestrian(up)
6 Action: honk(horn)
```

Figure 7.14: Grid Agent Run: Pedestrian.

The second collision avoidance scenario was when the Agent in a Box was driving the car. In this case, there was a parked car on the side of the road which needed to be avoided. As was 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 belief-triggered plan, marked at the highest priority for the reasoner and defined in listing 6.7, to avoid this obstacle. Following this plan, the car should swerve away from the obstacle as it approaches it. The car's obstacle avoidance logs and images are provided in figures 7.15, 7.16, and 7.17.

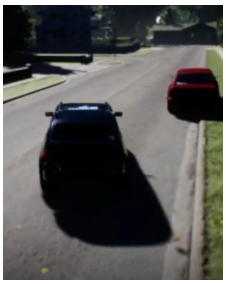
The obstacle avoidance behaviour was triggered by the perception of an obstacle, detected by the distance sensor. The plan context included a threshold which ensured that the plan was only applicable if the obstacle was within 5 m of the car. In the first instance, the car had not quite passed that threshold, so the car continued to steer using lane-keep assist.



```
Perception: gps(47.64185245724426, -122.14034349074483)
    compass(-5.8319778325) speed(7.07628059387)
    lane(-0.10786793476513527,0.0,0.0,0.6508452272497216,
    -27.035879137564546)
    obstacle(5.86613476276)
Agent:  lkaSteer(-0.10786793476513527)
Action: steering(-0.10786793476513527)
```

Figure 7.15: Autonomous Car Collision Avoidance: Approaching Obstacle.

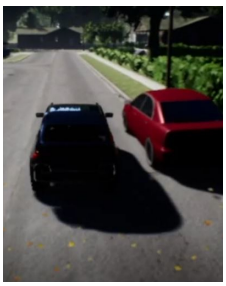
Once the car was within 5 m of the obstacle, the avoidance behaviour was applicable. As this was a triggered by a higher priority event, the reasoner selected this plan for the intention queue over the waypoint plans. The result was that the car turned to avoid the obstacle.



```
Perception: gps(47.64185511262413, -122.1403434666565)
    compass(-6.48447677242) speed(6.99733018875)
    lane(-0.1045858319380527 8,0.0,0.0,0.6310679611650488,
    -19.446601941747595)
    obstacle(4.99898219109)
Agent:  obstacleAvoid(steering((-0.3))
Action: steering(-0.3)
```

Figure 7.16: Autonomous Car Collision Avoidance: Turning to Avoid.

Once the obstacle had been avoided, the agent could then steer the car normally using the plans that implemented the waypoint goal. First, this required the use of 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.

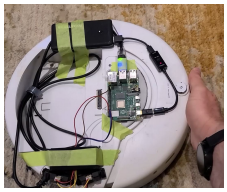


```
Perception: gps(47.641899643220256, -122.1403453806772)
    compass(-18.5631276018) speed(6.1413769722)
    lane(0.0,0.0,0.0,0.0,0.0) obstacle(86.882062912)
Agent:  compassSteer(0.05826444134296999)
Action: steering(0.05826444134296999)
Perception: gps(47.641899643220256, -122.1403453806772)
    compass(-18.9241630145) speed(6.11975860596)
    lane(0.797606907470042,0.0,0.0,0.32484076433121056,
    53.06369426751571) obstacle(86.882062912)
```

Figure 7.17: Autonomous Car Collision Avoidance: Continue.

In the last collision avoidance scenario, the mail agent applied collision recovery behaviour as defined in listing 6.17. The plan was triggered by the perception of the bumper sensor having been pressed, indicating that the robot had hit 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 7.18 provides a log of the recovery from a collision, where the robot turned and maneuvered in a semicircle around the

obstacle before turning to face the original direction.



```
Perception: battery(0.9)
            bumper(pressed)
            odomPosition(0.0,0.0,0.0)
            odomYaw(0.0)
Agent:  collision(avoid)
        !turn(left)
        !move(0.5,-3,10)
        !turn(left)
```

Figure 7.18: Mail Agent: Collision Recovery.

This section 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, and were also applicable in those contexts.

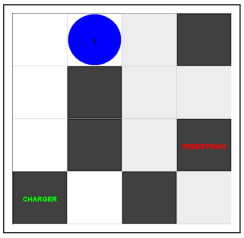
7.1.2.3 Resource Management

As was the case with the map update and collision avoidance behaviours, the Agent in a Box provided resource management behaviour for mobile robots and was demonstrated with the grid agent and the mail delivery robot. This functionality (defined in listing 5.8) was triggered by the addition of a `resource(_)` belief, as discussed in section 5.3.4, 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. The resource management plan's context contained 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 was demonstrated with both the grid agents and the prototype mail delivery robot.

In the case of the grid agent, the agent was started on its mission with a simulated battery with insufficient charge to make it to its destination. The agent should first generate a route to the destination and begin moving on the generated path. Part way to the destination it should notice that it no longer has enough power, trigger the resource management plan, suspend its mission, and navigate to the charging station. Once there the agent should dock, recharge, and wait for the charging to be completed. Once the charging is completed, the agent should undock and then resume its mission, which involved generating a new route to the destination and then moving along its path.

The run instance for the situation where the grid agent was low on battery power is provided

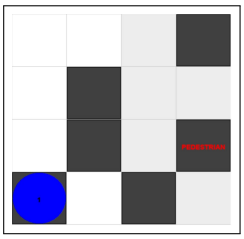
in figure 7.19. The agent started by dropping intentions, the achievement goals associated navigating to the destination. The agent then set a mental note with respect to the management of the battery, so that the battery charging plans wouldn't keep triggering. The Agent in a Box then navigated the 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.




```
Perception: [position(1,0), obstacle(up), obstacle(down),
    map(down), map(left), map(right), resource(2),
    charging(false)]
Agent: .drop_all_intentions
Agent: +managingResource
Agent: gettingRoute(n,a)
Agent: route([op(left,m),op(down,i),op(down,e),op(down,a)])
Agent: waypoint(move(a))
Action: move(left)
```

Figure 7.19: Grid Agent Run: Low Battery.

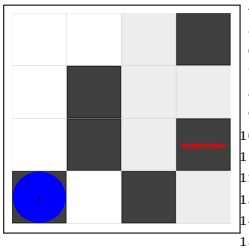
Once at the charging station, the Agent in a box docked the robot to the station to recharge. This is shown in figure 7.20.




```
Perception: [position(0,3), obstacle(down), obstacle(left),
    map(up), map(right), resource(1), charging(false)]
Agent: resource(chargingNeeded)
Agent: arrived(a)
Agent: chargeResource(atDock)
Action: station(dock)
```

Figure 7.20: Grid Agent Run: Docking With the Charging Station.

As the Agent in a Box charged the battery, it waited for the battery to be fully charged before disconnecting from the charging station, as shown in figure 7.21. The Agent in a Box then disconnected from the charging station, droped the mental note that the agent was managing the battery, and then resumed the previous mission, using its mission level mental note.




```
Perception: [position(0,3), obstacle(down), obstacle(left),
    map(up), map(right), resource(20), charging(true)]
Agent: Charging(wait)
Perception: [position(0,3), obstacle(down), obstacle(left),
    map(up), map(right), resource(19), charging(false)]
Agent: chargeResource(docked)
Agent: chargeResource(charged)
Action: station(undock)
Agent: chargeResource(unDocked)
Agent: -managingResource
Agent: resource(chargingFinished)
Agent: gettingRoute(a,d)
Agent: route([op(up,e),op(up,i),op(up,m),op(right,n),
    op(right,o),op(down,k),op(right,l),op(down,h),
    op(down,d)])
Agent: waypoint(move(e))
Action: move(up)
```

Figure 7.21: Grid Agent Run: Finished Charging.

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. A log of the mail agent using these plans to move the robot to the charging station and then dock are provided in Figure 7.22.

```
1  Inbox: chargerLocation(d)
2  Perception: resource(0.64)
3      bumper(unpressed)
4      odomPosition(1.13,1.23,0.0)
5      odomYaw(4.99)
6  Agent:  resource(chargingNeeded)
7          chargeResource(chargingNeeded)
8          navigate(arrived(d))
9          chargeResource(atDock)
10 Action: station(dock)
```

Figure 7.22: Mail Agent: Docking to Charge the Battery.

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.

7.1.3 Behaviour Validation Summary

The Agent in a Box's behaviour and prioritization have been validated by demonstrating it in a number of different application domains, including a grid agent, a simulated autonomous car, and a prototype mail delivery robot. These domains were used to highlight the core behaviours of the Agent in a Box, including mission management, navigation, manoeuvring, management of consumable resources (such as a battery), collision avoidance, and map updates.

The first focus in this section was to discuss how the Agent in a Box demonstrated how it could control the different mobile robots to achieve their goals without interruption. In effect, this demonstrated that the Agent in a Box could use its generic navigation plans, which generated routes for the agent to follow, as a set of `waypoint(_)` achievement goals to move the robots. These `waypoint(_)` achievement goals were implemented as domain-specific plans that did not include any specific concern over collision avoidance or resource management. The goal-directed plans that implemented the missions for each robot could be monitored and suspended as needed, a key feature of the Agent in a Box, and BDI agents in general.

With the mission behaviour validated, it was time to validate 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 prioritization 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 prioritization 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.

7.2 Performance

The previous sections have validated a number of useful features that the Agent in a Box provides for implementing mobile robots. The risk with providing all of these features is the potential of adding a significant performance bottleneck to robotic systems. This section assesses the runtime performance of the Agent in a Box, examining the execution time of the BDI reasoning cycle. The goal was to confirm that the reasoning cycle does not represent a significant burden, validating the practical usefulness of the Agent in a Box. These tests were completed using the computer specified in table 7.2.

Table 7.2: Benchmark Testing Configuration.

Component	Specification
Operating System	Windows 10
Central Processor	Intel Core i7-5820K CPU @ 3.30 GHz
System Memory	64 GB RAM
Graphics	NVIDIA GTX 970 with 4 GB RAM

The first of these performance tests focused on the decoupling of the reasoning cycle from the environment update of a simulator. This experiment was introduced in more detail in the approach chapter, specifically in section 5.1.1. The results from this experiment are provided in section 7.2.1.

The next performance concern was that the decision making process performed by the BDI agent would run too slowly to be useful. The test plan for assessing this performance involved running several different agent implementations for controlling the simulated autonomous car. In these tests, the car was driven toward an obstacle and the performance of the collision avoidance was observed. The experiment details and results are provided in section 7.2.2.

Another perspective on the performance of the Agent in a Box was gained through the use of profile testing. The agents' reasoning cycles were examined, focusing on the relative length of the agent's deliberations compared to the loading of the agent's perceptions and actions. These results are provided in section 7.2.3. They show that the BDI agent's decision making was not the most significant bottleneck to runtime performance, the deliberation ran faster then the

functions responsible for loading the perceptions and actions to and from the agent.

With confirmation of the agent's reasoning cycle not being a significant performance bottleneck there was still the possibility that the use of AgentSpeak for implementing the behaviour in the Agent in a Box resulted in reduced performance compared to if that same behaviour was implemented in other languages. This was assessed by comparing the performance of the reasoning cycle for the path generation functionality used by the Agent in a Box's navigation framework, implemented in three different languages. The comparison of these results is provided in section 7.2.4.

7.2.1 Decoupled Reasoning Cycle

As discussed in the approach chapter in section 5.1.1, there exists an impedance mismatch between the BDI agents, simulation tools, and robotics. The SAVI project addressed this impedance mismatch by connecting a Jason agent to a simulated environment, and demonstrating that the agent's reasoning cycle could be decoupled from a simulation cycle. This was later expanded for connecting an agent to ROS, as part of the Agent in a Box architecture. This test demonstrated that the agent's reasoning cycle was in fact decoupled from the simulation cycle by varying the simulation cycle (by varying the simulator's frame rate) and measuring the reasoning cycle period.

This early study used a simulation of a set of ground-based robots and UAVs tasked with an airport patrol mission, shown in figure 7.23. The agents patrolled the environment in search for threats. If a threat was seen, the agent would inform the other agents in the group and follow the threat. If the agent was not in pursuit of another threat, it would move to the nearest of any threats identified by the other agents.

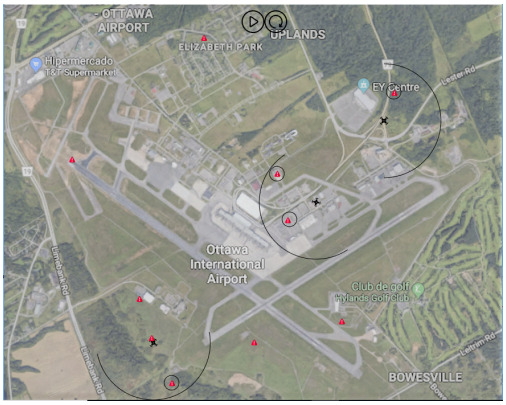


Figure 7.23: SAVI Test Scenario.

The results for this study are shown in figure 7.24. In this figure, the X axis is the requested frame rate in frames per second, which the simulator used as a desired goal it would try to

achieve. The Y axis is the measured simulation and reasoning periods. The reasoning period and the simulation period were found to be identical at lower frame rates. These periods started to diverge as the frame rate was increased above 65 frames per second. At slower periods, the reasoning system waits for up to date perceptions before reasoning. At higher speeds, the reasoning system reached its maximum performance and was unable to keep up with the simulation period.

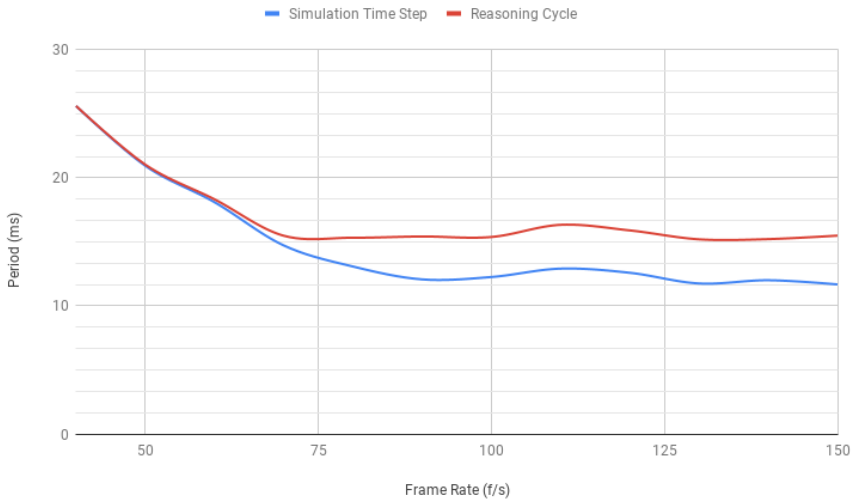


Figure 7.24: Difference in Simulation Time Step and Reasoning Cycle Periods at Different Frame Rates.

7.2.2 Runtime Performance

To understand the performance trade-off of using the Agent in a Box, the agent's performance was measured and compared to the performance of alternative agents implemented without the Agent in a Box. The key questions here were: Can the reasoner keep up with the demands of controlling a mobile robot? Does the use of the Agent in a Box behaviour framework affect whether the agent can keep up with those demands? We have seen in the related work, notably in the ARGO [55, 56] and Abstraction Engines [58] papers, a concern with whether a BDI agent can keep up with the perceptions generated by the sensors on a robot. For this experiment, we seek to measure if there is a difference in performance when using the Agent in a Box when compared to alternative implementations.

There are a number of ways that a difference in performance can manifest itself in controlling a robot. The agent's reasoning time can be measured by instrumenting the code. It is also possible to set up the robot in a scenario where the agent needs to react to some stimuli, such as a collision avoidance scenario, where the agent needs to react in time to avoid colliding with

an obstacle. This can be accomplished safely with the simulated autonomous car, where the car can be crashed without issue. The design of the car's collision avoidance behaviour includes a magic number in the plan context, the distance from the obstacle before the agent should react. This magic number was derived experimentally, by trying different ranges and examining the reaction of the agent qualitatively. This also included implicit assumptions of the sensor update frequency, number of sensor perceptions, and reasoning cycle rate. To perform this test we need to account for these assumptions. That can be done with the following:

- Measure the car's time from observing a collision risk (when the collision avoidance plan should be applicable) and the time when the stop action is generated.
- Further observe if the collision was successfully avoided or not.
- Measure the reasoning rate by instrumenting the code, the perception rate, and the times that actions are being published. Is the agent keeping up?
- Repeat these tests with different agent implementations that do not use the Agent in a Box behaviour framework.
- Look for the failure points - is there a significant difference in the reaction times/ break distance for the different agent implementations?

To complete this test a number of alternative implementations of the car's lane following and collision avoidance maneuver were needed. The navigation and resource management behaviours of the Agent in a Box were not included in the alternatives as neither were the focus for these tests, adding these features would have been needlessly time consuming. A challenge faced when building the alternatives for this agent was the design of the alternative implementations. First, it makes sense to compare the Agent in a Box to another BDI agent that does not use the Agent in a Box. It also makes sense to compare against a more classical implementation method, such as imperative programming using a block of conditional statements. Although it is fairly straight forward to implement the latter, implementing an alternative BDI agent presented an issue. The problem was to build an alternative that was indeed different from the Agent in a Box without simply rebuilding the generic components for this new BDI agent. The other concern was that the alternative BDI agents would not have the benefit of the Agent in a Box's event and option selection functions, meaning that extra care needed to be taken to ensure that the plans were provided in order of their relative priority and that there was mutual exclusion in their context guards, otherwise it would not be possible to properly drive the car as the wrong plan would be selected by Jason's default event and option selection functions.

Listing 7.1: Goal-Directed Agent Decision Making.

```
obstacleStop
    :-  obstacle(Distance)
        & Distance < 7.0.
+!waypoint(Location)
    :   atLocation(Location,_)
    <-  !controlSpeed(0);
        !controlSteering(0, lkaOff).
+!waypoint(Location)
    :   obstacleStop
    <-  !controlSpeed(0).
+!waypoint(Location)
    :   nearLocation(Location,_)
        & (not atLocation(Location,_))
        & destinationBearing(Location,Bearing)
        & (not obstacleStop)
    <-  !controlSteering(Bearing, lkaOff);
        !controlSpeed(3);
        !waypoint(Location).
+!waypoint(Location)
    :   (not nearLocation(Location,_))
        & destinationBearing(Location,Bearing)
        & (not obstacleStop)
    <-  !controlSteering(Bearing, lkaOn);
        !controlSpeed(8);
        !waypoint(Location).
+!waypoint(Location)
    <-  !waypoint(Location).
```

In order to address the issue of potentially reimplementing the generic components of the Agent in a Box in the alternative BDI agents, a design approach was taken to focus the design of the alternatives on different subsets of the AgentSpeak language by focusing the implementations to exclusively use either achievement goal triggers or belief addition triggers. The resulting agents are referred to as the *goal-directed* agent and the *reactive* agent. These design approaches, although somewhat artificial, in fact echoed early design approaches have been observed to be taken by developers when first learning to program in AgentSpeak. This implies that the design of the alternative BDI agents may mirror implementations from a novice AgentSpeak developer.

The implementation of the goal-directed BDI agent is in listing 7.1. This agent was implemented using only plans triggered by achievement goals, and there were no belief-triggered plans. The goal-directed agent only performed lane following and obstacle avoidance, all implemented in the `waypoint(_)` goal plans. The `waypoint(_)` achievement goal included checks for obstacles, using the `obstacleStop` rule shown at the beginning of the listing. The first plan stops the car when the destination has been reached, followed by a second plan that stops the car if an obstacle is in range. This is followed by the third and fourth plans that are responsible for steering the car with either the lane-keep recommendations, if they are available, or the compass steering. The fifth plan is a default plan which ensures that recursion is maintained until the waypoint is reached.

In addition to the goal-directed agent, the reactive agent implements the same behaviour using only belief-triggered plans, as shown in listing 7.2. Similar to the goal-directed agent, the agent has a `obstacleStop` rule, used for determining if there is an obstacle within the specified distance. Next is a set of belief-triggered plans which all triggered on the addition of

Listing 7.2: Reactive Agent Decision Making.

```
obstacleStop
    :-  obstacle(Distance)
        & Distance < 7.0.
+obstacle(Distance)
    :   obstacleStop
    <-  setSpeed(0).
+obstacle(Distance)
    :   (not obstacleStop)
        & navigate(Location)
        & atLocation(Location,_)
    <-  setSpeed(0).
+obstacle(Distance)
    :   navigate(Location)
        & nearLocation(Location,_)
        & (not atLocation(Location,_))
        & destinationBearing(Location,Bearing)
        & (not obstacleStop)
        & steeringSetting(Bearing, Steering)
    <-  steering(Steering);
        setSpeed(3).
+obstacle(Distance)
    :   navigate(Location)
        & (not nearLocation(Location,_))
        & destinationBearing(Location,Bearing)
        & (not obstacleStop)
        & lkaSteering(Steering)
    <-  steering(Steering);
        setSpeed(8).
+obstacle(Distance)
    :   navigate(Location)
        & (not nearLocation(Location,_))
        & destinationBearing(Location,Bearing)
        & (not obstacleStop)
        & (not lkaSteering(_))
        & steeringSetting(Bearing, Steering)
    <- steering(Steering);
        setSpeed(8).
```

obstacle beliefs, which were perceived by the agent. This approach was necessary as the agent's reasoner needed to select which triggering event to handle prior to checking which plans were applicable. Since Jason's default approach is to select the first event in its queue, there was no way to control which trigger would be selected, unlike with the Agent in a Box which provided behaviour prioritization of the triggers. As was also the case with the goal-directed agent, these plans were listed in descending order of priority and included context guards to ensure that the most appropriate plan was selected by the option selection function. The first plan handles the case where the obstacle is near the car, meaning that the car needs to stop. The second plan handled the case where the car was near the destination. This was followed by a third plan for handling the case where the car was close to the location, commanding the car to slow down. The fourth and fifth plans handled steering, either with or without the lane-keep assist, depending on if the lane steering recommendations were available.

The last alternative was designed using imperative software, implemented using Python, as shown in listing 7.3. This implementation was used in place of the BDI reasoner, monitoring the `perceptions` topic and publishing to `actions`. This script focused on simply driving the car by following the lane and stopping when an obstacle was observed within the decision point, referred to at the `stopRange` in the script. The first check in the conditional statement assesses if the car is near an obstacle, commanding the car to stop if needed. The second check determines

Listing 7.3: Imperative Agent Decision Making.

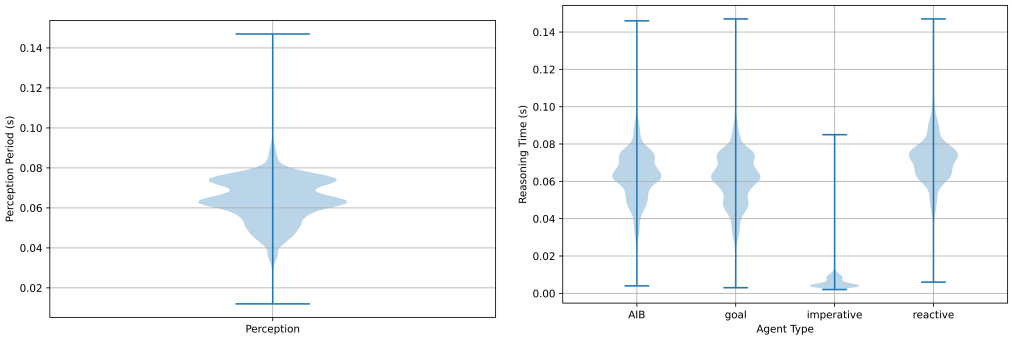
```
def decide(gps, compass, lane, speed, obstacle):
    global stopRange, currentSpeedSetting, speedSetPoint
    action = ''
    if (obstacle < stopRange):
        action = 'setSpeed(0)'
    elif currentSpeedSetting == 0:
        currentSpeedSetting = speedSetPoint
        action = 'setSpeed(' + str(currentSpeedSetting) + ')'
    else:
        (lkaSteering,_,_,c,d) = lane
        if ((c != 0) or (d != 0)):
            action = 'steering(' + str(lkaSteering) + ')'
        else:
            compassSteering = getCompassSteering(gps,compass)
            action = 'steering(' + str(compassSteering) + ')'
    if action != '':
        act(action,actionsPublisher)
        sendMessage(action,outboxPublisher)

def getCompassSteering(gps,compass):
    global destination, wgs84, declanation
    (curLat,curLon) = gps
    current = wgs84.GeoPoint(latitude=curLat, longitude=curLon, degrees = True)
    destinationBearing = destination.delta_to(current).azimuth_deg[0]
    courseCorrection = destinationBearing - (compass + declanation)
    if courseCorrection >= 20:
        steeringSetting = 1
    elif courseCorrection <= -20:
        steeringSetting = -1
    else:
        steeringSetting = courseCorrection/180
    return steeringSetting
```

if the car has started driving, setting the speed for the controller. The third and last option in this conditional statement handles steering. If the lane-keep assist module was able to detect the lane and provide a steering recommendation it is used to set the car's steering. Otherwise the car is steered using the car's destination, current location, and heading, observed using GPS and compass measurements using the provided compass steering function.

The tests were performed on the desktop computer specified in table 7.2. First, the reasoning time for each of these agents was measured. These times are reported in figure 7.25. There was a clear performance difference between the imperative agent and the BDI agents; clearly the imperative agent was able to make decisions much faster than the BDI agents. The next fastest agents were both the Agent in a Box and goal-directed agents, which appeared to have very similar reasoning performance. The reasoning period for both of these agents seemed to be comparable with the perception period. The slowest was the reactive agent, which had difficulty keeping up with the perceptions.

Although the results provided in figure 7.25 provide some insight into the reasoning performance of the agents, these results are still somewhat abstract. Ultimately, it is more intuitive to examine how these differences in reasoning affect the agent's performance in an environment. To do that, the agents were observed in a collision avoidance scenario. Specifically, this was tested with the agent for controlling the simulated autonomous car. The decision threshold for the obstacle avoidance maneuver was varied for the different agents. The effect of this was that the agent would have a different amount of time to react to the obstacle and stop. Several properties



(a) Perception period.

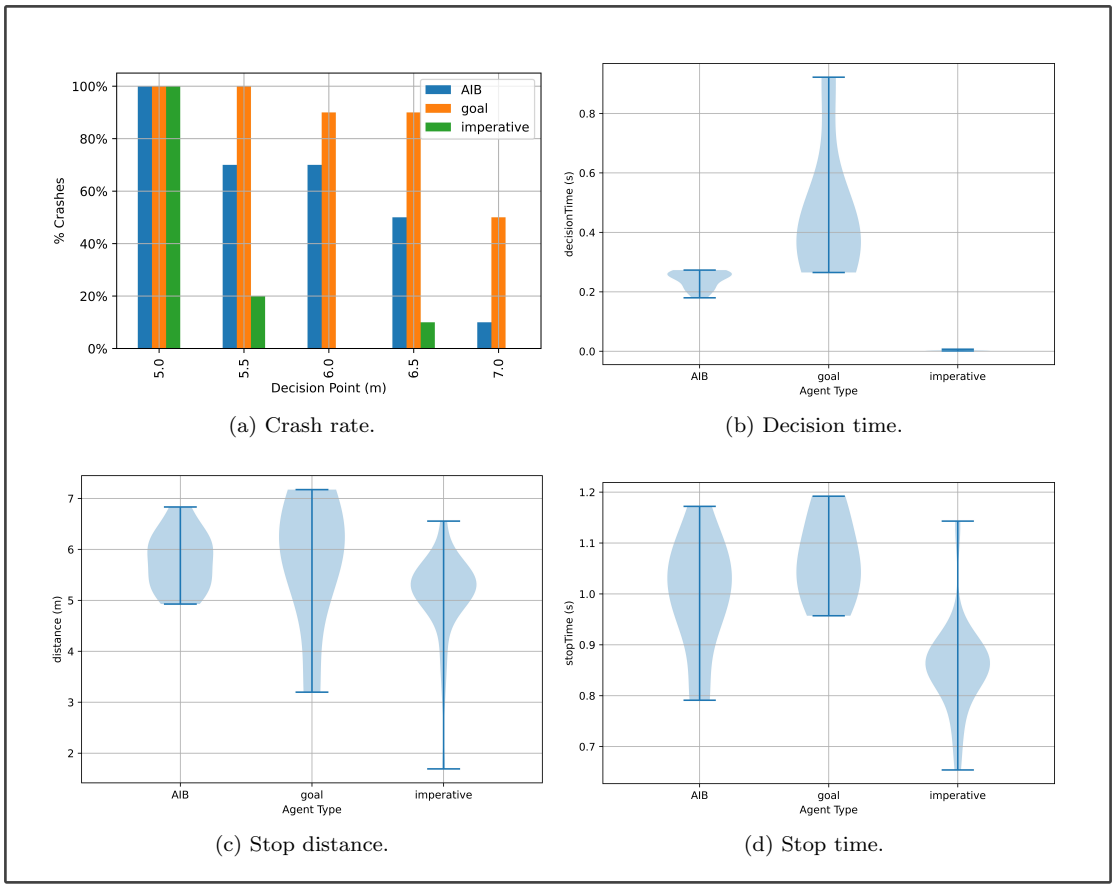
(b) Reasoning time.

Figure 7.25: Agent Perception Period and Reasoning Times.

of the agent runs were observed, including if the car crashed, the time needed for the agent to decide to stop after the decision threshold was met, the distance that the car travelled after the decision threshold was crossed, and the time it took for the car to stop. Each agent was run through the scenario 10 times for each distance threshold.

Looking at the results in figure 7.26, a clear difference in performance between the different agents is evident. Firstly, none of the agents were able to prevent a crash with a decision point of 5m. As the decision point was increased in increments of 0.5 m, differences in performance started to become evident. The imperative agent, which was previously observed to have the fastest decision time, avoided crashes in almost every subsequent test. This was significantly better than the BDI agents. Between the BDI agents, there were also clear differences in performance. First, keen readers will notice that the reactive agent is not shown in the graphs. This is because the reactive agent was not able to keep up with the sensory perceptions. The result was that it was not able to stay on the road and crashed into a fence next to the road before reaching the decision point. This occurred in every test. This left the Agent in a Box and the goal-directed agent. Between these two, the Agent in a Box was successful at avoiding more crashes than the goal-directed agent. This is also evident when looking at the decision time for the agents, where the goal-directed agent took the longest to make the decision to stop and the imperative agent was by far the fastest to decide to stop.

The observation that the Agent in a Box outperformed the goal-directed agent in the collision avoidance scenario raised a new question: Why is there a performance difference between these agents? To answer this question, a runtime experiment focusing on the performance of the reasoning cycle was performed to identify what was causing the difference in performance. This experiment used a standalone Jason project with a sequence of 50 perception sets passed to the agent for it to reason on. These perceptions were taken from the log files from the



(a) Crash rate.

(b) Decision time.

(c) Stop distance.

(d) Stop time.

Figure 7.26: Agent Collision Avoidance Test Results.

stop trial experiment and focused on a scenario where the agent should select a collision avoidance plan. Using JProfiler, the reasoning cycle was observed to assess which aspect of the reasoning cycle was causing the difference in decision time. The profile results comparing the Agent in a Box to the goal-directed agent in the collision avoidance scenario are provided in figure 7.27. This figure shows the difference in the call tree for the two agent runs, focusing on the differences in the average call times of agent's methods. As can be seen in this figure, the goal-directed agent spends significantly more time deliberating compared to the version implemented using the Agent in a Box. More specifically, the agent spends an additional 43 μs in the `applySemanticRuleDeliberate()` method.

With this result in hand, we can now reassess why the goal-directed agent spends so much more time deliberating its rules than the Agent in a Box. This was an interesting observation, as both agents used the same rules, the only difference in these agents was that the Agent in a Box used belief-triggered plan to implement the collision avoidance behaviour in a reactive manner whereas the goal-directed agent included this behaviour in the plans that were triggered by the adoption of the `waypoint` achievement goal. Table 7.3 summarizes how the plans that were triggered by different events make use of the rules. In the case of the Agent in a Box,

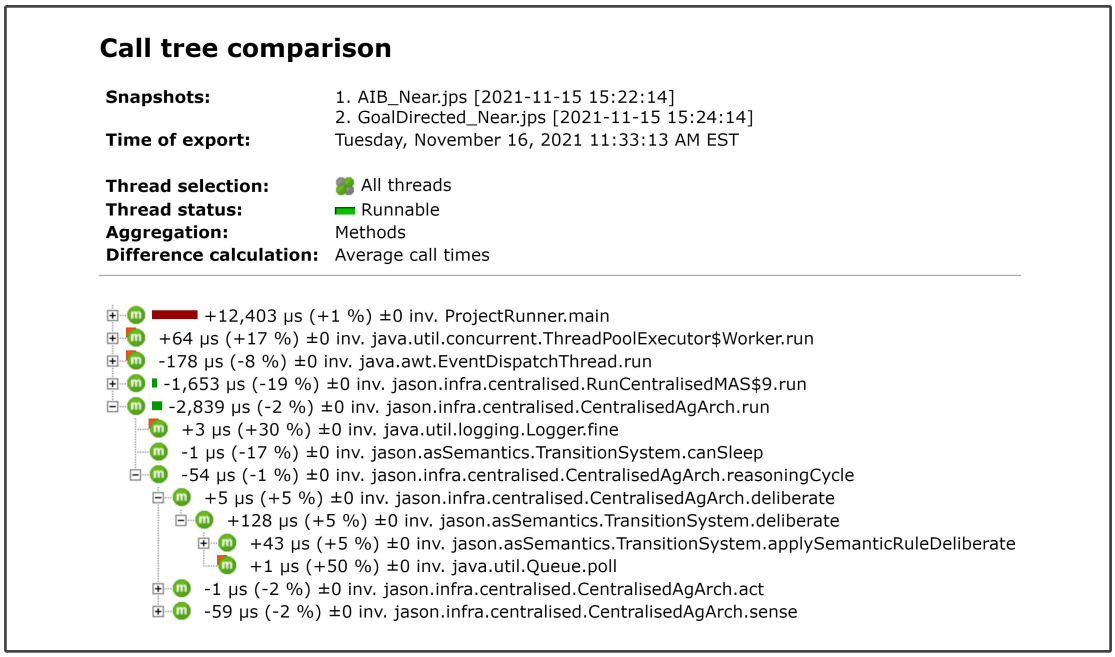


Figure 7.27: Profile Comparison of Stop Decision Bottleneck Between Agent in a Box and Goal-Directed Agents.

the obstacle avoidance behaviour used a single plan supported by a single rule. In the case of the goal-directed agent, the `waypoint` goal triggered plans included four rules. In addition, the `waypoint` plans triggered sub-plans for controlling the speed and steering of the car which also used several rules.

Table 7.3: Agent Use of Rules.

Rule	Trigger That Uses Rule		
	AIB	Goal-Directed	Reactive
obstacleStop	obstacle	waypoint	obstacle
atLocation	waypoint	waypoint	obstacle
nearLocation	waypoint	waypoint	obstacle
destinationBearing	waypoint	waypoint	obstacle
courseCorrection	controlSteering	controlSteering	obstacle
steeringSetting	controlSteering	controlSteering	obstacle
lkaSteering	controlSteering	controlSteering	obstacle

To understand the impact of these rules on the performance we must consider how a Jason agent selects which plan to set as an intention. Recall from section 2.3 that when the agent starts deliberating, it must first select which event trigger to use and then which plan applicable to that event will be set as the agent's intention. The reasoner only deliberates on rules that are used by plans triggered by the selected event. Rules that are not tied to the selected event are not processed. This means that in the collision avoidance scenario, the Agent in a Box only needed to deliberate on a single rule, and only had a single plan to choose from when setting its intention. By contrast, the goal-directed agent had four different rules that needed to be

computed in order to select to most appropriate plan for performing collision avoidance.

To summarize this finding, it seems that the use of the Agent in a Box leads to a natural reduction of the length of the context guards in plans and a reduction of necessary rules that need to be processed during the deliberation stage of the reasoning cycle. This points to a performance benefit of using the Agent in a Box for designing the behaviour of autonomous mobile robots. This seems to be a benefit of the separation of concerns between the plans: having each triggering event focus on a single issue with fewer plans needed for each triggering event. In order for this to work, it is necessary that the event selection function make appropriate decisions with respect to which triggering event should be selected. In the case of the Agent in a Box, this benefit is provided by the behaviour prioritization functionality.

Although this observation could lead to the conclusion that rules may be causing a processing burden for the agent, the use of rules can be useful for reducing code duplication in the context guards. This means that there is a possible trade-off between reasoning performance and code duplication in the plan contexts. Ideally, there should be a way to improve the reasoning performance with respect to processing the rules. There is a potential, for example, for acceleration of this processing by either implementing the rules (and contexts in general for that matter) using some other method such, as either declarative or in hardware, to make each plan context only have a single parameter. By doing this the work load of these slower methods in the reasoner could be reduced. The agent could then perceive its rule consequences rather than have a need to have the reasoner deliberate on the rules. That said, this could cause an increase in complexity which would affect the maintainability of the agent. The exploration of how the performance could be increased is discussed further in the future work in section 8.3.

7.2.3 Profile Testing

The performance of the reasoning system was measured using JProfiler in collaboration with Jason Miller, another graduate student working with BDI agents. The profiler was configured to take snapshots of each reasoning cycle. As discussed in section 2.3, the BDI 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 7.28.

Qualitatively, each agent was observed to have a degraded performance when the agent was run with the profiler. The reasoner was clearly running slower when the profiler was connected, 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

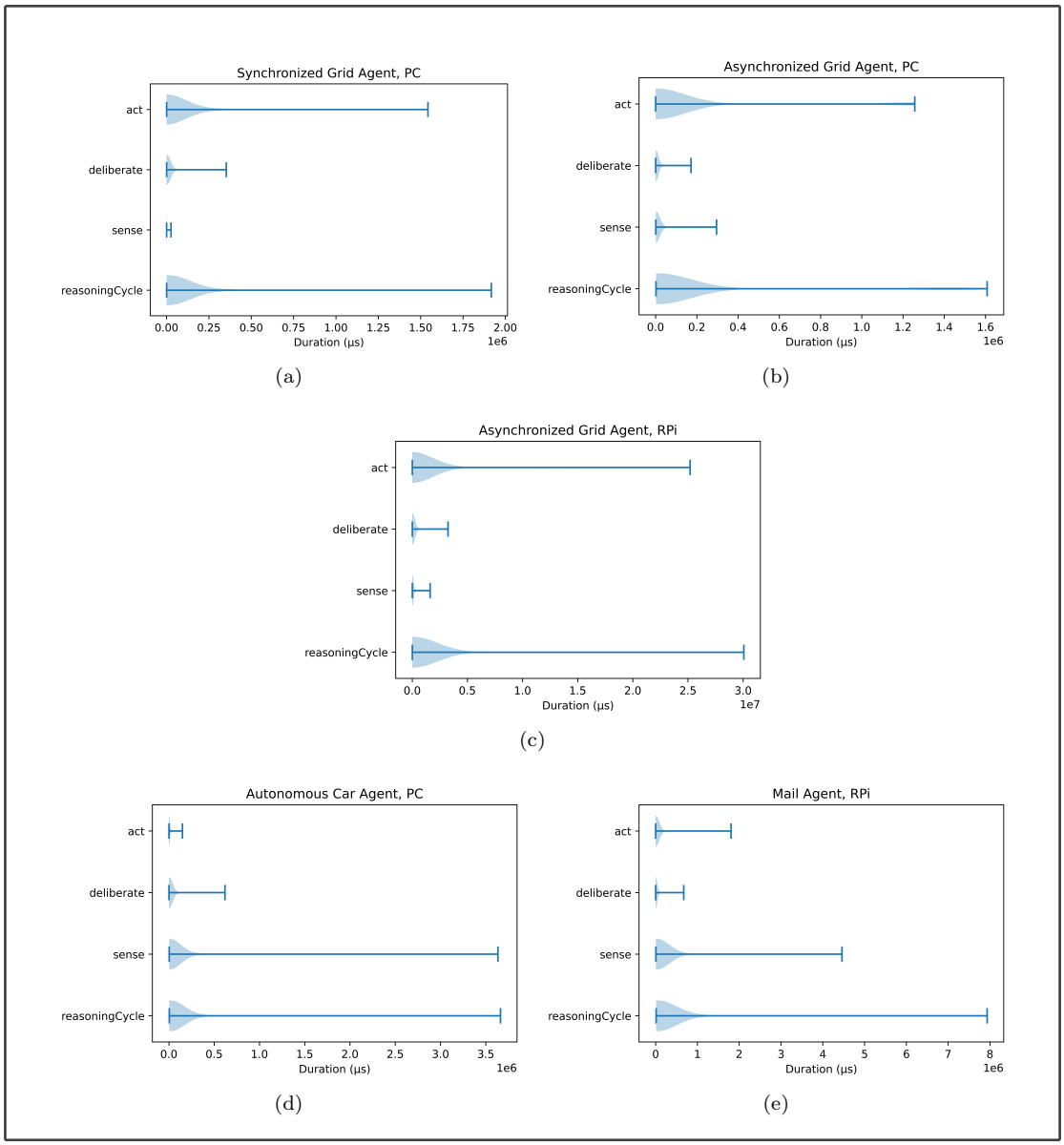


Figure 7.28: Reasoning System Performance Profiles.

key assessment here was to identify if the deliberation portion of the reasoning cycle caused a performance bottleneck for the system.

Figure 7.28a provides the profile results for a synchronized version of the grid agent, where the environment update and the agent reasoning cycle were run in a single thread of execution without ROS. 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 visualization, whereas the sense portion simply queried parameters from the environment as environment objects, a relatively fast process.

Figure 7.28b provides the performance profiles for the grid environment where the reasoning cycle and environment update were asynchronous. This environment was connected to the agent

using ROS using a desktop computer specified in table 7.2. This test was also run on a Raspberry Pi 4, as shown in Figure 7.28c. 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.

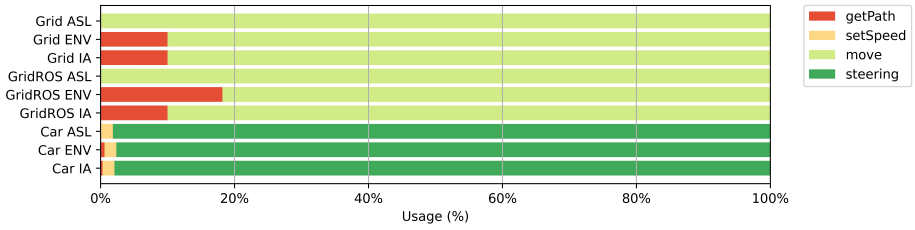
The sensing aspect of the reasoning cycle dominated the execution of both the autonomous car agent and the mail delivery agent, as shown in figures 7.28d and 7.28e respectively. This was the result of the agent waiting for new perception information before proceeding to the rest of the reasoning cycle. 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.

7.2.4 Validation of the Navigation Framework

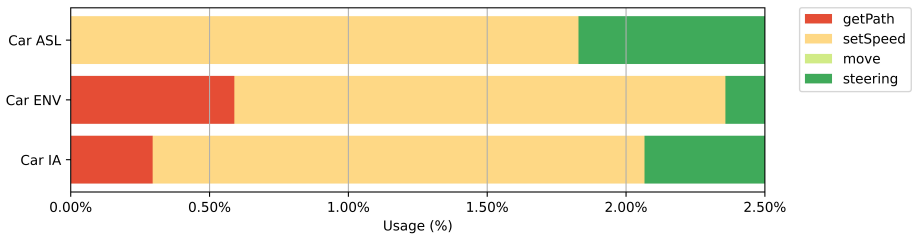
Three alternative designs were considered for the navigation component of the Agent in a Box. These were presented in section 5.3.1 of the approach chapter. The concern was that the use of the AgentSpeak language for implementing the Agent in a Box's behaviour would result in slower execution compared to software implemented in other languages. This was tested by comparing the runtime of the Agent in a Box's navigation framework, specifically the route generation using A* search. Three alternative designs of the navigation framework were considered. The three designs included: exclusive use of AgentSpeak programming for generating navigation solutions, the use of an internal action implemented in Java for generating the navigation solution, and the use of an environmental module implemented in Python for generating the navigation solution. These three designs were compared using three environments: the synchronized grid (which did not use ROS), the asynchronized grid (which did use ROS), and the simulated autonomous car. The experiments used for validating the navigation framework were performed on the computer specified in table 7.2 which was running ROS Melodic. For the performance comparisons, each agent performed 10 runs in each environment. The messages in and out of the agent and the actions performed by the agent were logged. As well, the duration of each reasoning cycle that the agent performed was logged. These logs were parsed to generate summary results which included: statistics related to the plans and actions that each of the agents used, the timelines of each agent's runs, and the reasoning cycles of each of the agents. The graphs in this section use a shorthand to save space on the plots: the AgentSpeak implementation is *ASL*, the environment-supported agent is *ENV*, and the agent which uses internal actions is *IA*. Videos of the various versions of the grid agent using different navigation approaches are available on

YouTube [98–100].

The proportion of actions used by each of the agents in each of the environments, as well as a zoomed in version for the car agent to make the less frequently used actions more visible, can be seen in figure 7.29. As expected, the agents that used the AgentSpeak implementation for navigation only used movement-related actions. The environment-supported agents used the `getPath(_,_)` action to retrieve the route prior to moving. The internal action agents used a similar action implemented as an internal action. Also noteworthy in these results is the fact that the environmentally-supported agents, which interfaced with their environments with ROS, used their `getPath(_,_)` action twice, whereas their non ROS version used it only once. The reason for this occurrence is that the agents that interact with their environment with ROS reason asynchronously from the environment, unlike their non ROS counterparts. As a result, the agents that interfaced using ROS performed additional reasoning cycles while waiting for perceptions generated by the `getPath(_,_)` action. Therefore, with these additional cycles, the agent had time to perform additional requests for their path.



(a) Action use.



(b) Zoomed in action use.

Figure 7.29: Navigation Framework - Comparison of Actions Used.

Figure 7.30 shows the proportions of the plans used by the agents. For both grid environments, the agents used plans for navigation and moving between waypoints. In the case of the environment-supported agents, the agents used another plan for loading the path generated by the environmentally based navigation module. This new plan was triggered by the agent's belief base acquiring a path belief, which occurred when the agent perceived the path generated in the environment. The environment-supported agent also used the default plan for the navigation goal while the agent waited for the path to be generated and perceived. Neither the AgentSpeak

implemented agents nor the internal action agents perceived path information, as their navigation methods loaded their path as part of their navigation plans. In some cases, the waypoint default plan was used. This was the case when the waypoint that the agent was directed to was the location that the agent was already occupying. The use of default plans for controlling the speed of the car was used when the car's speed controller had already been set to the appropriate speed.

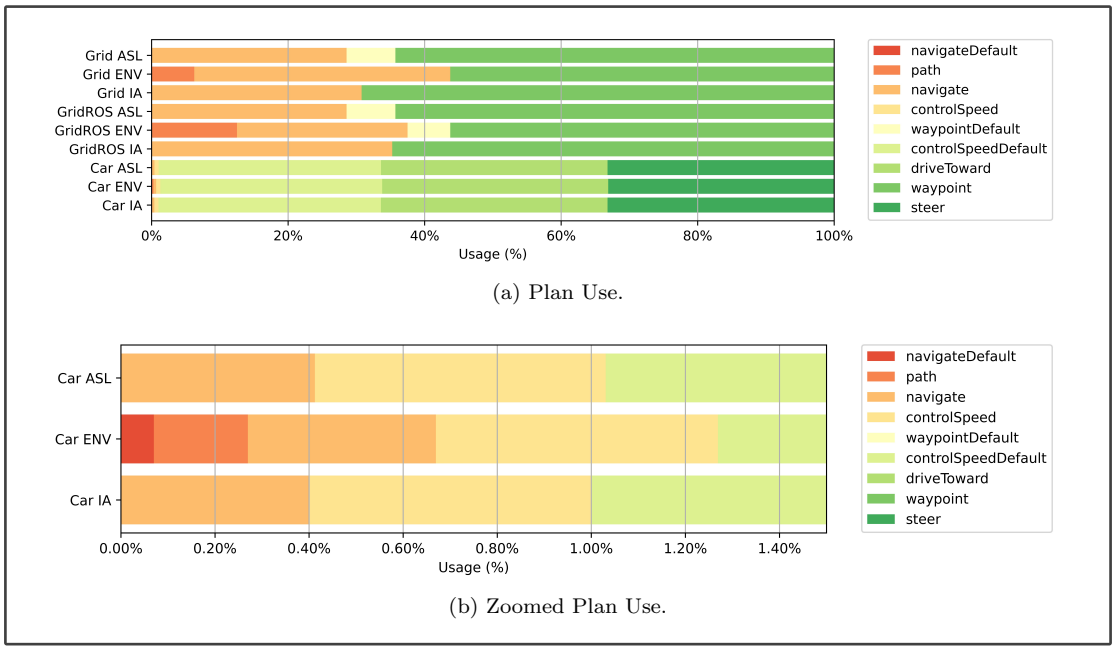


Figure 7.30: Navigation Framework - Comparison of Plans Used.

The timelines for each of the agents are provided in figure 7.31. In figure 7.32, zoomed in versions of these timelines are provided so that the early path planning portions are visible. These timelines are separated into four segments. First, the initialization of the agent, which is the period of time between when the agent adopts the goal of navigating to a destination and when the navigation process begins. Second is the agent's process for obtaining a route. This segment ends once the agent has a route loaded in its belief base. Third is the period when the agent is loading the goals associated with the route that it has loaded. This period ends once the agent performs its first movement-related action. The fourth and final time segment is when the agent is moving through the environment, ending when the agent arrived at the destination. In the case of the Jason grid world agent, it can be seen that the agents all performed their navigation routines within milliseconds of each other. The performance of each of these agents is essentially equivalent, especially given the synchronous nature of the agent's reasoning cycle with the environment. In the case of the ROS based agents, where the reasoning of the agent and the updating of the environment are asynchronous, more differences in performance can be seen. In these cases, a more dramatic change in the performance of the agents was observed, leaving

the AgentSpeak based navigator as the clear winner, completing the navigation task the fastest. That said, the difference in arrival times for each of the agents at the destination was found to be very small and likely negligible for most applications. Keen observers will also notice that the ROS interfaced agents perform their navigation on a much slower timescale to the non ROS interfaced synchronous agent. The reason for this is that the reasoner waits for a perception to be available prior to continuing the reasoning cycle. This ties the performance of the agent to the period of the perceived sensor data.

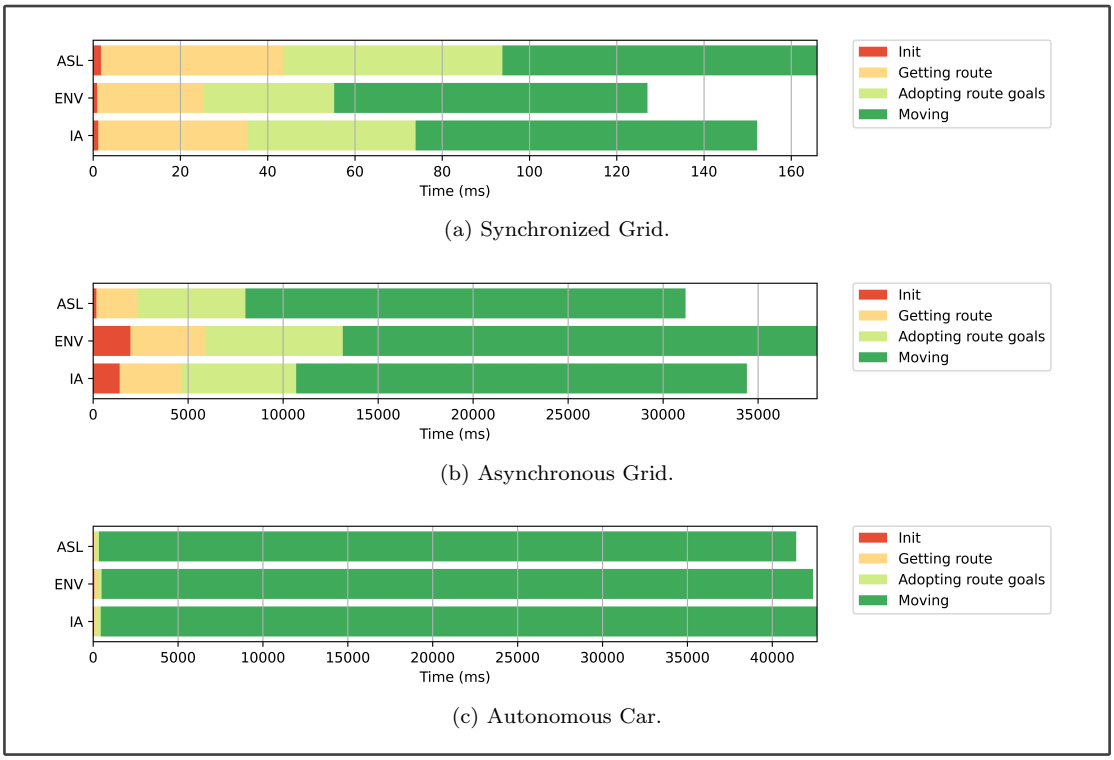


Figure 7.31: Agent Navigation Timelines.

Finally, violin plots of the period of the agent reasoning cycle for each agent in each environment are provided in figure 7.33. This figure shows that the main distributions of the reasoning cycle period were largely unaffected by the choice of navigation implementation. This was expected as once the agent loaded the navigation solution, the agent moved through the environment in a very similar way, regardless of the implementation of the navigation component. The difference was observed in the extreme cases where individual reasoning cycles were significantly longer than the others. These extreme cases were generally caused by the agent generating the navigation solution. As this portion of the agent's run is short relative to the length of the overall run, there was not a significant difference in the length of the reasoning cycle. In this case, the agent that used the internal action had the largest outliers, followed by the AgentSpeak agent. The environmentally-supported agent generally had the shortest periods.

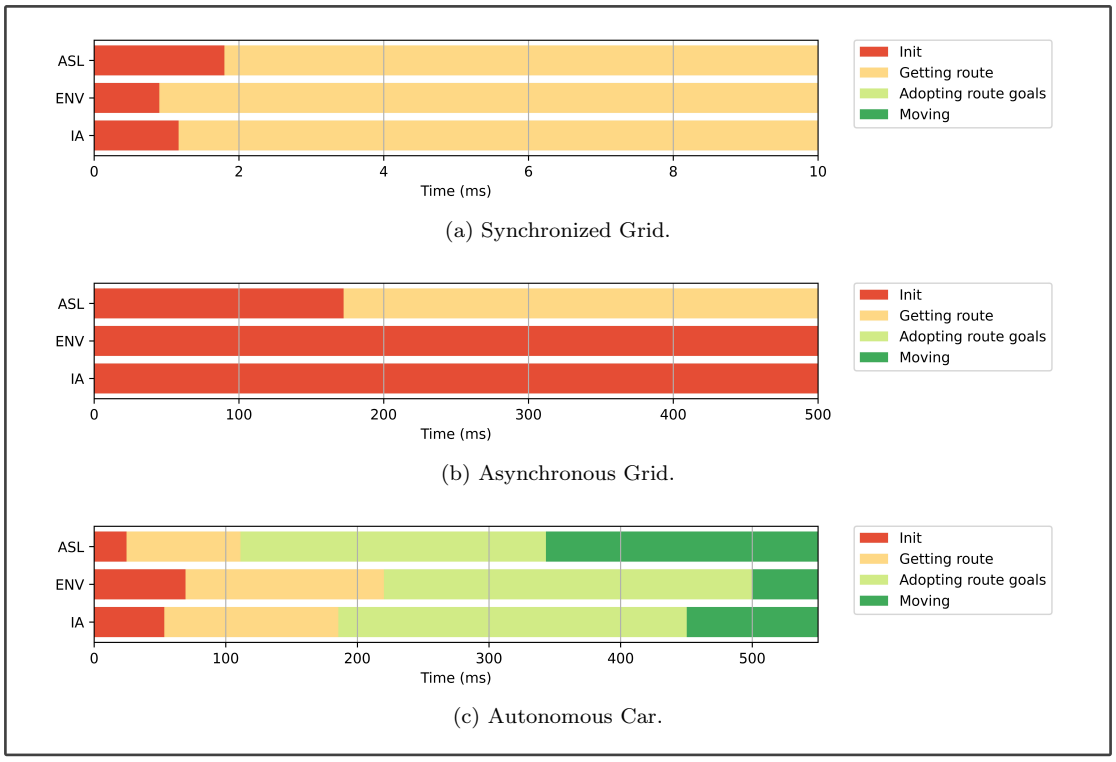
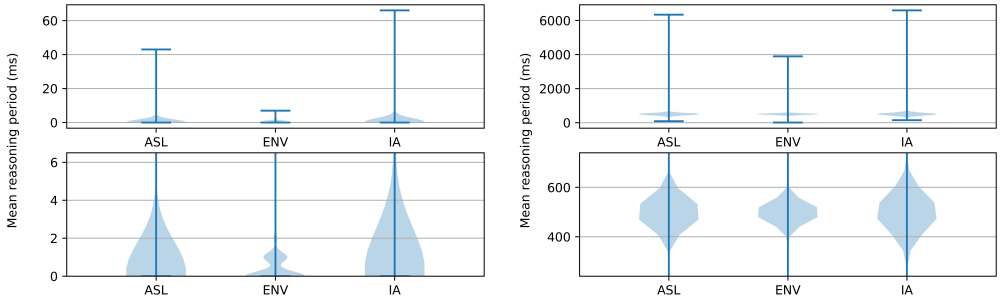


Figure 7.32: Zoomed in Agent Navigation Timelines.

The environmentally supported agent did not perform navigation internally, meaning that the reasoner was able to perform additional reasoning cycles while waiting for the solution to be generated.

A difference in the reasoning cycle periods between the different environments was observed. The difference was the result of the reasoner waiting for perceptions to be available prior to allowing the reasoning to continue. This ultimately shows that the update frequency of the sensors was a major bottleneck in the reasoning cycle. In the case of the environment that did not use ROS, an agent without this bottleneck can be observed. In this case the agent which used the AgentSpeak implementation for navigation had the longest reasoning cycle, followed by the internal action agent. The agent with the fastest reasoning rate was the environmentally supported agent, which was able to continue its reasoning cycle without the need for computing the navigation route. That said, the difference in the reasoning cycle was very small. However, it is expected that with less powerful computers, such as embedded computers, the difference could be more pronounced.

All of the navigation designs were found to work well, they all successfully performed the navigation functions and moved the agent to the required destination. It is anticipated that a user would not likely be able to tell which agent is interacting with the environment. Generally, the performance between the agents was almost identical from the perspective of a human observer.



(a) Synchronized Grid.

(b) Asynchronous Grid.

(c) Autonomous Car.

Figure 7.33: Navigation Framework Reasoning Cycle Period.

There are a number of features about the specific implementation paradigms that a programmer may wish to consider when selecting an appropriate design. For example, the representation of the map in AgentSpeak was such that the agent's knowledge of the map was a set of beliefs. Updating this map was as simple as adding or removing the needed beliefs from the belief base with a single line of code. In the case of the environmental support navigator and the internal action navigator, the process of updating the map was slightly more complex, although not unfeasible to a reasonably skilled software developer. In the case of the environmental support navigator a map update can be accomplished with the addition of new actions. Since the navigation module runs as a separate instance, the updating of the map was implemented by updating the underlying data structure. This process is similar with the internal action design, however in this case an additional factor needs to be considered. The additional factor is that internal actions are implemented as separate classes with their own state data. Adding a map update function to the library of internal actions would require passing information from one internal action object to another. This was completed by implementing the map as a singleton class, with all needed internal actions accessing it to either access or update the map. Although it is possible to implement all of these navigation versions, it is likely that a developer's personal preferences would play a significant role in selecting the approach taken for any given project.

The environment-supported navigator performed more reasoning cycles than the other agents. This was because the agent was able to reason while waiting for the external navigation module to complete the navigation task. Whereas for the internal action and AgentSpeak implementations, the navigation task was performed as part of the reasoning cycle. Although this added a negligible number of additional reasoning cycles, it could provide an opportunity for an agent to reason about other beliefs, desires, or intentions while waiting for a long task to be completed in the environment.

In a very early implementation of the navigation module, the turn-by-turn directions were implemented fully in an environmental module. There were several functional differences between that approach and the approach discussed here. In the more recent work, the turn-by-turn directions were separated from the generation of the route, allowing the agent to perform reasoning on the route. When the navigation module provided turn-by-turn directions, the module was significantly more complex, requiring constant updates of the position and orientation of the robot. With the turn-by-turn directions implemented within the agent, the reasoner now has the opportunity to perform reasoning on the route information. Conversely, this does mean that the reasoner requires some map knowledge in order to make these decisions, something that was not necessary when the turn-by-turn directions were externally generated.

7.3 Software Engineering Properties

One of the desired features of the Agent in a Box, and frameworks in general, is to improve the experience of developers who are designing software for a given purpose. In the case of the Agent in a Box, this purpose is to provide agents for controlling mobile robots. This experiment aims to assess if the Agent in a Box does in fact improve the development experience by comparing characteristics of agents implemented with the Agent in a Box against the alternatives. This assessment compares the software properties of the alternatives used for the runtime performance test, where the agent for controlling the simulated autonomous car was compared against several alternatives, as discussed previously in section 7.2.2. As these alternatives only provided obstacle avoidance and maneuvering aspects of the agent behaviour, this comparison does not consider the navigation or resource management aspects of the Agent in a Box.

There are a number of ways to measure the difference between implementations. Properties of software that are associated with maintainable code bases include coupling and cohesion [101]. In object-oriented software, coupling refers to how interconnected a class is with another class. The goal in software design is to have loose coupling, meaning that changes to one class don't break other classes forcing additional changes. The connections between classes should follow

well-defined interfaces. Cohesion refers to the number of tasks or responsibilities of each class or method. For software to have high cohesion, the modules, classes, or methods should each be responsible for one, well defined, tasks or concerns. Having higher cohesion increases the likelihood that the code can be reused. As this thesis focuses on BDI agents using AgentSpeak a means of assessing these properties in AgentSpeak code is needed. There is some precedent for using these metrics to assess declarative software implemented in languages such as Prolog [102, 103], these concepts will need to be adapted for assessing software written in AgentSpeak. This adaptation to AgentSpeak and the assessment results are provided in section 7.3.1.

Another software metric is cyclomatic complexity, first introduced by McCabe, which is a measure of the number of possible paths that exist through a program [104]. There is also precedent for using this metric for evaluating the complexity of declarative software implemented in Prolog [105]. It can be calculated using equation 7.1 where the parameters are properties of a control graph for the program. The nodes in the control graph represent lines of AgentSpeak code and the edges are execution paths between those lines through the different triggered plans. The number of edges on the control graph is represented as E, The number of nodes in the graph is N, and the number of connected components is P.

$$M = E - N + 2P \tag{7.1}$$

Ultimately, the goal should be to have software that is less complex. Therefore, the control graphs for the Agent in a Box, and the alternatives discussed in the next section, should be drawn and analysed so that the cyclomatic complexity of each approach can be assessed. The assessment of the cyclomatic complexity for the agents developed for controlling the simulated autonomous car is provided in section 7.3.2.

7.3.1 Assessment of Coupling and Cohesion

As discussed previously, the concepts of coupling and cohesion have been well defined for object-oriented software. The spirit of these properties is to assess how interconnected a class is with other classes, or in this case how interconnected a plan is with other plans as well as how many responsibilities each of these plans deals with. These ideas can be adapted for assessing the plans specified in an AgentSpeak program, rather than classes specified in an object-oriented language.

Consider first the property of coupling between different AgentSpeak plans. Recall, in object-oriented software, loose coupling between classes means that when a change is made to one class it does not require changes to other classes. In the case of AgentSpeak plans, this should mean

that a change to a plan should ideally not require changes to other plans. Even more so, it should not require changes to plans triggered by a different type of event. This implies that each triggering event should trigger the fewest possible number of plans, ideally with fewer terms in the context checks, and be responsible for fewer concerns. If additional concerns need to be handled, these should be handled with another triggering event, such as adopting an achievement goal to handle it.

Shifting the focus to cohesion, plans that have high cohesion are only responsible for a single concern, making them more reusable. If the triggered plans implement a single behaviour we would expect there to be higher cohesion compared to plans that handle additional behaviours. For example, a triggering event that triggers plans that handle the low level details of both steering and speed control, rather than only one of those two concerns would have lower cohesion. A way to improve the cohesion would be to use relevant sub-goals. As there are not many behaviours triggered by each event, there should not need to be many plans to implement this behaviour. Beliefs should only be maintained by plans triggered by the same triggering event, separating the concern for these beliefs to specific plan sets. This separation of concerns between the different behaviours encapsulates them within the plans that are triggered by each event.

By designing the AgentSpeak software with looser coupling and high cohesion, as presented in the previous two paragraphs, we should expect to have more triggering events with fewer plans per triggering event. These plans would have simper context guards and should be implemented without concern for the order in which they are provided in the plan base, nor what other goals the agent may have. This summary points to the types of properties that should be measured in order to assess the coupling and cohesion of an AgentSpeak program. The first of these was the number of triggering events in the program. The second was the number of plans associated with each triggering event. Third was a property of the context portion of each of those plans. Although plan contexts can take the form of various types of logic expressions, as per the AgentSpeak's EBNF found in Appendix A.1 of the Jason textbook [3], all the plans studied in this case study used contexts that were conjunctions of logic expressions. The number of these logic expressions was counted to measure the complexity of the plan contexts. The fourth, and last, was the number of concerns addressed by the plans triggered. The ways in which these properties can help in assessing the coupling and cohesion of AgentSpeak software is outlined in the next paragraph. The measurement of each of these properties for the three different BDI agents that drove the simulated autonomous car are provided in table 7.4.

The Agent in a Box had four different event triggers, each responsible for a single concern: either avoiding an obstacle, controlling the car, controlling the speed, or controlling the steering. The plans that implemented these concerns had relatively simple plan contexts, consisting of

Table 7.4: Properties of BDI Agent Alternatives.

		AIB	Goal-Directed	Reactive
# Triggering Events		4	3	1
# Plans per Event	Obstacle	1	–	5
	Waypoint	4	5	–
	ControlSpeed	3	3	–
	ControlSteering	3	3	–
# Logic Expressions per Context	Obstacle	1/1=1	–	21/5 = 4.2
	Waypoint	6/4=1.5	9/5 = 1.8	–
	ControlSpeed	3/3=1	3/3=1	–
	ControlSteering	3/3=1	3/3=1	–
# Concerns per Trigger	Obstacle	1	–	4
	Waypoint	1	2	–
	ControlSpeed	1	1	–
	ControlSteering	1	1	–

conjunctions of fewer logic expressions than the alternatives. It also had fewer plans associated with each triggering event. The second BDI agent was the goal-directed agent. The main difference between the goal-directed agent and the Agent in a Box agent was how collision avoidance was handled. This agent did not use any belief-triggered plans, meaning that the avoidance of collisions needed to be handled by the waypoint plans. That meant that the waypoint plans had two concerns to manage, rather than just one. This increased the number to logic expressions joined by conjunctions in the plan contexts as well as the number of plans. The plans for controlling the speed and steering of the car were similar to the ones for the Agent in a Box. There was another important difference between the goal-directed agent and the Agent in a Box: The goal-directed agent did not have access to the Agent in a Box's prioritization of plans, meaning that the plans needed to be listed in order based on their relative priority with each other or they needed to have mutual exclusion guaranteed in their contexts. This dramatically increased the coupling between the plans. The reactive agent, which used exclusively belief-triggered plans, also had this problem. This resulted in the most complicated of the BDI agents, where a single event trigger was associated with all of the aspects of controlling the car. The five plans that were implemented had relatively complicated contexts, far more so than either of the alternative agents. This agent also needed to have mutual exclusion between the plan contexts and the plans needed to be listed in their relative priority.

Another important aspect of the AgentSpeak software are the rules that have been defined. These rules also contribute to the properties of coupling and cohesion of the software. The rules are considered in two ways. Table 7.5 summarizes the properties of the rules that are considered. This includes the number of logic statements joined by conjunctions[1], the number of versions of

[1]Although AgentSpeak rules do not necessarily need to take the form of logical conjunctions, all the rules used by the agents considered did take this form.

the rule, and any internal actions used by the rule.

Table 7.5: Properties of Rules Used.

Rule	# Logic Statements Joined by Conjunctions	# Versions	Internal Actions Used
obstacleStop	1	1	–
atLocation	4	1	range
nearLocation	4	1	range
destinationBearing	3	1	bearing
courseCorrection	3	1	–
steeringSetting	2,2,3	3	–
lkaSteering	2	1	–

Although this first assessment of the properties of the rules provides some insight into the complexity of these rules, as these rules were all used by all three of the AgentSpeak implemented agents, it is more important to consider how these rules were used by the different agents. Table 7.6 (which is repeated from table 7.3) summarizes the uses of the rules. The primary difference between the Agent in a Box and the goal-directed versions was the handling of the obstacles. In the case of the Agent in a Box, this was handled by a dedicated plan with a separate triggering event whereas the goal-directed agent included that behaviour in the implementation of the `waypoint` achievement goal. Lastly, the reactive agent version used all of these rules with the same triggering event.

Table 7.6: Agent Use of Rules (Repeat of Table 7.3).

Rule	Trigger That Uses Rule		
	AIB	Goal-Directed	Reactive
obstacleStop	obstacle	waypoint	obstacle
atLocation	waypoint	waypoint	obstacle
nearLocation	waypoint	waypoint	obstacle
destinationBearing	waypoint	waypoint	obstacle
courseCorrection	controlSteering	controlSteering	obstacle
steeringSetting	controlSteering	controlSteering	obstacle
lkaSteering	controlSteering	controlSteering	obstacle

In addition to the BDI agents, an imperative version of the car controller was written using a Python function which was made up of a set of nested conditional statements. This function provided an action for the car given a set of perceptions, and was responsible for controlling the speed, steering, and avoiding collisions. The conditional statements in this function had a depth of three, making this a rather complicated function with relatively poor cohesion, although it did not interact with other modules, meaning it had better coupling.

Table 7.7 provides a relative comparison of the different implementations in terms of their performance on coupling and cohesion. The Agent in a Box scored highest on both coupling and cohesion. This was for several reasons: the plans could be provided in any order, each triggering

event was tied to a single responsibility, and the resulting plans were less complex than the plan implemented in the alternative agents. The Agent in a Box also provided a more capable agent, providing navigation, map, and resource management, features that were not included in the other agents. This meant that the alternatives would have needed additional code to provide these features. A detraction for the the Agent in a Box agent however was that the developer would need to understand the role of each of the components that they needed to provide and how they would be used by the Agent in a Box. The imperative agent did not have any connections with other modules and therefore the coupling metric was not applicable. The imperative agent did score poorly on cohesion though, as it was implemented with a nested conditional statement that needed to address all the behaviours of the agent. If any changes were required, perhaps to add resource management behaviour, the nested conditional statements would need to be unravelled and redesigned. The goal-directed agent was second place for coupling, and second place for cohesion, ranked behind the Agent in a Box. For coupling this was because of the lack of prioritization of behaviour, meaning that the plans needed to be listed in relative priority while ensuring mutual exclusion. For cohesion this was because the concerns for collision avoidance were combined with the concern for maneuvering the car. The worst scores were held by the reactive agent, which had very poor coupling and cohesion.

Table 7.7: Summary of Coupling and Cohesion of Agent Alternatives.

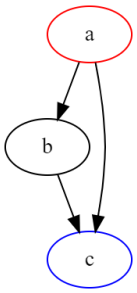
Version	Relative Rank Best: 1, Worst: 4	
	Coupling	Cohesion
Agent in a Box	1	1
Goal-Directed	2	2
Reactive	3	4
Imperative	N/A	3

7.3.2 Assessment of Cyclomatic Complexity

To assess the cyclomatic complexity of the different agents that were implemented, it was first necessary to draw the node graphs for each of the components of the agents. The first of the agents assessed was the Agent in a Box agent for driving the simulated autonomous car. There were four components to this agent. These included the plan reactive plan for collision avoidance, plans for the waypoint achievement goals, plans for controlling the speed of the car, and plans for controlling the steering of the car.

The first component to assess is the Agent in a Box's collision avoidance plan, defined in listing 6.7. The node graph for this component is shown in figure 7.34. The collision avoidance plan is a belief-triggered plan that is triggered on the perception of an obstacle and applicable when that obstacle is within range of the car, represented in the figure as node `a`. There are

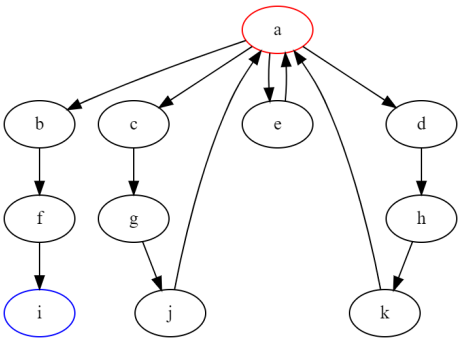
two execution paths for this component: either the plan is applicable, meaning the collision avoidance plan will run (shown as node `b`), or it is not and the plan is not executed. Both paths end at node `c`, the completion of the plan. This behaviour included a single connected component: the Agent in a Box behaviour prioritization framework.



Property	Value
# Edges (E)	3
# Nodes (N)	3
# Connected Components (P)	2
Cyclomatic Complexity (M)	4

Figure 7.34: Agent in a Box - Collision Avoidance.

Figure 7.35 provides the graph and cyclomatic complexity analysis for the Agent in a Box implementation of the plans that implement the `waypoint` achievement goal, defined in listing 6.8. As can be seen in the graph, the adoption of this achievement goal is the trigger for four plans. One which handles the case where the car has arrived at the waypoint (nodes b through i), the case where the car is approaching the waypoint location and needs to reduce speed (the path from node c through j), another where the car is not near the waypoint location (nodes d through k), and a default plan which ensures recursion (node e). These plans connect to three other modules including the Agent in a Box behaviour prioritization framework, the steering achievement goal, and the speed control achievement goal. This set of plans had three connected components, including the Agent in a Box framework, and sub-goals for controlling the steering and speed of the car.

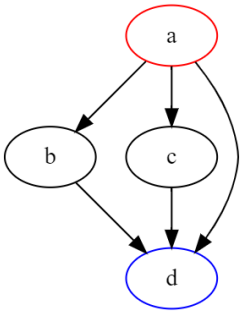


Property	Value
# Edges (E)	13
# Nodes (N)	11
# Connected Components (P)	3
Cyclomatic Complexity (M)	8

Figure 7.35: Agent in a Box - Waypoint.

Figure 7.36 provides the graph for the steering plans for both the Agent in a Box and

the goal-directed agent implementations for the car controller, defined in listing 6.11. The main difference between the two was that the Agent in a Box included a the Agent in a Box prioritization framework among its connected components and the goal-directed version did not. This allowed the plans for the Agent in a Box to be written in any order, whereas the plans of the goal-directed agent needed to be written in order of their relative priority so that the default selection functions would select the most appropriate plan. Both agents included a set of rules used for assisting with the calculation of the steering setting among the connected components. This meant that the Agent in a Box had two connected components for these plans whereas the goal-directed version only had one. There were three plans for controlling the steering, all triggered by the `!controlSteering(_,_)` event, represented by node `a`. The first of these handles the case where the lane-keep assist is either not available or is not being used meaning that the car needs to be steered using the compass (node `b`). The second of the plans uses the lane-keep assist provided steering setting (node `c`). Lastly, there is a default plan for error cases, which does not have a plan body, therefore it does not have any node associated with it. All paths end at node `d` which completes the plan.

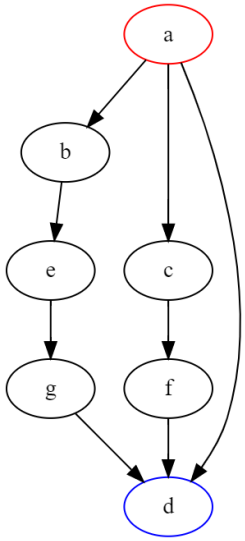


Property	Value
# Edges (E)	5
# Nodes (N)	4
AIB # Connected Components (P)	2
AIB Cyclomatic Complexity (M)	5
GD # Connected Components (P)	1
GD Cyclomatic Complexity (M)	3

Figure 7.36: Agent in a Box and Goal-Directed - Steering.

The graph of the speed controlling functionality for both the Agent in a Box and goal-directed agents, as defined in listing 6.10 are provided in figure 7.37. Similar to the steering functionality, the main difference between the two was that the Agent in a Box used the prioritization framework whereas the goal-directed agent did not have access to this functionality, meaning that the Agent in a Box had a single connected component and the goal-directed agent did not. Again, this allowed the Agent in a Box version to have the plan provided in any order, unlike the goal-directed version. There were three speed control plans, all triggered by the `!controlSpeed(_)` event, that could be run depending on their contexts, represented by node `a`. The first, represented by the path of nodes from `b` through `g`, handled updating the agent's mental note belief associated with previous speed settings and then set the desired speed of the car using the `setSpeed(_)` action. The second plan, nodes `c` and `f`, handled the case where there

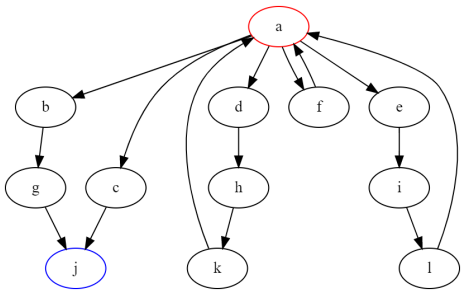
was no previously set speed setting. Lastly there was a default plan for the scenario where the speed did not need to be adjusted as the setting was already properly set. All the paths lead to the end of the plan, represented by node `d`.



Property	Value
# Edges (E)	8
# Nodes (N)	7
AIB # Connected Components (P)	1
AIB Cyclomatic Complexity (M)	3
GD # Connected Components (P)	0
GD Cyclomatic Complexity (M)	1

Figure 7.37: Agent in a Box and Goal-Directed - Speed.

The goal-directed agent implementation of the `waypoint` achievement goal, defined in listing 7.1, is shown in figure 7.38. The plans for this scenarios were similar to those for the Agent in a Box version, however in this case there was an additional plan (the path of nodes from `c` to `l`) for handling the collision avoidance case. This set of plans had two connected components: the sub-goals for controlling the steering and speed of the car.

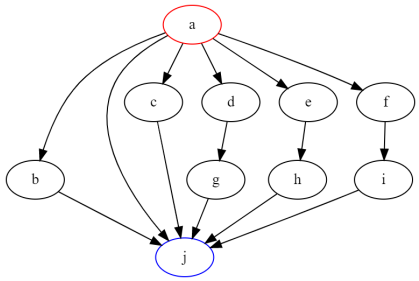


Property	Value
# Edges (E)	15
# Nodes (N)	12
# Connected Components (P)	2
Cyclomatic Complexity (M)	7

Figure 7.38: Goal-Directed - Waypoint.

The behaviour graph for the reactive agent, which was defined in listing 7.2, is shown in figure 7.39. In this case, all collision avoidance, steering, and speed control logic were provided for a set of belief-triggered plans with no other connected components. The plan represented by node `b` handled collision avoidance and the plan handled by node `c` handled the scenario where the agent had arrived at the waypoint. The plan represented by nodes `d` and `g` handled slowing

the car as it neared the waypoint and used compass steering. The plan represented by nodes `e` and `h` was for the case when the car was far from the destination and it needed to steer with the lane-keep assist. Similarly, the plan represented by nodes `f` and `i` handled the scenario where the lane-keep assist was not available.



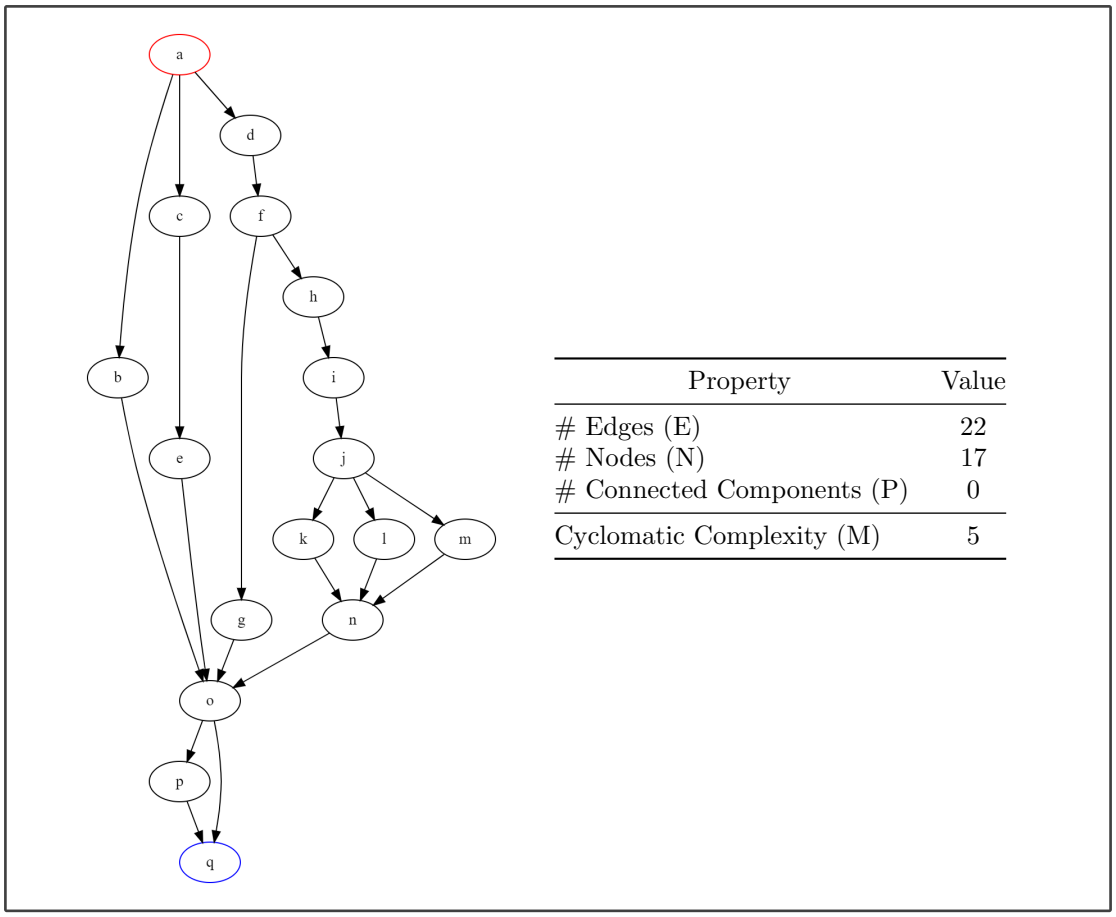
Property	Value
# Edges (E)	14
# Nodes (N)	10
# Connected Components (P)	0
Cyclomatic Complexity (M)	4

Figure 7.39: Reactive Agent.

The final agent compared was the agent implemented using imperative programming using Python, provided in listing 7.3. This version did not have any connected components. The paths starting with nodes `b`, `c`, and `d` were used to determine if the car was near an obstacle, if the car was stationary, or if the car was moving and either had lane-keep assistance or not. If the lane-keep assistance was not available, it needed to use compass steering (nodes `h`, `i`, `j`, `k`, `l`, `m`, and `n`). to steer. Lastly, the setting of the action was handled by nodes `o`, `p`, and `q`.

With the cyclomatic complexity analysis completed for the components of the different agent implementations they are compared in table 7.8. From this table, it is apparent that the reactive and imperative agents were found to have the lowest cyclomatic complexity. The goal-directed and Agent in a Box were made up of more components. Of those, the Agent in a Box had higher cyclomatic complexity than the goal-directed agent due to the connections to the prioritization framework, which added the need for an additional belief to be set in order for the reasoner to properly prioritize the event triggers. However, having this additional functionality enabled the developer to provide the plans for the Agent in a Box in any order. Whereas the goal-directed and reactive agents needed to have their plans provided in order from highest to lowest priority in order for the default event and option selection functions to select the most appropriate plan to run. Furthermore, the goal-directed, reactive, and imperative agents all had more nodes and edges in their graphs, making the maintenance of those graphs likely more complicated than the maintenance of the Agent in a Box graphs.

There is also another piece to this complexity discussion: the functionality of the agents themselves. In the case of the Agent in a Box, the framework provided a number of features at no cost to the developer. These include navigation, map update, and resource management.



Property	Value
# Edges (E)	22
# Nodes (N)	17
# Connected Components (P)	0
Cyclomatic Complexity (M)	5

Figure 7.40: Imperative Agent.

This means the developer using the Agent in a Box only needed to provide functionality for the agent to avoid collisions and move between waypoints, in this case using steering and speed control sub-goals. By comparison, although the alternative agents provided the same ability to avoid collisions and move down the street, these agents did not provide any navigation or resource management functionality. This means that if these functionalities were to be needed, they would need to be developed for each of those agents, ultimately increasing their complexity. Therefore, although the code for implementing the movement and collision avoidance for the Agent in a Box came with additional cyclomatic complexity than the alternatives, the act of simply providing this functionality yielded a significantly more capable agent.

7.3.3 Software Engineering Properties Summary

This section has provided an assessment of the software engineering properties of the Agent in a Box by focusing on the car control agent and comparing it to alternative implementations to examine the properties of coupling, cohesion, and cyclomatic complexity. These alternatives were implementations used for the performance test trials. There were a number of limitations to

Table 7.8: Summary of Cyclomatic Complexity Parameters.

Graph	# Edges (E)	# Nodes (N)	Names of Connected Components	# Connected Components (P)	Cyclomatic Complexity ($M = E - N + 2P$)
AIB Collision Avoidance	3	3	AIB Prioritization	1	2
AIB Waypoint	13	11	Steering Goal Speed Goal AIB Prioritization	3	8
AIB Speed	8	7	AIB Prioritization	1	3
AIB Steering	5	4	AIB Prioritization Steering Rules	2	5
Goal-Directed Speed	8	7	–	0	1
Goal-Directed Steering	5	4	Steering Rules	1	3
Goal-Directed Waypoint	15	12	Steering Goal Speed Goal	2	7
Reactive	14	10	–	0	4
Imperative	22	17	–	0	5

these assessments. These included the design of these alternative agents, which were somewhat contrived, using limited approaches for designing BDI agents. That said, these approaches mirror approaches that were taken when first learning to use AgentSpeak and BDI, and therefore highlight possible design approaches that a newer BDI developer may take. Furthermore, the comparison was of agents that did not need navigation or resource management, however the Agent in a Box provided those features in addition to the features compared. This means that the other agents would have needed additional plans, functions, etc. to provide these additional features. They would therefore have had worse scores on cyclomatic complexity, coupling, and cohesion in order to support those features. The Agent in a Box, however, already had those features and would not get any more complicated and not need any additional plans.

Despite these limitations, a number of observations have been made. The agent designed with only reactive plans had the lowest overall cyclomatic complexity, but scored poorly on coupling and cohesion. The imperative agent had similar cyclomatic complexity to the reactive agent but again it scored poorly on coupling and cohesion. The Agent in a Box actually had the worst cyclomatic complexity scores, primarily due to the connections it had to the prioritization framework components, however it scored very well on coupling and cohesion compared to the alternatives. This was because the Agent in a Box agent was able to keep each triggering topic focused on a single concern, with fewer conditional terms needed in the context guards for those plans. This indicates that the Agent in a Box enforces good design practices on the

developer. In addition, the Agent in a Box provided additional features, including navigation, map management, and resource management. The Agent in a Box also had the advantage of allowing the plans to be implemented in any order, unlike all the alternatives, which either had to ensure that the plans had mutually exclusive contexts or had to ensure that they were provided in order of their relative priority. Ultimately, this means that although the Agent in a Box's plans were more complicated than the alternatives, for implementing purely collision avoidance and maneuvering behaviour, these same plans came with the added benefit of providing navigation, resource management, plan prioritization, etc. This functionality was missing from the alternative agents, meaning that those agents would need significant effort to add that functionality.

7.4 Summary

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

The validation experiments took three main themes. First, the properties of the Agent in a Box were validated. 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 were 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 prioritization 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 while reducing the development burden for the developer of mobile robots.

The second theme of the validation experiments examined the runtime performance of the Agent in a Box, specifically of the agent's reasoning cycle. This included a number of experiments including: demonstrating that the reasoning cycle was indeed decoupled from the environment's refresh in an early simulation experiment, a set of reaction trials to assess the responsiveness of the agent compared to several alternatives, and profile tests of the reasoner, comparing the performance of different implementations of the navigation framework's route generation function

in different languages. The experimental results demonstrated that, although the Agent in a Box doesn't keep up with a purely imperative implementation, it does provide a performance boost compared to the alternative BDI agents tested. This was found to be related to the agent's processing of rules. It seems that the use of the Agent in a Box leads to a natural reduction of the length of the context guards in plans and a reduction of necessary rules that need to be processed during the deliberation stage of the reasoning cycle. Additionally, the profile results found that agent's decision making was not a substantial performance bottleneck to the Agent in a Box. Although there were small performance differences observed between the different implementations of the navigation framework compared, it is unlikely that a human observing a robot implemented with the Agent in a Box would observe any difference between the different implementations.

The third and final theme was an assessment of the software properties of the Agent in a Box in comparison to the alternative agents used in the stop trials for the runtime performance assessments. In examining the agents it was found that the Agent in a Box had very attractive properties in terms of coupling and cohesion. It seems that the Agent in a Box enforces good practices on developers. The Agent in a Box did not score well was cyclomatic complexity – it scored poorly compared to the alternative agents. This poor score can be attributed to the fact that the Agent in a Box had more connections to other components, such as the behaviour prioritization framework, than the alternatives. That said, these additional connections enabled the developer to provide plans in any order, without concern for their relative priority, providing a benefit to the developer. Another factor that affected the complexity of the alternatives was that the alternative agents would only preform maneuvering and collision avoidance, unlike the Agent in a Box, which came with navigation and resource management as part of the framework. This means that, although the implementation of the maneuvering aspects of the Agent in a Box were more complex, these aspects of the implementation were all the developer would need to do to take full advantage of the Agent in a Box, whereas the alternative agents would need additional development to provide those features.

Chapter 8

Conclusion

In the introduction, the following questions were asked and subsequently answered by this thesis: "Can a BDI agent framework be used for developing and controlling autonomous mobile robots? How would such a framework address the limitations identified by Bordini et. al. [12]? Can it be expected that agents developed with such a framework would work well in environments representative of the real world? Do agents implemented with the Agent in a Box benefit from improved ease of development? What metrics can be used for assessing this? Are there performance trade-offs for using the Agent in a Box? If so, what are they?". These questions were answered through the development and demonstration of the Agent in a Box, which provided the key modules needed by a variety of mobile robotic agents. These modules include a means for connecting the agent to the environment, a customized reasoning system which appropriately prioritizes the agent's behaviour, and common behaviours. Using case studies on a variety of environments, including a grid, a simulated autonomous car, and a prototype mail delivery robot, the Agent in a Box was successfully demonstrated and validated. The key accomplishments of this thesis are highlighted in section 8.1. The limitations of this thesis are discussed in section 8.2, followed by a recommendation in section 8.3.

8.1 Key Accomplishments

The Agent in a Box, a framework for using BDI agents to control autonomous mobile robots has been presented in this thesis. The Agent in a Box provides a means of connecting the agent to an environment using ROS as well as the design of the nodes needed for connecting the robot's sensors and actuators to the agent so that the agent can make use of them. The Agent in a Box also provides customization to Jason's BDI reasoning system so that the agent can properly prioritize behaviours using the event and option selection functions. As well, the Agent in a

Box includes implemented plans for general agent behaviour needed by various different mobile robots. The Agent in a Box has been demonstrated with several application domains including grid-based environments, a simulated autonomous car, and a prototype mail delivery robot. By using the Agent in a Box, the use of BDI agents implemented with Jason and AgentSpeak was demonstrated to successfully control the robots. Furthermore, the use of the Agent in a Box reduced the development burden for each application domain as the behaviour common to each application domain, such as navigation, was provided by the Agent in a Box. The plans which implement the agent's behaviours can be implemented without concern for how these plans can be interrupted. For example, the plans for moving the robot between waypoints could be implemented without concern for collision avoidance, for example. As long as the Agent in a Box was provided a collision avoidance plan, the framework will ensure that the robot avoids collisions using the provided plans, and then resume the interrupted mission.

By designing the Agent in a Box, and applying it to the application domains, all requirements of the notional agent for a mobile robot have been addressed. Table 8.1 provides a summary of the requirements of the Agent in a Box that were established in the introduction chapter and used for evaluating the state of the art. The table shows that every requirement of an agent for mobile robotics has been demonstrated in this thesis. The demonstrated requirements include the flexibility for a variety of sensors and actuators used for controlling the different robots that were used for the case studies. It also included a demonstration of the reasoning system with embedded computers, such as the Raspberry Pi, which was used for controlling the mail delivery robot. Also demonstrated were the key features of BDI identified by Bordini et. al. [12], such as the interleaving of plans, management of interruptions before the agent continues on its mission. This was an example of the agent recovering gracefully from failure. This thesis also addressed several drawbacks that Bordini et. al. identified, including the need for (1) developers to provide all the agent's plans, (2) the need for customization of the reasoner to support non-functional requirements and (3) the lack of support for generating plans at runtime [12]. The Agent in a Box addresses each of these three drawbacks:

1. The need for a developer to provide all the agent's plans.

    - The Agent in a Box provides generic plans for the behaviour common to all mobile robotic agents meaning that all the developer needs to provide are the custom plans for the robot's specific mission, method of movement, etc. This reduces the development burden for individual application domains and enforces good software engineering practice.

2. The need for customization of the reasoner.

- The Agent in a Box provides customization of the reasoning cycle, specifically for the event and option selection functions, resolving how the agent should select plans for execution. This allows the Agent in a Box to select the most appropriate plan given that mobile robots should prioritize safety first, followed by resource management, map updates, completing the mission, navigation, and movement.

3. The lack of support for generating plans at runtime.

   - The navigation framework provides the automatic generation of paths as plans at run-time. These plans consist of sequences of waypoint achievement goals that can be automatically monitored and interrupted as needed by Jason.

Table 8.1: Agent in a Box Validated Requirements.

		Included
Connecting to the Environment	Flexible for different sensors and actuators	✓
	Multiple platforms	✓
	Controlled real robot	✓
	Controlled simulated robot	✓
	ROS compatible	✓
Agent Behaviour	Uses popular BDI reasoner (ex: Jason)	✓
	Behaviour framework	✓
	Behaviour prioritization	✓
	Obstacle avoidance	✓
	Resource management	✓
	Navigation	✓

Further testing of the Agent in a Box assessed how its runtime performance compared to alternative implementations. More specifically, the responsiveness of the Agent in a Box was compared to several alternatives that did not use the same design. Although the Agent in a Box's performance was easily beat by an imperative programming implementation, it did outperform the alternative BDI agent implementations. The Agent in a Box also had improved coupling and cohesion compared to the alternatives, although it suffered in terms of cyclomatic complexity. This seems to indicate that the Agent in a Box not only enforces good design practices on the developer but also that these practices can lead to improved performance at runtime.

The work that has culminated in this thesis has generated a number of publications. These have been summarized in the introductory chapter. They include a paper which provided SAVI, which allowed for a separation of concerns between the development of complex multi-agent behaviours and simulated environments in which to test them [7]. Building on the work with SAVI, an integration of this concept with ROS was developed and integrated with an iRobot Create2 robot as a prototype for mail delivery in the Carleton University tunnels [18]. This work was extended for a special issue journal [19]. The navigation aspects of mobile robots using BDI were discussed in a subsequent publication [8]. This publication included a mechanism for the

agent to generate a route as a set of achievement goals at run-time. A journal paper on the Agent in a Box, which provides the means for applying BDI agents to mobile robotics, has been published to a special issue journal on "Intelligent Control of Mobile Robotics".

8.2 Limitations

The abilities of BDI agents and the Agent in a Box should not be overstated. Although this thesis has provided promising results, there are a number of limitations to it that need to be acknowledged. This section outlines these limitations. First, the limitations of the case studies are discussed in section 8.2.1. This focuses primarily on the engineering challenges faced when implementing the robots for the case studies, but also acknowledges the limitations of the case studies themselves. Next, the limitations of the Agent in a Box itself are discussed in section 8.2.2. Again, this focuses on a number of challenges that have been faced when working with the components of this Agent in a Box. Finally, a number of limitations to the validation of this thesis are discussed in section 8.2.3. Most notably, this includes the practical challenges of validating a framework in a thesis and in research papers. This has presented a number of challenges throughout the course of this research.

8.2.1 Limitations of the Case Studies

This section describes a number of limitations of the case studies that were implemented as part of this thesis. These include engineering challenges and limitations to the environments that were used in the case studies.

The prototype mail delivery robot, implemented with the iRobot Create (Roomba) had a number of limitations. Although this robot was intended to be a mail delivery robot for the Carleton University tunnel system, this robot has never been demonstrated outside of a *laboratory setting*. In fact, most of the demonstrations were performed in the basement of a private dwelling, a much smaller environment than the proposed tunnel system. Furthermore, the selection of the iRobot Create for the mail delivery robot presented a variety of challenges. For example, this robot's only sensors for navigation and pointing are the bumper sensor and the robot's odometer (a sensor that was found to be unreliable) unless additional sensors are attached. This reduced the usefulness of this robot for demonstrating the properties of the Agent in a Box. It made the process of robot navigation very difficult as the robot was basically blind. Although efforts were made to implement the mail delivery robot using line following, which did yield some positive results, this method of driving the robot was very cumbersome and prone to error. Another attempt was made to use Bluetooth beacons for relative navigation with the use of an external

wall following sensor. This effort also proved fruitless in the available environment. Although the beacons could be used to estimate the range of the robot to the beacon based on signal strength, the measurements from the beacons were not sufficiently consistent to localize the robot in a room the size of a residential basement. That said, in a larger environment, where positioning error on the order of several meters may not be a significant detriment to performance, such as in the Carleton tunnels, the beacons may still offer some utility to navigation. That said, the robot is still lacking a reliable sensor for sensing the direction that the robot is pointing, adding to the challenges of driving this robot.

Another limitation of the mail delivery robot is that it did not have an actuator for actually collecting and holding mail, and then delivering the mail that it was holding. Although this limitation can be addressed with some engineering effort, this was outside of the scope of this thesis. Lastly, when the mail robot was sent to the docking station, the robot would stop listening to commands sent to it over the serial connection. This meant that once the mail robot was docked for charging, the robot could not be commanded to undock from the charging station unless it was manually disconnected from the charger and power cycled. It is believed that this was a bug in the iRobot Create itself, not in a module that could be fixed. This affected the demonstration of the battery charging and recharging experiments.

The simulated autonomous car also had a number of limitations. For example, the car's LIDAR sensor only pointed straight ahead, meaning that the car could not detect obstacles to the side. This led to the risk that the car can side swipe obstacles that it should have otherwise been able to see. Additionally, the performance of the car's lane-keep assist was dramatically affected by the lighting environment. Specifically, it would get confused in shadows. The result is that the car could get lost and crash when shadows obscure the road. Lastly, the car was never demonstrated with moving obstacles, such as pedestrians wandering on the road. The car also did not have a practical way of refueling, meaning that there was no way of demonstrating the resource management behaviour in this environment.

8.2.2 Limitations of the Agent in a Box

In addition to the limitations of the case studies, there were a number of limitations to the Agent in a Box itself. These include limitations of working with BDI agents in general as well as some practical challenges of working with the Agent in a Box. These are discussed in this section.

A BDI agent can not be reasonably expected to reason about raw point cloud data in AgentSpeak, or perform image processing within the reasoner. The Agent in a Box is dependent on working with external nodes which interface with these sensors and generate perceptions in AgentSpeak which are then passed to the reasoner.

The current implementation of SAVI ROS BDI waits for perceptions to be provided before a reasoning cycle can run. Ideally, the reasoner should be allowed to run as fast as it can. This also requires that the perception translator send all perceptions to the reasoner together. If the sensors all update rather frequently, or at least at similar frequencies, this doesn't cause a significant problem. However, if there is a sensor that updates less frequently than the others, this sensor becomes a bottleneck to the entire Agent in a Box architecture, as the perception translator would be waiting for that sensor to update prior to sending perceptions to the reasoner. In turn, the reasoner would have to wait for these perceptions to arrive prior to reasoning. This problem can be avoided by taking care to have all the sensors update at a frequency that makes sense for the performance of the reasoner.

Although the Agent in a Box provides a variety of useful features, working with the various ROS nodes in the Agent in a Box architecture can be rather cumbersome, involving manually setting up and managing a large number of ROS terminals. The various ROS nodes can also be rather difficult to configure and run. This was especially true of SAVI ROS BDI, which uses the deprecated `ros-java` library for connecting the reasoner to ROS, a library that was not straightforward to install. There are a number of possible approaches to simplifying the process of working with this architecture. Some methods that may help alleviate these types of issue could be to make greater use of scripting and launch tools, migrating the project to ROS2, and containerizing SAVI ROS BDI in a Docker container to simplify the installation process.

8.2.3 Limitations of the Validation

Frameworks are difficult to validate. It is difficult to claim that a proposed framework is better than every other possible alternative. It is also difficult to claim that a proposed framework works in every possible scenario. The validation of the Agent in a Box faced these challenges as well. Although the case studies, using a variety of different types of mobile robots, provided good evidence with respect to the properties of the Agent in a Box, it does not necessarily mean that the Agent in a Box is useful for every possible mobile robot for every possible application domain. This section discusses some of the limitations of the validation of the Agent in a Box.

The validation provides test results comparing implementations of the agent for driving the simulated car as well as the navigation framework's route generation functionality in AgentSpeak, Java, and Python. These results provided a quantitative comparison of how these different methods perform at runtime. The first tests showed that the Agent in a Box benefits from improved performance compared to the alternative BDI agents. Although these tests provided confidence that the use of the Agent in a Box can yield attractive properties it also showed that the use of imperative programming can result in even better runtime performance. However,

these alternatives were designed specifically for the test scenario, providing only a subset of the Agent in a Box functionality. It is unlikely that the addition of more functionality to the alternatives would result in improved runtime performance, therefore this limitation is unlikely to have resulted in any change in conclusions. The profile tests focused on assessing if the agent's deliberation took longer than the processing of the perceptions and actions in the reasoning cycle. Although it was found that the deliberation was not a major bottleneck, the use of the profiler resulted in a reduction in performance of the agent. This meant that, although the profiler provided quantitative results, these were not considered to be an accurate representation of the true execution time of the reasoning cycle's functions. This problem can be addressed by removing the profiler and instrumenting the code, which was done for several of the experiments, however it looses the flexibility and detail that comes with using a full fledged profiler.

Another aspect of the validation was an assessment of the software engineering properties of the framework. This included assessing the coupling, cohesion, and cyclomatic complexity of an agent implemented with the Agent in a Box and comparing it to the alternative implementations. These assessments provided useful insight with respect to the maintainability of agents implemented with the Agent in a Box as well as how difficult it may be for a developer to work with the Agent in a Box. That said, the alternative implementations did not include all possible features that the Agent in a Box provides, although, as with the performance experiments, it is anticipated that adding additional functionality to the alternatives would not result in improved software properties. Also, the selected metrics of coupling, cohesion, and cyclometric complexity are not the be-all end-all metrics for measuring code maintainability, although they are certainly an improvement over the use of lines of code as a software metric. Lastly on the subject of the software engineering properties, although these properties can be used to assess how a developer might work with the Agent in a Box, a developer experience study was not completed as part of this thesis.

Lastly, the Agent in a Box was not tested in scenarios where there were multiple conflicting obstacles. In fact, the obstacle avoidance maneuvers written for the case studies were all hard-coded based on observations of the prototyping environments used. For example, the car's collision avoidance maneuver involved either swerving the car to the left or stopping to avoid a parked car. This maneuver did not involve any check to see if there was an obstacle to the left of the car. A more appropriate collision avoidance behaviour would involve a more fulsome use of the car's sensors, possibly using the ROS navigation stack to plot a route around obstacles.

8.3 Recommendation and Future Work

Despite the limitations discussed in the previous section, this thesis has demonstrated the use of BDI for autonomous mobile robots, which provides a means for guaranteeing mobile robot behaviour is rational, through the use of the features of Jason's BDI reasoner and the AgentSpeak language. This thesis has demonstrated that the Agent in a Box for mobile robotics provides behaviours that are relatively easy to understand and apply to various application domains. The Agent in a Box provides the elements needed for autonomous mobile robots in general, meaning that the developer for specific application domains only needs to focus on the elements specific to that domain. It also showed that the use of the Agent in a Box imposes good design practices on the developer, resulting in software with improved coupling and cohesion compared to several alternatives. These design practices also resulted in improved performance when compared to other BDI agents that did not use the Agent in a Box. Unfortunately, the Agent in a Box was not able to provide runtime performance on par with an agent implemented with imperative programming.

This thesis was successful at demonstrating that BDI agents implemented using the Agent in a Box can be successfully applied for controlling autonomous robots. This claim is supported by the well received peer-reviewed publications that have been written over the course of the work on this thesis. Therefore this thesis recommends the use of BDI agents, and more specifically the Agent in a Box, for implementing autonomous mobile robots in applications where the movement of the agent and the desired obstacle avoidance behaviour can be represented symbolically and that the performance difference between it and equivalent behaviour implemented using imperative programming can be tolerated.

Building on the work from this thesis, there are several opportunities for additional contributions. For example, the performance tests found that although the Agent in a Box was able to outperform alternative BDI agent implementations, it was not able to keep up with an agent implemented using imperative programming. This implies that BDI could benefit from improved runtime performance. One approach for improving its performance could be to explore hardware acceleration. This idea is inspired in part by the performance boost found with neural networks by processing them on Graphics Processing Units (GPUs) rather than general purpose CPUs. It may be possible for the reasoning cycle to be modified to make use of programmable hardware, such as an Field-Programmable Gate Array (FPGA), to implement the processing of rules in hardware at gate speeds. A possible starting point may be to shift the logic associated with some rules and the context check to hardware, as this was found to be a main contributor to the difference in performance between the Agent in a Box agent and the alternative BDI agents.

Another area with potential for additional contribution in this area is to further explore

how a BDI agent could benefit from the integration of a full fledged planner, rather than being totally dependent on prewritten plans. The Agent in a Box has shown that there is potential for BDI agents to benefit from integrated planning by using an AgentSpeak implementation of A* search to generate the paths for the agent to its destination as plans that can be monitored and suspended as needed. The next step to this would be to explore how else a BDI agent could benefit from or be detracted by being integrated with a planner responsible for generating more complex plans.

Finally, another future work would be to make the Agent in a Box a true Agent in a *Box*, meaning a device that is already installed on a piece of hardware and that can be connected to anything that is ROS enabled. This hardware could include the possible hardware optimization and planners discussed in the previous two paragraphs. It could also include the use of machine learning as part of the sensor and actuator nodes, which could compliment the use of BDI reasoning.

Chapter 9

Bibliography

[1] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal on Robotics and Automation*, vol. 2, pp. 14–23, March 1986.

[2] M. Wooldridge, *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd ed., 2009.

[3] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. The Atrium, Southern Gate, Chichester, West Sussex, England: John Wiley &; Sons Ltd., 2007.

[4] R. H. Bordini, J. F. Hübner, and M. Wooldridge, "Programming Multi-Agent Systems in AgentSpeak Using Jason (Lecture Slides)." `http://jason.sourceforge.net/jBook/SlidesJason.pdf`, 2007. Accessed: 2019-06-27.

[5] Marder-Eppstein et.al., "navigation." `http://wiki.ros.org/navigation`. Accessed: 2021-06-14.

[6] M. S. Menegol, J. F. Hübner, and L. B. Becker, "Evaluation of Multi-agent Coordination on Embedded Systems," in *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection* (Y. Demazeau, B. An, J. Bajo, and A. Fernández-Caballero, eds.), (Cham), pp. 212–223, Springer International Publishing, 2018.

[7] A. Davoust, P. Gavigan, C. Ruiz-Martin, G. Trabes, B. Esfandiari, G. Wainer, and J. James, "An Architecture for Integrating BDI Agents with a Simulation Environment," in *Engineering Multi-Agent Systems* (L. A. Dennis, R. H. Bordini, and Y. Lespérance, eds.), (Cham), pp. 67–84, Springer International Publishing, 2020.

[8] P. Gavigan and B. Esfandiari, "Bdi for autonomous mobile robot navigation," in *Engineering Multi-Agent Systems* (N. Alechina, M. Baldoni, and B. Logan, eds.), (Cham), pp. 137–155, Springer International Publishing, 2022.

[9] M. Wooldridge, *Reasoning about rational agents.* Cambridge Massachusetts: MIT Press, 2003.

[10] M. Bratman, *Intention, plans, and practical reason*, vol. 10. Cambridge, Mass: Harvard University Press, 1987.

[11] A. S. Rao and M. P. Georgeff, "BDI agents: From theory to practice," in *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, (San Francisco, California, USA.), pp. 312–319, 1995.

[12] R. H. Bordini, A. El Fallah Seghrouchni, K. Hindriks, B. Logan, and A. Ricci, "Agent programming in the cognitive era," *Autonomous Agents and Multi-Agent Systems*, vol. 34, 2020.

[13] J. F. Hübner and R. H. Bordini, "Jason: a Java-based interpreter for an extended version of AgentSpeak." `http://jason.sourceforge.net`. Accessed: 2019-02-16.

[14] A. S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language," in *Agents Breaking Away* (W. Van de Velde and J. W. Perram, eds.), (Berlin, Heidelberg), pp. 42–55, Springer Berlin Heidelberg, 1996.

[15] N. Xie, G. Ras, M. V. Gerven, and D. Doran, "Explainable deep learning: A field guide for the uninitiated," *ArXiv*, vol. abs/2004.14545, 2020.

[16] V. J. Koeman, L. A. Dennis, M. Webster, M. Fisher, and K. Hindriks, "The "why did you do that?" button: Answering why-questions for end users of robotic systems," in *Engineering Multi-Agent Systems* (L. A. Dennis, R. H. Bordini, and Y. Lespérance, eds.), (Cham), pp. 152–172, Springer International Publishing, 2020.

[17] L. Dennis and N. Oren, "Explaining bdi agent behaviour through dialogue," in *Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021)* (U. Endriss, A. Nowe, F. Dignum, and A. Lomuscio, eds.), pp. 429–437, International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2021.

[18] C. Onyedinma, P. Gavigan, and B. Esfandiari, "Toward Campus Mail Delivery Using BDI," in Proceedings of the First Workshop on *Agents and Robots for reliable Engineered Autonomy,* Virtual event, 4th September 2020 (R. C. Cardoso, A. Ferrando, D. Briola,

C. Menghi, and T. Ahlbrecht, eds.), vol. 319 of *Electronic Proceedings in Theoretical Computer Science*, pp. 127–143, Open Publishing Association, 2020.

[19] C. Onyedinma, P. Gavigan, and B. Esfandiari, "Toward Campus Mail Delivery Using BDI," *Journal of Sensor and Actuator Networks*, vol. 9, p. 56, Dec 2020.

[20] P. Gavigan and B. Esfandiari, "Agent in a box: A framework for autonomous mobile robots with beliefs, desires, and intentions," *Electronics*, vol. 10, no. 17, 2021.

[21] P. Gavigan and B. Esfandiari, "Quantifying the relationship between software design principles and performance in jason: a case study with simulated mobile robots," in *Engineering Multi-Agent Systems* (A. Chopra, J. Dix, and R. Zalila-Wenkstern, eds.), 2022.

[22] R. E. Johnson, "Documenting Frameworks Using Patterns," *SIGPLAN Not.*, vol. 27, p. 63–76, 10 1992.

[23] D. Riehle, *Framework Design: A Role Modeling Approach*. PhD thesis, SWISS FEDERAL INSTITUTE OF TECHNOLOGY, ZURICH, 2000.

[24] M. Fowler, "Inversion Of Control." `https://martinfowler.com/bliki/InversionOfControl.html`. Accessed: 2021-05-10.

[25] A. Turner, "JUnit: A Cook's Tour." `https://www.geog.leeds.ac.uk/people/a.turner/src/andyt/java/grids/lib/junit-3.8.1/doc/cookstour/cookstour.htm`. Accessed: 2021-05-05.

[26] Y. Shoham, "Agent-oriented programming," *Artificial Intelligence*, vol. 60, no. 1, pp. 51 – 92, 1993.

[27] AOSGroup, "JACK." `http://www.aosgrp.com/products/jack/`. Accessed: 2019-02-04.

[28] M. Aschermann, P. Kraus, and J. P. Müller, "LightJason: A BDI Framework Inspired by Jason," Tech. Rep. IfI Technical Report IfI-16-04, Department of Computer Science, TU Clausthal, Clausthal-Zellerfeld, Germany, October 2016.

[29] "LightJason." `https://lightjason.org/`. Accessed: 2019-03-18.

[30] B. Muller and L. Dennis, "Gwendolen: A BDI Language for Verifiable Agents," in *AISB 2008 Symposium: Logic and the Simulation of Interaction and Reasoning*, Apr. 2008.

[31] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "JaCaMo Project." `http://jacamo.sourceforge.net/`. Accessed: 2019-05-16.

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

[33] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, (Shanghai, China), May 9-13 2011.

[34] NA, "Point Cloud Library." `https://pointclouds.org/`. Accessed: 2021-05-06.

[35] D. Coleman, I. Şucan, S. Chitta, and N. Correll, "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study," *Journal of Software Engineering for Robotics*, vol. 5, no. 2, pp. 3–16, 2014.

[36] I. A. Sucan and S. Chitta, "MoveIt." `https://moveit.ros.org/`. Accessed: 2021-05-06.

[37] Marder-Eppstein et.al., "move_base." `http://wiki.ros.org/move_base`. Accessed: 2021-04-19.

[38] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The Office Marathon: Robust Navigation in an Indoor Office Environment," in *International Conference on Robotics and Automation*, 2010.

[39] G. Granosik, K. Andrzejczak, M. Kujawinski, R. Bonecki, L. Chlebowicz, B. Krysztofiak, K. Mirecki, and M. Gawryszewski, "Using robot operating system for autonomous control of robots in eurobot, ERC and robotour competitions," *Acta Polytechnica CTU Proceedings*, vol. 6, p. 11, 11 2016.

[40] M. Dastani, J. Dix, P. Novák, and J. Hübner, "Multi-Agent Programming Contest." `https://multiagentcontest.org/2019/`. Accessed: 2019-05-28.

[41] P. Wallis, R. Ronnquist, D. Jarvis, and A. Lucas, "The automated wingman - Using JACK intelligent agents for unmanned autonomous vehicles," in *Proceedings, IEEE Aerospace Conference*, vol. 5, pp. 5–5, March 2002.

[42] S. Karim and C. Heinze, "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, (New York, NY, USA), pp. 19–26, ACM, 2005.

[43] J. Boyd, "The essence of winning and losing." Presentation, 2010 Archival version available [online]: `http://pogoarchives.org/m/dni/john_boyd_compendium/essence_of_winning_losing.pdf`, 1995. Accessed: 2022-01-25.

[44] M. S. Menegol, "UAVExperiments." https://github.com/msmenegol/UAVExperiments. Accessed: 2019-05-24.

[45] M. S. Menegol, "vooAgente4Wp." https://drive.google.com/file/d/0B7EcHgES6He8VEtwR0xPZjdBbk0/view. Accessed: 2019-05-08.

[46] G. Rezende and J. F. Hubner, "Jason-ROS." https://github.com/jason-lang/jason-ros. Accessed: 2019-05-24.

[47] G. Rezende, "MAS-UAV." https://github.com/Rezenders/MAS-UAV. Accessed: 2019-05-24.

[48] M. G. Morais, "rason." https://github.com/mgodoymorais/rason/tree/master/jason_ros.

[49] F. Meneguzzi and R. Wesz, "Jason ROS Releases." https://github.com/lsa-pucrs/jason-ros-releases.

[50] I. Calaça, T. Krausburg, and R. C. Cardoso, "JROS." https://github.com/smart-pucrs/JROS.

[51] R. Wesz, "Integrating Robot Control Into The AgentSpeak(L) Programming Language," Master's thesis, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil, 2015.

[52] L. Fichera, F. Messina, G. Pappalardo, and C. Santoro, "A Python framework for programming autonomous robots using a declarative approach," *Science of Computer Programming*, vol. 139, pp. 36 – 55, 2017.

[53] NA, "Eurobot: International Students Robotic Contest." http://www.eurobot.org/. Accessed: 2019-07-15.

[54] NA, "Unict Team website." http://unict-team.dmi.unict.it/. Accessed: 2019-07-15.

[55] C. E. Pantoja, M. F. Stabile, N. M. Lazarin, and J. S. Sichman, "ARGO: An Extended Jason Architecture that Facilitates Embedded Robotic Agents Programming," in *Engineering Multi-Agent Systems* (M. Baldoni, J. P. Müller, I. Nunes, and R. Zalila-Wenkstern, eds.), (Cham), pp. 136–155, Springer International Publishing, 2016.

[56] N. M. Lazarin and C. E. Pantoja, "A robotic-agent platform for embedding software agents using raspberry pi and arduino boards," *9th Software Agents, Environments and Applications School*, 2015.

[57] M. F. Stabile and J. S. Sichman, "Evaluating Perception Filters in BDI Jason Agents," in *2015 Brazilian Conference on Intelligent Systems (BRACIS)*, pp. 116–121, Nov 2015.

[58] L. A. Dennis, J. M. Aitken, J. Collenette, E. Cucco, M. Kamali, O. McAree, A. Shaukat, K. Atkinson, Y. Gao, S. M. Veres, and M. Fisher, "Agent-Based Autonomous Systems and Abstraction Engines: Theory Meets Practice," in *Towards Autonomous Robotic Systems* (L. Alboul, D. Damian, and J. M. Aitken, eds.), (Cham), pp. 75–86, Springer International Publishing, 2016.

[59] R. C. Cardoso, A. Ferrando, L. A. Dennis, and M. Fisher, "An interface for programming verifiable autonomous agents in ros," in *Multi-Agent Systems and Agreement Technologies* (N. Bassiliades, G. Chalkiadakis, and D. de Jonge, eds.), (Cham), pp. 191–205, Springer International Publishing, 2020.

[60] R. Toris et.al., "rosbridge-depricated." `http://wiki.ros.org/rosbridge-depricated`. Accessed: 2022-01-26.

[61] Robot Web Tools, "rosbridge_suite." `http://wiki.ros.org/rosbridge_suite`. Accessed: 2022-01-26.

[62] M. T. Hama, "uavas." `https://github.com/marcelohama/uavas-platform`. Accessed: 2019-05-28.

[63] A. Bojarpour, "RoboticCPS." `https://github.com/alibojar/RoboticCPS`. Accessed: 2019-05-28.

[64] C. Wei and K. V. Hindriks, "An Agent-Based Cognitive Robot Architecture," in *Programming Multi-Agent Systems* (M. Dastani, J. F. Hübner, and B. Logan, eds.), (Berlin, Heidelberg), pp. 54–71, Springer Berlin Heidelberg, 2013.

[65] J. Han, C. Wang, and G. Yi, "Cooperative control of UAV based on Multi-Agent System," in *2013 IEEE 8th Conference on Industrial Electronics and Applications (ICIEA)*, pp. 96–101, June 2013.

[66] M. H. Dominguez, S. Nesmachnow, and J. Hernández-Vega, "Planning a drone fleet using artificial intelligence for search and rescue missions," in *2017 IEEE XXIV International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*, pp. 1–4, Aug 2017.

[67] I. Rüb and B. Dunin-Kęplicz, "Bdi model of connected and autonomous vehicles," in *Computational Collective Intelligence* (N. T. Nguyen, R. Chbeir, E. Exposito, P. Aniorté, and B. Trawiński, eds.), (Cham), pp. 181–195, Springer International Publishing, 2019.

[68] P. Ehlert, "Intelligent Driving Agents: The Agent Approach to Tactical Driving in Autonomous Vehicles and Traffic Simulation," Master's thesis, Delft University of Technology, January 2001.

[69] E. Vinitsky, A. Kreidieh, L. L. Flem, N. Kheterpal, K. Jang, C. Wu, F. Wu, R. Liaw, E. Liang, and A. M. Bayen, "Benchmarks for reinforcement learning in mixed-autonomy traffic," in *Proceedings of The 2nd Conference on Robot Learning* (A. Billard, A. Dragan, J. Peters, and J. Morimoto, eds.), vol. 87 of *Proceedings of Machine Learning Research*, pp. 399–409, PMLR, 29–31 Oct 2018.

[70] M. Aschermann, S. Dennisen, P. Kraus, and J. P. Müller, "LightJason, a Highly Scalable and Concurrent Agent Framework: Overview and Application (Demonstration)," in *Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018)* (M. Dastani, G. Sukthankar, E. André, and S. Koenig, eds.), pp. 1794–1796, 2018. /news/2018-07-aamas.

[71] K. Gillespie, M. Molineaux, M. Floyd, S. Vattam, and D. W. Aha, "Goal Reasoning for an Autonomous Squad Member," in *2015 Annual Conference on Advances in Cognitive Systems: Workshop on Goal Reasoning*, 2015.

[72] M. W. Floyd, J. Karneeb, P. Moore, and D. W. Aha, "A Goal Reasoning Agent for Controlling UAVs in Beyond-Visual-Range Air Combat," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, p. 4714–4721, AAAI Press, 2017.

[73] M. Pěchouček and V. Mařík, "Industrial Deployment of Multi-agent Technologies: Review and Selected Case Studies," *Autonomous Agents and Multi-Agent Systems*, vol. 17, pp. 397–431, Dec. 2008.

[74] P. Hofmann, P. Lettmayer, T. Blaschke, M. Belgiu, S. Wegenkittl, R. Graf, T. J. Lampoltshammer, and V. Andrejchenko, "Towards a framework for agent-based image analysis of remote-sensing data," *International Journal of Image and Data Fusion*, vol. 6, no. 2, pp. 115–137, 2015. PMID: 27721916.

[75] D. Singh, L. Padgham, and B. Logan, "Integrating BDI agents with agent-based simulation platforms," *Autonomous Agents and Multi-Agent Systems*, vol. 30, pp. 1050–1071, Nov 2016.

[76] A. Davoust, P. Gavigan, C. R.-M. A. G. Trabes, B. Esfandiari, G. Wainer, and J. James, "Simulated Autonomous Vehicle Infrastructure." `https://github.com/NMAI-lab/SAVI`. Accessed: 2019-02-19.

[77] P. Gavigan, "SAVI_ROS_BDI." https://github.com/NMAI-lab/savi_ros_bdi. Accessed: 2020-02-18.

[78] P. Gavigan, "agent_in_a_box_agent." https://github.com/NMAI-lab/agent_in_a_box_agent. Accessed: 2022-01-19.

[79] J. F. Hubner, "Jason Search Demo." https://github.com/jason-lang/jason/tree/master/demos/search. Accessed: 2021-02-19.

[80] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. USA: Prentice Hall Press, 3rd ed., 2009.

[81] S. Russell and P. Norvig, "AIMA3e-Java (JDK 8+)." https://github.com/aimacode/aima-java. Accessed: 2021-02-19.

[82] jrialland, "python-astar." https://github.com/jrialland/python-astar. Accessed: 2020-08-24.

[83] P. Gavigan, "Jason Mobile Agent ROS." https://github.com/NMAI-lab/jason_mobile_agent_ros. Accessed: 2021-02-19.

[84] P. Gavigan, "Jason Mobile Agent." https://github.com/NMAI-lab/jasonMobileAgent. Accessed: 2021-02-19.

[85] P. Gavigan, "Agent in a Box Demo - Grid Environment." https://youtu.be/bsr3K4U3wd8. Accessed: 2021-02-19.

[86] "AirSim." https://github.com/Microsoft/AirSim. Accessed: 2019-03-27.

[87] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," in *Field and Service Robotics*, 2017.

[88] S. Yao, "Autonomous-driving vehicle test technology based on virtual reality," *The Journal of Engineering*, vol. 2018, pp. 1768–1771, 08 2018.

[89] P. Gavigan, "AirSim Navigating Car." https://github.com/NMAI-lab/AirSimNavigatingCar. Accessed: 2021-02-19.

[90] P. Gavigan, "AirSim Car BDI Agent." https://youtu.be/yX20gJjjbMg. Accessed: 2021-02-19.

[91] P. Gavigan, "Agent in a Box Demo - Car Lane Keep and Obstacle Avoidance." https://youtu.be/tvqkNnpKIPo. Accessed: 2021-04-05.

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

[93] J. Perron, "create_autonomy." http://wiki.ros.org/create_autonomy. Accessed: 2020-03-08.

[94] P. Gavigan and C. Onyedinma, "saviRoomba." https://github.com/NMAI-lab/saviRoomba. Accessed: 2020-05-09.

[95] P. Gavigan, "Mail agent - mail mission." https://youtu.be/4nVOVI1GJOM. Accessed: 2021-07-19.

[96] P. Gavigan, "Mail agent - collision recovery." https://youtu.be/bKHR-DaXZq0. Accessed: 2021-07-19.

[97] P. Gavigan, "Mail agent - docking to recharge battery." https://youtu.be/hvq_vduv-OM. Accessed: 2021-07-19.

[98] P. Gavigan, "Jason mobile agent - AgentSpeak Navigation." https://youtu.be/ooB15Ve54sI. Accessed: 2021-02-19.

[99] P. Gavigan, "Jason mobile agent - Internal Action Navigation." https://youtu.be/ooB15Ve54sI. Accessed: 2021-02-19.

[100] P. Gavigan, "Jason mobile agent - Environment Supported Navigation." https://youtu.be/r0CiwjxapZA. Accessed: 2021-02-19.

[101] D. J. Barnes and M. Kolling, *Objects First with Java: A Practical Introduction Using BlueJ*. Boston: Pearson, 2017.

[102] S. Kramer and H. Kaindl, "Coupling and cohesion metrics for knowledge-based systems using frames and rules," *ACM Trans. Softw. Eng. Methodol.*, vol. 13, p. 332–358, jul 2004.

[103] A. Serebrenik, T. Schrijvers, and B. Demoen, "Improving prolog programs: Refactoring for prolog," in *ICLP*, 2004.

[104] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[105] T. T. Moores, "Applying complexity measures to rule-based prolog programs," *Journal of Systems and Software*, vol. 44, no. 1, pp. 45–52, 1998.

Appendix A

Comparison to Finite State Machine (FSM)

FSMs provide a useful and powerful method for implementing embedded software and are good at capturing the behaviour of systems that are both event-driven and depend on internal memory. When using this approach, an agent responds to different events based on which state it is in. Since the FSM requires that every possible event be handled in every possible state, this provides a simple and less bug-prone method for defining the complete functionality of the agent. The goal of comparing the development of plans and rules in AgentSpeak to the development of an FSM was to find any parallels between these two methods. Any parallels could indicate a possible design approach that can be used for developing new AgentSpeak software. The comparison focused on the autonomous car driving agent, specifically the collision avoidance and road following behaviour used for the software properties assessments in section 7.3.

The state table for the car controlling FSM agent, which can drive down the street and avoid a collision, is provided in table A.1. The same agent is shown graphically in figure A.1. The agent was designed to have three states, based on the speed setting for the cruise control for the car. This aligned with a belief that the BDI agent maintained with respect to the cruise control setting, were used to prevent the agent from needlessly commanding the same speed setting for the cruise control repeatedly. The next consideration was the events that the FSM would monitor. These included the proximity to the waypoint, whether the lane-keep assist was able to provide a recommended steering setting or not, and whether or not there was an obstacle observed. These events were in alignment with the rules that were defined for for the BDI agents to support decision making. Next, the actions were for setting the speed for the cruise controller or the steering. The steering was either set to a constant value, for the collision avoidance case,

or to either compass steering or lane-keep steering. The specific values of the compass steering setting and lane-keep steering setting are calculated in the environment, as is the generation of the value of the waypoint event, whether it be at the waypoint, near the waypoint, or far from the waypoint. This means that the management of the location of the next waypoint and the path to the destination both needed to occur in the environment.

Table A.1: State Transition Table for the Simulated Car Agent.

#	Start State	Event			Action	New State
		Waypoint (at/near/far)	LKA (T/F)	Obstacle (T/F)		
1	SpeedSet0	at	F	F	–	SpeedSet0
2	SpeedSet0	at	F	T	–	SpeedSet0
3	SpeedSet0	at	T	F	–	SpeedSet0
4	SpeedSet0	at	T	T	–	SpeedSet0
5	SpeedSet0	near	F	F	setSpeed(3)	SpeedSetSlow
6	SpeedSet0	near	F	T	setSpeed(3)	SpeedSetSlow
7	SpeedSet0	near	T	F	setSpeed(3)	SpeedSetSlow
8	SpeedSet0	near	T	T	setSpeed(3)	SpeedSetSlow
9	SpeedSet0	far	F	F	setSpeed(8)	SpeedSetFull
10	SpeedSet0	far	F	T	setSpeed(8)	SpeedSetFull
11	SpeedSet0	far	T	F	setSpeed(8)	SpeedSetFull
12	SpeedSet0	far	T	T	setSpeed(8)	SpeedSetFull
13	SpeedSetFull	at	F	F	setSpeed(0)	SpeedSet0
14	SpeedSetFull	at	F	T	setSpeed(0)	SpeedSet0
15	SpeedSetFull	at	T	F	setSpeed(0)	SpeedSet0
16	SpeedSetFull	at	T	T	setSpeed(0)	SpeedSet0
17	SpeedSetFull	near	F	F	setSpeed(3)	SpeedSetSlow
18	SpeedSetFull	near	F	T	steer(-0.3)	SpeedSetFull
19	SpeedSetFull	near	T	F	setSpeed(3)	SpeedSetSlow
20	SpeedSetFull	near	T	T	steer(-0.3)	SpeedSetFull
21	SpeedSetFull	far	F	F	compassSteer	SpeedSetFull
22	SpeedSetFull	far	F	T	steer(-0.3)	SpeedSetFull
23	SpeedSetFull	far	T	F	lkaSteer	SpeedSetFull
24	SpeedSetFull	far	T	T	steer(-0.3)	SpeedSetFull
25	SpeedSetSlow	at	F	F	setSpeed(0)	SpeedSet0
26	SpeedSetSlow	at	F	T	setSpeed(0)	SpeedSet0
27	SpeedSetSlow	at	T	F	setSpeed(0)	SpeedSet0
28	SpeedSetSlow	at	T	T	setSpeed(0)	SpeedSet0
29	SpeedSetSlow	near	F	F	compassSteer	SpeedSetSlow
30	SpeedSetSlow	near	F	T	steer(-0.3)	SpeedSetSlow
31	SpeedSetSlow	near	T	F	lkaSteer	SpeedSetSlow
32	SpeedSetSlow	near	T	T	steer(-0.3)	SpeedSetSlow
33	SpeedSetSlow	far	F	F	setSpeed(8)	SpeedSetFull
34	SpeedSetSlow	far	F	T	steer(-0.3)	SpeedSetSlow
35	SpeedSetSlow	far	T	F	setSpeed(8)	SpeedSetFull
36	SpeedSetSlow	far	T	T	steer(-0.3)	SpeedSetSlow

The assessment showed that there was a relationship between the events that were used by the FSM and the plans used by the BDI agent designed with the Agent in a Box. This

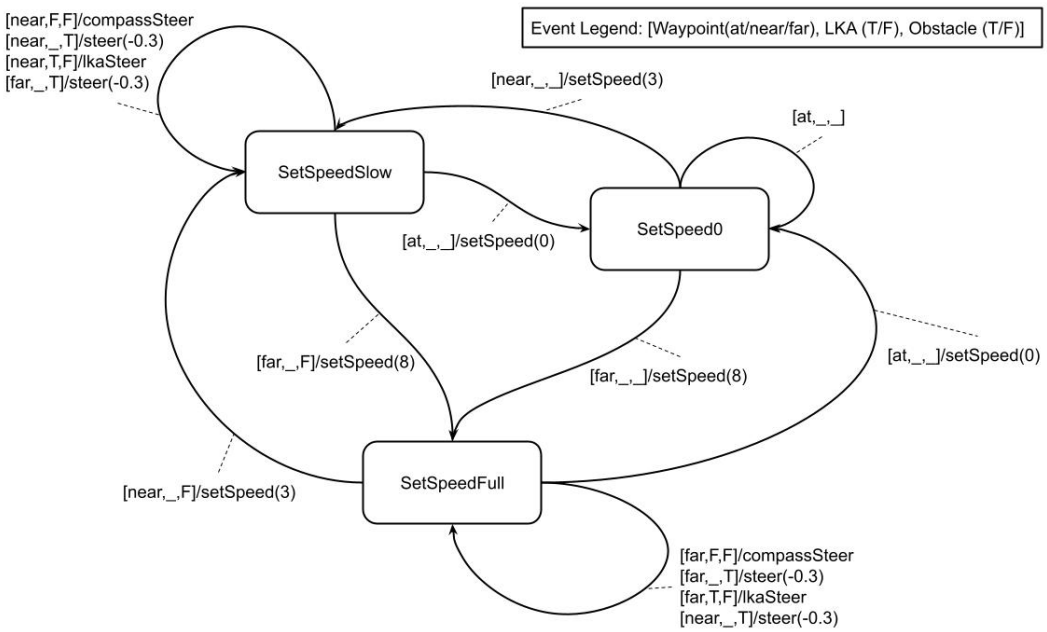


Figure A.1: Finite State Machine Diagram for the Simulated Car Agent.

implies that a developer using the Agent in a Box, or AgentSpeak in general, could apply the methods used for designing a FSM when designing the rules that needed for supporting the needed AgentSpeak plans. Furthermore, the states themselves aligned with the beliefs that the agent needs to maintain. In the case of this assessment, the states were associated with the speed setting that had been sent to the cruise controller. These observations may point to a method of guiding new developers to write idiomatic AgentSpeak code by considering the use of a FSM designing their agent. That said, the use of BDI enables the implementation of these rules and beliefs internally in the agent. The FSM, for example, was heavily dependent on the environment to support its knowledge of a destination.

Appendix B

Acronyms

AI Artificial Intelligence.

AJPF Agent Java PathFinder.

AOP Agent Oriented Programming.

AREA Agents and Robots for reliable Engineered Autonomy.

BDI Beliefs-Desires-Intentions.

CPU Central Processing Unit.

CRSNG Conseil de recherches en sciences naturelles et en génie du Canada.

DRDC Defence Research and Development Canada.

EBNF Extended Backus–Naur Form.

EMAS International Workshop on Engineering Multi-Agent Systems.

FPGA Field-Programmable Gate Array.

FSM Finite State Machine.

GA Genetic Algorithm.

GNSS Global Navigation Satellite System.

GPS Global Positioning System.

GPU Graphics Processing Unit.

LIDAR laser imaging, detection, and ranging.

MAS Multi Agent System.

NSERC Natural Sciences and Engineering Research Council of Canada.

OODA Observe Orient Decide Act.

PROFETA Python RObotic Framework for dEsigning sTrAtegies.

RAM Random Access Memory.

RDDC Recherche et développement pour la défense Canada.

ROS Robot Operating System.

SAVI Simulated Autonomous Vehicle Infrastructure.

SITL Software in the Loop.

SLAM Simultaneous Localization and Mapping.

UAV Unmanned Aerial Vehicle.

XML Extensible Markup Language.