FEDERAL UNIVERSITY OF SANTA CATARINA

DEPARTMENT OF AUTOMATION AND SYSTEMS, CENTER OF TECHNOLOGY

POSTGRADUATE PROGRAM IN AUTOMATION AND SYSTEMS ENGINEERING

Gustavo Rezende Silva

ACTIVE PERCEPTION WITHIN BDI AGENTS REASONING CYCLE WITH APPLICATIONS IN MOBILE ROBOTS

Florianópolis

2020

Gustavo Rezende Silva

ACTIVE PERCEPTION WITHIN BDI AGENTS REASONING CYCLE WITH APPLICATIONS IN MOBILE ROBOTS

Monograph submitted to the Post-graduate Program in Automation and Systems Engineering of Federal University of Santa Catarina for degree acquirement in Master in Automation and Systems Engineering.

Supervisor: Prof. Jomi Fred Hübner, Phd.
Co-Supervisor: Prof. Leandro Buss Becker, Phd.

Florianópolis

2020

Cataloging at source by the University Library of the Federal University of Santa Catarina.

File compiled at 23:46h of the day Monday 8th March, 2021.

Gustavo Rezende Silva
Active Perception within BDI Agents Reasoning Cycle with Applications in Mobile Robots / Gustavo Rezende Silva; Supervisor, Prof. Jomi Fred Hübner, Phd. ; Co-Supervisor, Prof. Leandro Buss Becker, Phd. - Florianópolis, 23:46, 13 of October of 2020.
78 p.

Monograph - Federal University of Santa Catarina, Department of Automation and Systems, Center of Technology, Postgraduate Program in Automation and Systems Engineering.

Includes references

1. BDI Agents. 2. Mobile robotics. 3. Active Perception. 4. UAV. I. Prof. Jomi Fred Hübner, Phd. II. Prof. Leandro Buss Becker, Phd. III. Postgraduate Program in Automation and Systems Engineering IV. Active Perception within BDI Agents Reasoning Cycle with Applications in Mobile Robots

Gustavo Rezende Silva

ACTIVE PERCEPTION WITHIN BDI AGENTS REASONING CYCLE WITH APPLICATIONS IN MOBILE ROBOTS

This Monograph was considered appropriate to get the Master in Automation and Systems Engineering, and it was approved by the Postgraduate Program in Automation and Systems Engineering of Department of Automation and Systems, Center of Technology of Federal University of Santa Catarina.

Prof. Jomi Fred Hübner, Phd.
Federal University of Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Mestre em Engenharia de Automação e Sistemas.

ICP-Edu
Documento assinado digitalmente
Werner Kraus Junior
Data: 09/03/2021 06:22:41-0300
CPF: 531.085.239-53
Verifique as assinaturas em https://v.ufsc.br

Prof. Werner Kraus Jr., Phd.
Coordinator of Postgraduate Program in
Automation and Systems Engineering

ICP-Edu
Documento assinado digitalmente
Jomi Fred Hubner
Data: 08/03/2021 20:57:45-0300
CPF: 596.334.769-68
Verifique as assinaturas em https://v.ufsc.br

Prof. Jomi Fred Hübner, Phd.
Supervisor
Federal University of Santa Catarina

ICP-Edu
Documento assinado digitalmente
Leandro Buss Becker
Data: 25/03/2021 05:40:52-0300
CPF: 688.357.870-68
Verifique as assinaturas em https://v.ufsc.br

Prof. Leandro Buss Becker, Phd.
Co-Supervisor
Federal University of Santa Catarina

Florianópolis, 13 of October of 2020.

This work is dedicated to all humanity, I hope that somehow the knowledge produced in this project will contribute to science and the world.

ACKNOWLEDGEMENTS

I am grateful to the Federal University of Santa Catarina, to the Department of Automation and Systems, to the Graduate Program in Automation and Systems Engineering, to the teaching and administrative staff for the opportunity to take the master's course. In particular, I am grateful to my advisors Leandro B. Becker and Jomi F. Hubner for their support, guidance, and dedication. I also thank my family, girlfriend and friends who supported me and were by my side during my journey up to this moment. I also extend my thanks to everyone who somehow passed through my life and contributed to me becoming the person I have become. My sincere thanks!

RESUMO

Em sistemas multi agentes o principal processo responsável por obter informações sobre o ambiente é a percepção, geralmente este processo é realizado passivamente independente do estado interno do agente. Entretanto, principalmente quando inseridos em ambientes reais, um problema frequente é que os agentes têm percepção parcial do ambiente, não conseguindo perceber tudo aquilo que é necessário. Para contornar este problema, uma solução é ativamente realizar ações para perceber o que é de interesse do agente, ao invés de apenas perceber passivamente o que está disponível no ambiente, por exemplo, em um sistema de visão computacional, a câmera pode ser reposicionada para ter uma melhor visão de um objeto. Com isso, este trabalho tem como objetivo elaborar um modelo de percepção ativa integrado com o ciclo de raciocínio de agentes BDI. Ainda, um dos objetivos é testar a percepção ativa em ambientes os mais próximos de reais possíveis, em razão disso foi desenvolvido uma arquitetura embarcada que visa promover a utilização de agentes cognitivos em cooperação com o *The Robotic Operating System* para programar a inteligência de robôs. Foram realizados experimentos utilizando agentes BDI com ROS para comandar veículos aéreos não tripulados para analisar os benefícios e impactos de se utilizar agentes cognitivos e percepção ativa para programar a inteligência de robôs.

Palavras-chaves: Agentes BDI. Robótica Móvel. Percepção ativa. VANT.

RESUMO EXPANDIDO

INTRODUÇÃO

Técnicas de sistemas multiagentes (SMA) parecem ser uma abordagem vantajosa para programar a parte cognitiva de robôs, uma vez que oferecem ferramentas teóricas e práticas para o desenvolvimento de sistemas autônomos. Em SMA um dos mecanismos que os agentes possuem para reunir conhecimento sobre o estado do ambiente é por meio de um processo denominado percepção, este é responsável por perceber o mundo e traduzir essas informações em abstrações de alto nível compreensíveis pelo agente.

O processo de percepção pode ser classificado em dois tipos diferentes, a abordagem passiva ou ativa. A percepção passiva é realizada independente do estado interno do agente. Já na percepção ativa, o agente determina o que precisa ser sentido e então realiza uma ação adequada para perceber o que é necessário.

Em cenários do mundo real, um problema que frequentemente surge é que os robôs têm percepção parcial do ambiente. Para contornar esse tipo de problema, uma solução é perceber ativamente o que interessa ao robô, em vez de apenas perceber passivamente o que está disponível no ambiente.

Nos agentes BDI tradicionais apenas a percepção passiva é normalmente levada em consideração. Assim, este trabalho propõe um modelo de percepção ativa integrado à arquitetura de agentes BDI. Uma vez que as vantagens da percepção ativa são destacadas em cenários complexos do mundo real, o modelo proposto deve ser avaliado em cenários o mais próximo possível do real. Para isso, também está sendo proposta uma arquitetura para a programação de robôs inteligentes baseada nos conceitos cognitivos do BDI.

OBJETIVOS

Este trabalho tem como principal objetivo a concepção, desenvolvimento, e avaliação de um mecanismo de percepção ativa integrado com o ciclo de raciocínio dos agentes BDI. Além disso, outro objetivo é desenvolver uma arquitetura embarcada que permita utilizar agentes BDI para programar a inteligência de robôs, de forma a possibilitar que o modelo de percepção ativa proposto seja aplicado em robôs.

METODOLOGIA

Para alcançar os objetivos propostos a primeira tarefa consiste em expandir os conceitos e definições de agentes BDI para incluir a percepção ativa, em seguida, com base nos novos conceitos é proposta uma possível modificação do ciclo de raciocínio dos agentes BDI para incluir a percepção ativa. Em sequência, é analisada diversas opções existentes para transformar o modelo proposto em uma implementação. Então, é discutido como de fato implementar a percepção ativa para agentes BDI utilizando a linguagem de programação de agentes Jason.

Como um dos objetivos é testar a proposta de percepção ativa em ambientes reais, é desenvolvida uma arquitetura embarcada para possibilitar que agentes Jason sejam utilizados para programar robôs através do uso do ROS.

Com o intuito de verificar a eficiência da arquitetura desenvolvida e se a utilização de agentes BDI para programar a inteligência de robôs oferece alguma vantagem, são elaborados dois experimentos que consistem na utilização de veículos aéreos não tripulados programados com

Jason para realizar missões de busca e resgate.

RESULTADOS E DISCUSSÃO

Neste trabalho é demonstrado que é possível integrar agentes BDI com o ROS e utiliza-los para programar a inteligência de robôs. Com a execução de experimentos são obtidas evidências que demonstram que a utilização de agentes BDI facilita o processo de programação de comportamentos complexos. Porém, a utilização de agentes BDI para este fim trás como desvantagem um maior custo computacional, entretanto em plataformas como rapsberry pi 3 isto não é proibitivo. E também, é mostrado que é possível desenvolver um mecanismo de percepção ativa integrado com o ciclo de raciocínio dos agentes BDI. Através da realização de alguns experimentos é apresentado como a percepção ativa pode impactar na realização de missões de busca e resgate com VANTs. E ainda, é apresentado como a dinamicidade do ambiente se relacionada com a frequência de realização da percepção ativa, e como deve ser feito a parametrização da mesma.

CONSIDERAÇÕES FINAIS

Este trabalho propõem uma arquitetura embarcada composta de agentes BDI e ROS para a programação de robôs inteligentes, e um modelo de percepção ativa integrada com o ciclo de raciocínio de agentes BDI. E são apresentadas evidências que apontam as vantagens da utilização de agentes BDI para programar a parte cognitiva de robôs, e a importância da utilização de percepção ativa em alguns cenários.

Palavras-chaves: Agentes BDI. Robótica Móvel. Percepção ativa. VANT.

ABSTRACT

In multi-agent systems the main process responsible for obtaining information about the environment is perception, generally this process is performed passively regardless of the agent's internal state. However, especially when inserted in real environments, a frequent problem is that agents have partial perception of the environment, failing to perceive everything that is necessary. To circumvent this problem, a solution is to actively take actions to perceive what is of interest to the agent, instead of just passively perceiving what is available in the environment, for example, in a computer vision system, the camera can be repositioned to have a better view of an object. Thus, this work aims to develop an active perception model integrated with the reasoning cycle of BDI agents. Also, one of the goals is to test active perception in environments as close to real as possible, for this reason, an embedded architecture was developed that aims to promote the use of cognitive agents in cooperation with *The Robotic Operating System* to program the intelligence of robots. Experiments were performed using BDI agents with ROS to command unmanned aerial vehicles to analyze the benefits and impacts of using cognitive agents and active perception to program robot intelligence.

Keywords: BDI Agents. Mobile robotics. Active Perception. UAV.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AP	Active perception
APB	Active perception belief
APD	Active perception desire
API	Active perception intention
APP	Active perception plan
BDI	Belief, desire, intention
FSM	Finite state machine
MAS	Multi-Agent systems
OS	Operating System
PP	Passive perception
RAPB	Regular active perception belief
RB	Regular belief
RD	Regular desire
RI	Regular intention
RP	Regular plan
ROS	The Robot Operating System
SA	Situational awareness
SITL	Software in the loop
TAPB	Timed active perception belief
TB	Timed belief
UML	Unified modeling language
UAV	Unmanned aerial vehicles

CONTENTS

1 INTRODUCTION

When designing robots one of the difficulties is to develop autonomous software that is capable to perceive the environment, reason about what it knows, and then choose appropriate actions. To solve this challenge, multi-agents systems (MAS) techniques seem to be an advantageous approach since it offers theoretical and practical tools to develop autonomous systems (BORDINI; HÜBNER; VIEIRA, 2005; BORDINI; HUBNER; WOOLDRIDGE, 2007). Among the benefits, agents can properly balance reactivity and pro-activeness, specially those agents built on top of the BDI model (BDI stands for Belief, Desire, Intention).

In MAS one of the mechanisms that agents have to gather knowledge about the state of the environment is via a process called perception, which is responsible for sensing the world and translating those information into high-level abstractions understandable by the agent, in BDI agents this is represented by beliefs. As proposed in (SO; SONENBERG, 2009), the perception process can be classified into two different types, the bottom-up (passive) or top-down (active) approach.

The *passive perception* is described as a process that does not require the agent to deliberate about its sensing needs, it perceives the environment in the same way independently of its own internal state. On the other hand, the *active perception* process was characterized by being goal-driven, which means that the goals of the agent determine what needs to be sensed and then a proper action is taken in order to perceive what is needed.

In (BAJCSY; ALOIMONOS; TSOTSOS, 2018) active perception was thoroughly reviewed proposing the following definition: "An agent is an active perceiver if it knows why it wishes to sense, and then chooses what to perceive, and determines how, when and where to achieve that perception". This statement will serve as guideline for this work.

In real world scenarios one problem that often arises is that robots have partial perception of the environment, mostly due to computational power restrictions, sensor limitations, and objects being occluded or out of range. In order to circumvent this type of problem one solution is to actively perceive what is of interest to the robot instead of only perceiving passively what is available in the environment, e.g., in a computer vision system the camera can be repositioned in order have a better view of an object, a robot indoors may move outside to check if there is still light or not.

In the traditional BDI agents (WOOLDRIDGE, 1999) only the passive perception is usually taken into account. The regular BDI architecture assumes that the agent knowledge is updated. However, it is possible that agents have partial or outdated information about the environment, specially when they are inserted in the real world. Therefore, it would be advantageous for the agents to update or acquire knowledge about the world before deciding their actions.

Thus, this work proposes a model for active perception integrated with BDI agents architecture. Since the advantages of active perception are highlighted in complex real world scenarios, the proposed model should be evaluated in scenarios as close to real as possible. Therefore, the active perception mechanism developed is evaluated using robots as testbed. For this, it is also being proposed an architecture for programming intelligent robots based on the cognitive concepts of BDI. To verify if the developed architecture is feasible to be embedded and practically used to operate robots, a couple of experiments are performed. Also, to highlight the advantages and limitations of using BDI agents to program robots, its usage is compared to more traditional approaches. Then, new experiments including active perception are executed to evaluate its impact in the deliberation process, and in the computational resources consumption.

The reminder parts of this work are organized as follows. Chapter 2 provides the literature review for what is discussed in this work; Chapter 3 describes the model, some design aspects, and implementation details of the active perception model being proposed; Chapter 4 contains the description of the proposed architecture used to integrate BDI agents and hardware; Chapter 5 details the performed experiments; Chapter 6 outlines the conclusions and the future works.

2 LITERATURE REVIEW

This chapter addresses the literature review of the main concepts covered in this work. Including intelligent agents, BDI agents, active perception, and a brief review of the related works of agents and hardware integration.

2.1 INTELLIGENT AGENTS

As stated in (WOOLDRIDGE, 1999), the task of defining intelligent agents is not an easy one, even because there is no consensus for the concept of intelligence. Despite this, the author came up with the following definition: "An intelligent agent is one that is capable of flexible autonomous action in order to meet its design objectives", and flexible means that it posses reactivity, pro-activeness, and social ability. *Reactivity* is the ability to perceive the environment and promptly reacting according to what is perceived; *pro-activeness* is the capability of taking the initiative to perform actions in order to achieve its goals, this is called goal-driven behaviour; *social ability* is the capacity of interacting with other agents.

It is relevant to emphasize that an important characteristic of intelligent agents is the balance between reactivity and pro-activeness (WOOLDRIDGE, 1999). In the case that an agent is only reactive, it will not perform actions to achieve a goal, it will simply be reacting to the environment which probably will not lead to the accomplishment of goals. On the other hand, if an agent is purely pro-active, goal-driven, it will take actions to accomplish goals but it will never check if the conditions that led it to commit to those goals still stands, which may result in an agent trying to conclude a goal that is no longer possible to be completed.

2.2 BDI AGENTS

According to Bratman et al.(M. E. BRATMAN; ISRAEL; POLLACK, 1988) an ideal but unrealistic solution to the problem of balancing between reactivity and pro-activeness would be to compute at each instant of time which is the best possible course of action. However, it is not possible since agents have a limited amount of resources to perform computation. Therefore, the author proposed an architecture for practical reasoning based in the cognitive notions of *belief*, *desire*, and *intention* (BDI) (M. BRATMAN, 1987), which in short is the combination, in the right amount, of reactivity and pro-activeness.

Agents based on the BDI architecture are composed of three main components: beliefs, desires, and intentions. *Beliefs* are the representation of what the agent knows about the world and itself, *desires* are what the agent wants to achieve, and *intentions* are the desires that the agent decided and committed to accomplish.

As an example of how the practical reasoning happens in BDI agents, lets analyze a scenario of a search and rescue mission using unmanned aerial vehicles (UAVs). In this context, the agent is an UAV that carries a buoy. Lets suppose that this UAV receives a message informing the location of several drowning victims, immediately the agent begins to have a *desire* to deliver a buoy for all the victims. However, it is not possible to drop a buoy for all of them, thus the agent must choose one and commit to rescuing it. Based on the *beliefs* of the victims' locations, the agent commits to deliver a buoy to the nearest.

The desire that the agent has committed becomes an *intention*.

According to Wooldridge (WOOLDRIDGE, 1999) *intentions* are crucial for the practical reasoning process and they have certain properties, such as *driving means-ends reasoning*, *constraining future deliberation*, *persisting*, and *influencing beliefs upon which future practical reasoning is based*.

Driving means-end reasoning: consists that the agent really tries to achieve its *intention*. To do this, it must define how it will achieve its goals and, if it fails, he must try to choose different ways of achieving it, instead of abandoning the intention immediately. In the case of the UAV, the agent must trace a path and follow it to reach the victim, and in the case that the path is not right or blocked by trees or something, it should try to follow a different path.

Constraining future deliberation: indicates that the agent should not consider options that are not compatible with its intentions. In our example, the agent should not consider not dropping a buoy for a victim in need.

Persisting: intentions should persist while there is no reason for abandoning it. Plausible reasons for dropping an intention are that the intention was achieved, it is no longer possible to achieve its goal, or the purpose for having that intention does not exist anymore. In the context of the rescue mission, it could be because the agent successfully dropped the buoy for the victim, the UAV does not have enough battery to reach its goal, or the victim was already rescued by someone else.

Influencing beliefs upon which future practical reasoning is based: the agent should plan its future based on the assumption that it will successfully fulfill its intention, because it would be irrational to persist with an intention if the agent does not believe it will be achieved. In the case of the UAV, the agent could plan what it would do after the rescue considering that it successfully dropped the buoy for the victim.

With all this in mind, Wooldridge (WOOLDRIDGE, 1999) defined the process of practical reasoning of BDI agents as can be seen in listing 2.1, in summary it consists in the mapping of the current percepts and beliefs into actions.

First, based on the current beliefs and perceptual inputs a *belief revision function* (brf) updates the agent's beliefs. In the example of the UAV, considering that the message containing the victims' locations is a perceptual input, the brf would take this perception and all the others and update the agent's belief base to include the victims locations and any other new info acquired.

Then, an *options generating function* determines the new desires of the agent based on its current intentions and beliefs. For the UAV agent, this function would take the new beliefs about the victims' location and their current intentions and, based on this, would generate the desire to rescue all victims, assuming that the agent had no conflicting intentions.

Following, a *filter function* takes the current beliefs, desires, and intentions of the agent and determines its new set of intention, this represents the deliberation process. In the context of the rescue mission, the filter function would take the beliefs of the victims' location, the desire for rescuing all of them, the current intentions and, with this, commit to the desire of rescuing the nearest victim.

Lastly, an *action selection function (execute)* selects an action to be performed based on the agents intentions. For the UAV, this would represent taking off, flying trough a path that leads to the victim, and dropping a buoy.

Listing 2.1 – BDI reasoning cycle

```
function action(p:P):A
begin
    B = brf(B,p)
    D = options(B,I)
    I = filter(B,D,I)
    return execute(I)
end function action
```

2.3 ACTIVE PERCEPTION

In this work, active perception is reviewed in the context of intelligent agents. Firstly, the review will address works that have a more theoretical point of view, such as (WEYNS; STEEGMANS; HOLVOET, 2004; SO; SONENBERG, 2009), then it will concentrate on practical works, like (BEST; CLIFF, et al., 2018; BEST; FAIGL; FITCH, 2018; J-M et al., 2012; UNTERHOLZNER; HIMMELSBACH; WUENSCHE, 2012; RAFAELI; KAMINKA, 2017).

In (WEYNS; STEEGMANS; HOLVOET, 2004) the authors point out the lack of theories and general models for perception in MAS, despite its importance, highlighting that most MAS adopt a simplistic model for perception or *ad hoc* solutions. Therefore, they proposed a generic model for active perception in situated MAS, which is composed of three functional modules: sensing, interpreting, and filtering.

Sensing is the process of mapping the state of the environment to a representation, which depends on two factors: *foci*, and *perceptual laws*. The latter is the set of environmental constraints of the representation, e.g. something behind an obstacle can not be perceived, in the physical world the *perceptual laws* are intrinsic to the environment, but in simulated/virtual environments they must be explicitly defined. The former, *foci*, is the direction of perception, which allows the agent to choose what type of information it wishes to perceive, e.g. an agent can select to smell or see. Following, *interpreting* is the translation of a representation into a perception, which is an expression that is understandable by the machinery of the agent. Lastly, *filtering* is the process of selecting only the perceptions that match a specific criteria. The authors make the following comparison to biological systems: "Focus selection can be viewed as choosing a particular sense to observe the environment, while filter selection is comparable to the direction of attention, both driven by the current interests".

This model of active perception is interesting since it allows the agent to direct its focus to relevant aspects of the environment. However, it considers that what the agents want to perceive is directly available, which in physical systems may not be true, e.g. using a visual focus the object of interest may be out of range. One solution for this problem would be to include a mechanism in the *foci* step to ensure that the agent is able to perceive what it wishes, e.g. the visual system could be repositioned in order to be in range of the object of interest.

In (SO; SONENBERG, 2009), the authors emphasize the importance of the perception process for intelligent agents, since it is the main mechanism used by agents to gather knowledge about the environment, and most of the decisions taken by the agents are based on what it knows about the environment. Then, it is pointed out that the definition of an agent's perception (sensing) behaviour usually consists on defining the strategies for two factors: *dynamism*, frequency of sensing, and *selectivity*, choosing what to sense. The authors argue that the answer for the sensing behaviour lies in active (goal-driven) perception, and that situational awareness (SA) is an appropriate approach for solving active perception. Therefore, they proposed a SA mechanism that enables the agent to switch between goal-driven and data-driven behaviour, where the top-down goal-driven process works by projecting what is known about the environment into the near future revealing what must be sensed, in the case of BDI which beliefs must be updated. One missing point of this work is that it does not address the problem of integrating it within any agent's architecture.

In (UNTERHOLZNER; HIMMELSBACH; WUENSCHE, 2012), an active perception framework was developed in order to enable an autonomous car to redirect its sensors to focus on relevant surrounding area, in urban traffic scenarios. To accomplish this, three main criteria are taken into account, the importance of other vehicles, the available information about different vehicles, and the sensor coverage of the vehicle's relevant surrounding area. However, the proposed solution solves only the active perception problem specific to its respective use case.

Despite the existence of several works that explore the concept of active perception there are almost none that addresses the problem of integrating active perception within agents reasoning cycle in a more general way. One of the few works available is (RAFAELI; KAMINKA, 2017) where the authors proposed a solution to integrate active perception at the architecture level of a BDI agent. For this, they focused on enhancing the BDI architecture, which usually has as presupposition that the agent has all the necessary beliefs about the world, however, is not uncommon for beliefs to be unknown or outdated. Thus, the architecture was modified in order to apply active perception plans to reveal missing beliefs, unknown or outdated, that are used as preconditions for plans.

With that in mind, the authors (RAFAELI; KAMINKA, 2017) proposed four algorithms: IAP, ITAP, SAP, and DSAP. The first one, IAP, reveals all the missing beliefs of the agent, thereby, it guarantees that the optimal plan is selected. However, since it is likely to perform unnecessary active perception plans its performance is not ideal, except when executing active perception plans that have no cost. The second one, ITAP, allows the agent to choose between performing an active perception plan or a feasible plan, then if perception plans have cost it may choose, at any time, to perform a feasible plan instead of performing all the perception plans. With this, the algorithm is able to reduce the cost of the active perception, but it is not guaranteed that the optimal plan will be selected. The third one, SAP, demands that the agent commits to a plan before performing the active perception plans that reveals it, and then if it becomes feasible that the agent can choose to execute it or select another plan to reveal. Lastly, DSAP, also demands the agent to commit to a plan to be revealed and additionally allows the selection of the order of execution of the active perception plans that reveal it.

These algorithms seem to be a suitable approach to solve the problem of integrating active perception with the BDI architecture. However, it lacks information related with how the active perception mechanism was developed and properly integrated within the BDI reasoning cycle. Besides, there is no implementation available or experimental results showing its effectiveness.

2.4 AGENTS AND HARDWARE ARCHITECTURES

The use of BDI agents to control robots is already being explored (VERBEEK, 2003; MORAIS, 2015; PANTOJA et al., 2016; MENEGOL; HÜBNER; BECKER, 2018). Based on a previous work on the same research group, (MENEGOL; HÜBNER; BECKER, 2018) proposed an architecture for embedding Jason agents and effectively embedded the solution into a real unmanned aerial vehicle (UAV), proving that it is feasible to use BDI agents to command real-world robots. While these proposals keep the hardware details transparent for the agent programmer, the integration works in an *ad hoc* manner. The high-level (BDI agent) and low-level layers (robot hardware) are connected via specific protocols and ports – no standardization has been used or defined for using agents to command hardware. As a consequence, if the hardware is exchanged, part of the architecture must be reprogrammed for the new specific use case. Therefore, it is not trivial to reuse the architecture proposed by Menegol et al. for other applications. Also, it does not provide any interface for utilizing any of the robotic software already developed by the roboticist community, such as navigation, localization, and control stacks.

Wesz (2015)(WESZ, 2015) proposed JaCaROS an architecture composed of Jason, CArtAgO (RICCI et al., 2009), and ROS to integrate BDI agents with hardware. In summary, CArtAgO artifacts are used as the main abstraction for sensors and actuators, which communicate with the hardware software via ROS topics and services. The authors already provide some artifacts for handling a few existent sensors and actuators. However, for each different hardware it is necessary to implement a specific artifact using Java, where it is needed to handle how actions are converted into ROS messages, how the messages coming from ROS are converted into beliefs, and how the agent is updated in relation to the artifact. Using a different hardware requires that a significant peace of software is programmed, demanding that the programmer possess knowledge in Java and CArtAgO, resulting in a non trivial process. On the other hand, this solution supports customization quite well. One interesting point of this method is that to interact with the hardware the agent must only know about how to operate the artifact. Another advantage of this method is that with the use of ROS it is possible to leverage all the robotic software that already exists within the framework.

In order to promote the integration of hardware and BDI agents, (MORAIS, 2015) also developed a solution that combines Jason and ROS. The author modified the architecture of the Jason agents to receive perceptions and to send actions using standardized ROS topics. The agents communicate with intermediary nodes called decomposers and synthesizers, the former is responsible for translating high-level actions into commands for the hardware, and the latter receives data from the hardware and translate it to perceptions understandable by the agents. Thus, the exchange of hardware requires that new decomposers and synthesizers nodes are programmed. Both can be programmed in any language supported by ROS since they are decoupled from the Jason agents. Once again, this process is not trivial. An advantage of this approach is that the integration with the hardware is totally transparent for the agent. Also, since it uses ROS, the existing robotic stack can be utilized.

Although inspired by all these work, we focus particularly on the improvement of the architecture proposed by (MORAIS, 2015). This will be accomplished by establishing standards for using ROS alongside Jason, and by designing an intermediary node that is more generic, mitigating the need of reprogramming when the hardware is changed.

2.5 LITERATURE REVIEW CONCLUSIONS

Although there are several studies on BDI agents and active perception, there are few studies that discuss how to include active perception in BDI agents, especially integrated in their reasoning cycle. Even the few studies that address this theme do not go into much detail about how the proposed model was defined and implemented, much less how it was evaluated.

Regarding the integration of BDI agents with hardware, there are already some works that solve this problem, however, most of them work in an ad hoc manner, requiring that for each different application a considerable part of the integration architecture must be reprogrammed.

With this, this work aims to propose an active perception mechanism integrated with BDI agents reasoning cycle, detailing the necessary definitions, how the BDI's reasoning cycle can be modified to include active perception, what aspects must be taken into account to transform the model into an implementation, how the implementation can be done, and how the active perception mechanism can be evaluated in realistic scenarios. Also, another objective is to propose and evaluate a generic architecture to integrate BDI agents with hardware, which is used in the experiments performed to assess the active perception.

3 ACTIVE PERCEPTION WITHIN BDI AGENTS REASONING CYCLE

In this chapter, the active perception mechanism that is being proposed and the methodology used in its development process are detailed. In summary, the development was divided into three phases: modeling (section 3.1), design (section 3.2), and implementation (section 3.3). The first step consists in laying out some definitions and proposing a possible approach to modify the BDI's reasoning cycle to include active perception. The second phase addresses some aspects that must be taken into account when transforming the proposed model into a real implementation. The third stage details how the implementation of active perception can be done in a BDI agent programming language.

3.1 MODEL

This section describes the model of active perception that is being proposed in this work. First, some definitions used in this document are presented, then how the traditional BDI reasoning cycle can be modified to include active perception, and one possible approach for modeling the active perception mechanism.

3.1.1 Definitions

Before discussing the model of active perception that is being proposed, it is important to review some concepts of the traditional BDI reasoning cycle, and to introduce some new definitions related to active perception that are used throughout this work. The concepts covered in this section and further discussed are: *perception*, *beliefs*, *desires*, *intentions*, *plans*, and *active perception selection pressure*.

3.1.1.1 Perception

A crucial concept for understanding this work is the definition of the two different types of perception that are being explored here, passive and active perception. Figure 3.1 summarize this distinction.

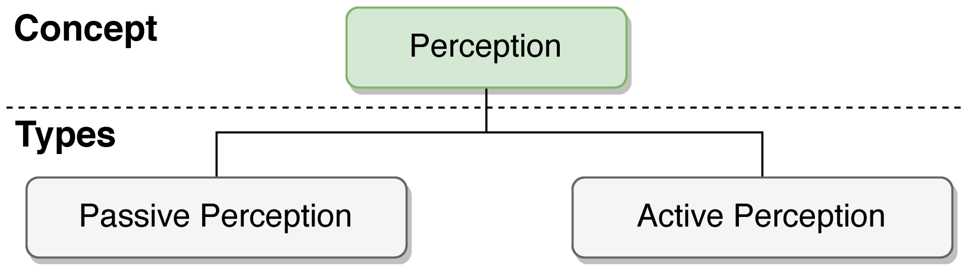


Figure 3.1 – Perception taxonomy

Perception is the process of sensing the world and translating information into high-level abstractions understandable by the agent, in BDI agents this information is represented by beliefs. The perception process can be classified into two different types, the bottom-up (passive) or top-down (active)(SO; SONENBERG, 2009).

Passive perception (PP) is the type of perception that is used in the traditional BDI reasoning cycle. At the beginning of each cycle the agent takes the perceptual inputs from its perception sources, e.g. sensors, and add it to its belief base. This process happens independently from the agent internal state (beliefs, desires, and intentions), that is why it is called a passive process.

As stated in (SO; SONENBERG, 2009), passive perception is as a process that does not require the agent to deliberate about its sensing needs, the agent perceives the environment in the same way independently of its own internal state.

Active perception (AP) is the process of actively trying to perceive the beliefs that are needed for the reasoning process. For this, the agent takes actions based on its internal state (beliefs, desires, and intentions) to put itself in a world state where it is able to passively perceive what it needs. Thereby, active perception can be summarised as the process of taking actions which leads to a desired passive perception.

In (SO; SONENBERG, 2009) this process was characterized by being goal-driven, which means that the goals of the agent determine what needs to be sensed and then action is taken in order to perceive what is needed.

3.1.1.2 Beliefs

An important definition for comprehending the active perception model proposed in this work is the distinction between different types of beliefs and their states, this differentiation can be seen in figure 3.2.

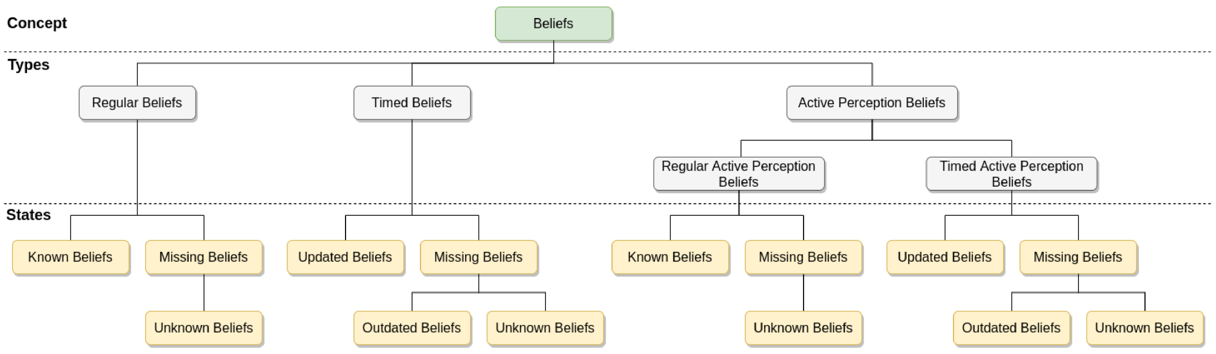


Figure 3.2 – Beliefs taxonomy

Beliefs are the representation of the information that the agent has about the world or itself. They can be acquired via perception, messages from other agents, or through the agents own reasoning process. Following, the types of beliefs are defined and the states in which each type of belief can be.

3.1.1.2.1 Types

Regular beliefs (RB) are the subset of beliefs that is present in the traditional BDI architecture. As previously discussed one of the processes for acquiring beliefs is via perception, the RB set is subjected only to the passive perception process, it does not make use of active perception.

$$RegularBeliefs \subseteq Beliefs \tag{3.1}$$

Timed beliefs (TB) are a subset of beliefs, their difference from regular beliefs is that they have a lifetime, when TB are not updated for longer than its lifetime they are considered to be outdated and, therefore, it can no longer be trusted. Regarding perception, just like RB it is only subjected to passive perception.

$$TimedBeliefs \subseteq Beliefs \tag{3.2}$$

Active perception beliefs (APB) are a subset of beliefs, they differ from regular beliefs and timed beliefs regarding perception, APB can be also subjected to active perception instead of only passive perception.

$$ActivePerceptionBeliefs \subseteq Beliefs \tag{3.3}$$

Since active perception capabilities can be added both to regular beliefs and timed beliefs, the active perception beliefs set is separated in two subsets: **regular active perception beliefs** (RAPB), and **timed active perception beliefs** (TAPB). The former has the characteristics of RB and APB, and the latter from TB and APB.

$$RegularActivePerceptionBeliefs \subseteq ActivePerceptionBeliefs \tag{3.4}$$

$$RegularBeliefs \cap ActivePerceptionBeliefs = RegularActivePerceptionBeliefs \tag{3.5}$$

$$TimedActivePerceptionBeliefs \subseteq ActivePerceptionBeliefs \tag{3.6}$$

$$TimedBeliefs \cap ActivePerceptionBeliefs = TimedActivePerceptionBeliefs \tag{3.7}$$

3.1.1.2.2 States

Known beliefs are the set of beliefs that the agent knows about, are contained in the belief base of the agent, and are lasting, that is, once they are known it only ceases to be when they are removed from the belief base. Both regular beliefs and regular active perception beliefs have this state.

$$(KnownBeliefs \cap RB) \cup (KnownBeliefs \cap RAPB) = KnownBeliefs \tag{3.8}$$

Unknown beliefs are the set of beliefs that the agent has no information about, in other words, the ones that are not in the agent's belief base. Regular beliefs, timed beliefs, regular active perception beliefs, and timed active perception beliefs have this state.

$$\begin{gathered}(UnknownBeliefs \cap RB) \cup (UnknownBeliefs \cap TB) \cup \\ (UnknownBeliefs \cap RAPB) \cup (UnknownBeliefs \cap TAPB) = UnknownBeliefs\end{gathered} \tag{3.9}$$

Updated beliefs are the set of beliefs that the agent knows about and can be considered updated. Different from known beliefs, the beliefs in this set are not lasting, it has a lifetime and once it is over they are removed from this set. Both timed beliefs and timed active perception beliefs have this state.

$$(UpdatedBeliefs \cap TB) \cup (UpdatedBeliefs \cap TAPB) = UpdatedBeliefs \tag{3.10}$$

Outdated beliefs are the set of beliefs that were previously considered updated beliefs but their state changed because they were not updated for longer than their lifetime. Both timed beliefs and timed active perception beliefs have this state.

$$(OutdatedBeliefs \cap TB) \cup (OutdatedBeliefs \cap TAPB) = OutdatedBeliefs \tag{3.11}$$

Missing beliefs are the set of beliefs that contains the unknown and outdated beliefs.

$$Outdatedbeliefs \cup UnknownBeliefs = MissingBeliefs \tag{3.12}$$

3.1.1.2.3 Beliefs states and transitions

The transitions among the states of **regular beliefs** is detailed by a finite state machine (FSM), as can be seen in figure 3.3. Unknown RB can become a known RB with the **usual processes**: passive perception process, exchanging messages with other agents, or through their own reasoning. Also, a known RB can be updated or removed via the usual processes.

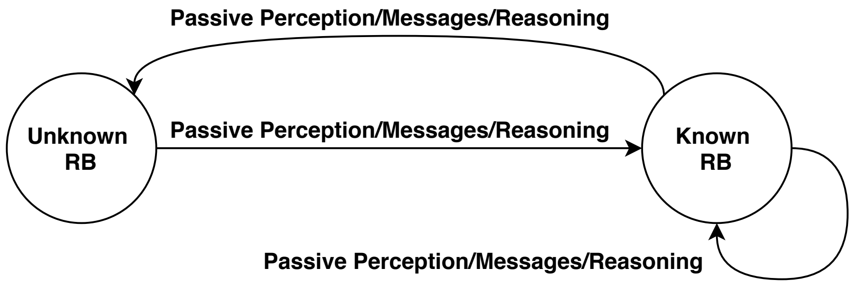

Passive Perception/Messages/Reasoning
Unknown RB
Passive Perception/Messages/Reasoning
Known RB
Passive Perception/Messages/Reasoning

Figure 3.3 – Regular beliefs FSM

The finite state machine for **timed beliefs** can be seen in figure 3.4. The main difference from RB FSM is that updated TB can become an outdated TB. Updated beliefs have a lifetime, i.e., when the time elapsed from its last updated is greater than its lifetime its state changes to outdated. Outdated beliefs can become updated beliefs via the usual processes.

Now for **regular active perception beliefs** the FSM can be seen in figure 3.5. The difference with the RB FSM is that for an unknown RAPB to become an updated RAPB and for a known RAPB to update itself, it can use active perception in addition to usual processes.

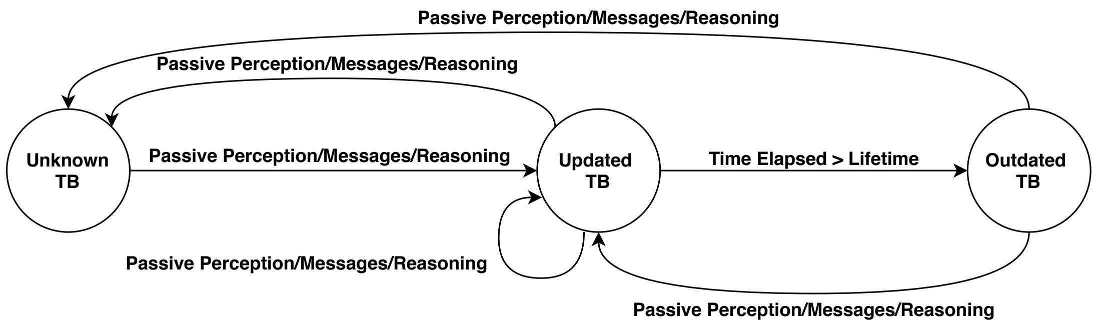

Passive Perception/Messages/Reasoning
Passive Perception/Messages/Reasoning
Unknown TB
Passive Perception/Messages/Reasoning
Updated TB
Time Elapsed > Lifetime
Outdated TB
Passive Perception/Messages/Reasoning
Passive Perception/Messages/Reasoning

Figure 3.4 – Timed beliefs FSM

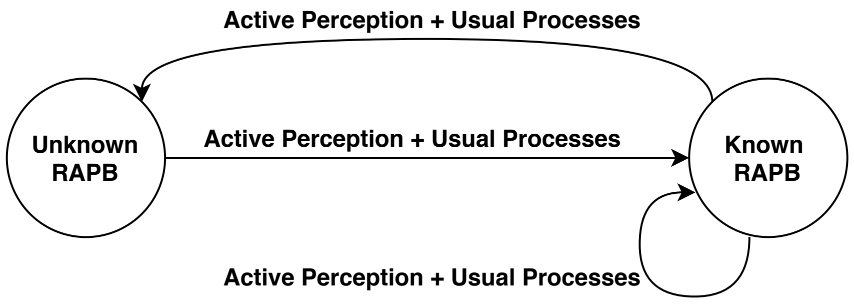

Active Perception + Usual Processes
Unknown RAPB
Active Perception + Usual Processes
Known RAPB
Active Perception + Usual Processes

Figure 3.5 – Regular active perception beliefs FSM

The **timed active perception beliefs** finite state machine is represented in figure 3.6. Its difference from the TB FSM is that outdated TAPB and unknown TAPB can become updated TAPB via an active perception process in addition with the usual processes.

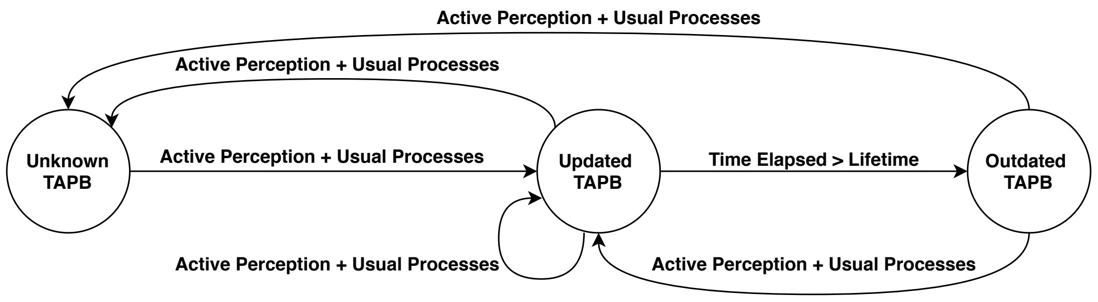

Active Perception + Usual Processes
Active Perception + Usual Processes
Unknown TAPB
Active Perception + Usual Processes
Updated TAPB
Time Elapsed > Lifetime
Outdated TAPB
Active Perception + Usual Processes
Active Perception + Usual Processes

Figure 3.6 – Timed active perception beliefs FSM

When active perception beliefs are needed for the reasoning cycle and they are in the missing state, the agent actively tries to perform active perception to change its state. However, active perception is not restricted to this situation, the agent can choose to apply AP whenever it desires, even when the belief is already updated.

3.1.1.3 Desires

The concept of desires is also expanded in this work to encompass active perception, as can be seen in figure 3.7.

Desires are what the agent wants to achieve.

Regular desires (RD) are the subset of desires that are used in the traditional BDI reasoning cycle. RD aim to change the state of the *world*. An example would be opening a door or moving from one location to another.

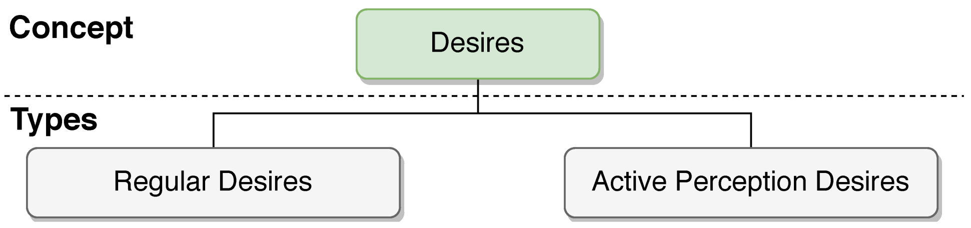


Figure 3.7 – Desires taxonomy

$$RegularDesires \subseteq Desires \tag{3.13}$$

Active perception desires (APD) are the subset of desires that aims to use an active perception process, e.g. an agent wants to apply AP in order to change the state of a missing belief to updated belief. This type of desires tries to change the agent *beliefs*.

$$ActivePerceptionDesires \subseteq Desires \tag{3.14}$$

$$RegularDesires \cup ActivePerceptionDesires = Desires \tag{3.15}$$

3.1.1.4 Intentions

Similarly to changes made in desires, the definition of intentions is also expanded, as can be seen in the figure 3.8.

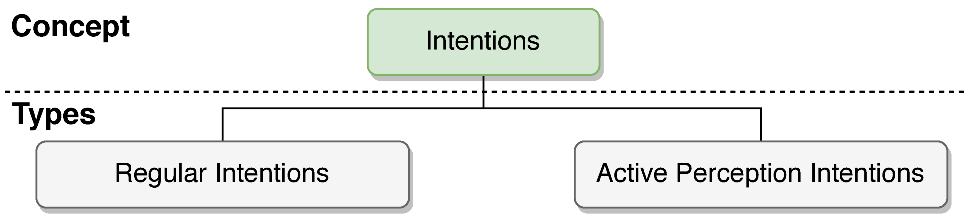


Figure 3.8 – Intentions taxonomy

Intentions are the desires that the agent decided and committed to accomplish.

Regular intentions (RI) are the subset of intentions that are used in the traditional BDI reasoning cycle. This set is composed of the regular desires that the agent committed to achieve.

$$RegularIntentions \subseteq Intentions \tag{3.16}$$

Active perception intentions (API) are the subset of intentions that contains the active perception desires that the agent committed to accomplish.

$$ActivePerceptionIntentions \subseteq Intentions \tag{3.17}$$

$$RegularIntentions \cup ActivePerceptionIntentions = Intentions \tag{3.18}$$

3.1.1.5 Plans

Another important concept for understanding the active perception mechanism is the clear definition of the types of plans and their roles, which can be summarized with figure 3.9.

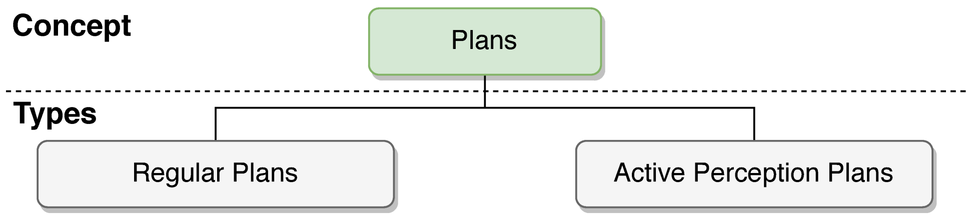


Figure 3.9 – Plans taxonomy

Plans are a set of instructions that when successfully executed lead to the fulfillment of an intention.

Regular plans (RP) are the subset of plans that are used in the traditional BDI architecture to map the agents desires into intentions. RP may have preconditions that must be fulfilled in order to be performed, e.g. a UAV must have enough battery to fly to its goal before starting the mission;

$$RegularPlans \subseteq Plans \tag{3.19}$$

Active perception plans (APP) are a subset of plans, these plans consist of a set of actions that are performed to try to change the state of a specific missing belief to updated/known belief, or to refresh an updated/known belief.

$$ActivePerceptionPlans \subseteq Plans \tag{3.20}$$

The plans set can be formalized with the equation:

$$RegularPlans \cup ActivePerceptionPlans = Plans \tag{3.21}$$

Revealing is the process of applying an active perception plan to change the state of missing beliefs to updated beliefs (RAFAELI; KAMINKA, 2017).

3.1.1.6 Active perception selection pressure

Two important aspects to be taken into account when designing the active perception mechanism are the resulting frequency at which active perception plans are performed and the credibility of each active perception belief. The active perception plan frequency is important because applying AP plans constantly results in more computational resources being used, less time being spent in performing regular plans, and more energy being consumed. In real world scenarios this is aggravated, since performing AP plans consists in executing actions, which may take a lot longer than in computational environments, and usually energy is not an abundant resource.

The credibility of active perception beliefs is the probability of an AP belief being correct, and it is directly related to the frequency at which the AP plans are performed and the dynamism of the environment. Suppose that an agent has the AP belief that it is daytime and it has no information about its location and current season, if it applies an active perception plan to verify if it is daytime at 1:00 pm and then only perform an AP plan at 7:00 pm, it could not be sure if it was still daytime at 6:30 pm, since the sunset depends on the region of the world and the current season, therefore, in this situation the credibility of this active perception belief would be low. However, if the active perception plan was performed within an interval of 5 minutes or if the agent knew in which region of the world it is located and what is the current season, it would be more sure if it is daytime at 6:30 pm, thus, in this new scenario the credibility of this active perception belief would be high.

Thus, it is important to find the right balance between the active perception plans frequency and active perception beliefs credibility. However, there is no right answer for this trade off, since it is directly tied to the dynamism of the environment, and each specific use case. For future reference this trade off will be called active perception selection pressure, when the AP frequency and the AP beliefs credibility is high the active perception selection pressure is also high, and when they are both low the active perception selection pressure is also low. This problem is in certain way similar to the balance between reactivity and pro-activeness (WOOLDRIDGE, 1999).

3.1.2 Modified reasoning cycle

The traditional BDI architecture proposed by Wooldridge (WOOLDRIDGE, 1999), discussed in section 2.2, consider that the agent has all the necessary beliefs required to deliberate which intentions it should commit to achieve. However, especially in real world scenarios, this assumption is not reasonable since beliefs may be unknown or outdated, and if this is not taken into account, the agent deliberation process may lead to the selection of suboptimal actions. One approach for solving this problem is to drop the assumption that all beliefs are known and updated, and based on the agent internal state (beliefs, desires and intentions) try to actively perceive what is required for the deliberation process.

To demonstrate the process of active perception, the example used in section 2.2 can be modified. Suppose that the UAV agent now carries a buoy and a camera facing down. The mission to be performed by the agent is still the same, when the UAV receives information about drowning victims it must rescue the nearest one. The difference now is that the rescue plan will be split into 3 parts, taking off, flying to the victim's location, and dropping the buoy. The first two parts remain the same, however, the latter, dropping the buoy, is modified to not assume that the belief of the victim's location is always known and updated, in other words, consider that the victim's location is an timed active perception belief. This is because the victim may have moved or has already been rescued and, in both cases, it would not be wise to drop the buoy for a victim that is not there, if this happens the agent should fly to the next nearest victim and try to rescue it instead. Thus, before dropping the buoy, the agent actively tries to perceive if the victim is in the informed position, or even in a small area around it.

Taking the process of practical reasoning proposed by Wooldridge (WOOLDRIDGE, 1999) as starting point, the active perception can be included as an intermediary process between the *options generating function* and *filter function*, this can be seen in listing 3.1.

The idea is that, based on the new desires of the agent and its active perception

Listing 3.1 – BDI reasoning cycle with Active Perception

```
function action(p:P):A
begin
    B = brf(B,p)
    D = options(B,I)

    APD = optionsAP(D, APB)
    if(has APD)
        API = filterAP(APB, APD, I)
        APB = execute(API)

    I = filter(B,D,I)
    return execute(I)
end function action
```

beliefs, an *active perception options generating function* (optionsAP) is applied to verify if any of the active perception beliefs (APB) that are related to the agents desires are missing beliefs, if there are any missing beliefs the agent acquire an active perception desire (APD) to reveal them. In the context of the UAV example, lets assume that the belief of the victim's location has a lifetime of 1 minute, in the case that the agent reach the victim's position after 1 minute or more of receiving the information, that belief can be considered a missing belief, thus it would generate an APD to verify if the victim is in the indicated position or in a small area around it. The optionsAP is focused on generating APD for missing beliefs, but if the agent desires it can generate an APD for any of the APB independently of its state during its regular reasoning cycle.

If there are any APD, based on the APB, APD, and current intentions an *active perception filter function* (filterAP) generates an active perception intention (API) to reveal the missing beliefs. In the example of the UAV, since there is only one APD, the filterAP function would commit to the desire of verifying if the victim is in the indicated location or in a small area around it.

Lastly, based only on the API the *execute function* select the actions that tries to reveal the missing beliefs. For the UAV, this would represent turning on the camera, running recognition algorithms, tracing a flight path, and actually flying.

As pointed out, our proposal maintains all the steps of the traditional BDI reasoning cycle, resulting on a model that contains a mix of passive and active perception. This is advantageous since the advantages of both approaches are leveraged, and the disadvantages of each method are mitigated by the other.

3.2 DESIGN

The process of transforming the proposed model with a high level of abstraction to an implementation with a lower level of abstraction can be performed in several different ways, and it requires that several details are taken into account. Therefore, to facilitate this process, in this section, these aspects are described and analyzed with an intermediate level of abstraction before committing to the implementation phase.

Following, it is described how plans are represented and selected and how the active perception mechanism can be included in this selection (section 3.2.1). Then, some designing choices are discussed, such as how to differentiate a regular belief from an active perception belief (section 3.2.2), how to reveal missing beliefs (section 3.2.3), how to test the revealed missing beliefs (section 3.2.4), for how long a belief revealed should be considered updated (section 3.2.5), and the order of execution of the active perception plans (section 3.2.6).

3.2.1 Plan representation and selection

Plans play a major role in the deliberation process since it maps desires into intentions and contain the set of instructions needed to fulfill the intention. In this work, plans are composed of a name, an associated desire, a set of instructions (plan body), and a set of preconditions (context) that must be fulfilled in order to perform the plan body. They are represent as in listing 3.2. Where *plan1* is the name of the plan, *desire1* is the desire that is being mapped into an intention, its preconditions is the logical formula *(belief1 and belief2)*, and the plan body is the set of instructions *action1 and action2*. A single desire may have multiple different plans that maps it to an intention, all these associated plans are called relevant plans.

Listing 3.2 – Plans

```
plan1 for desire1:
    preconditions = belief1 and belief2
    plan body = action1 and action2 ...
```

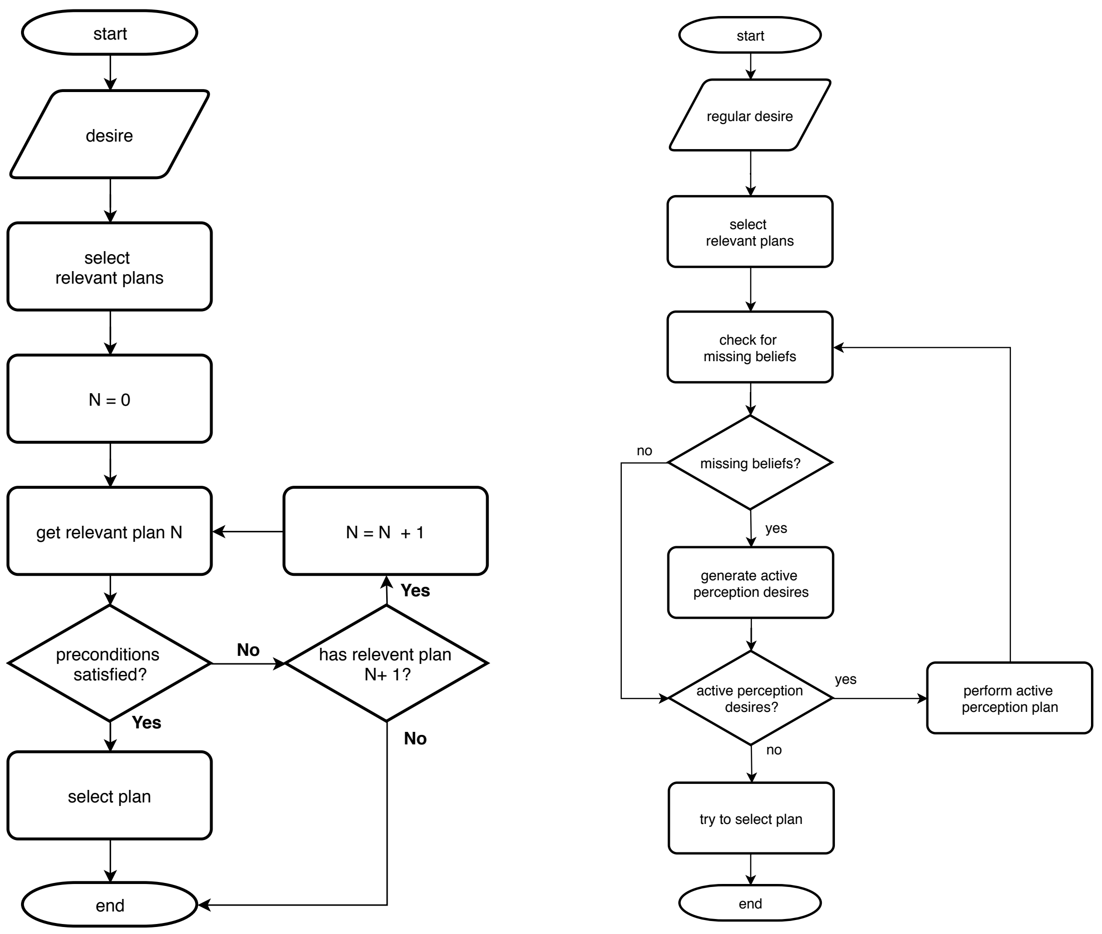
The process of selecting a plan and committing to an intention given a desire is represented in figure 3.10a. When an agent has a desire it looks for all relevant plans, then proceeds to verify one by one if its preconditions are satisfied, when a relevant plan that has all its preconditions fulfilled is found it is chosen to be performed, in case none is found the desire does not become an intention.

One approach for including the proposed active perception mechanism into the deliberation process is represented in figure 3.10b. Before checking if any of the relevant plans can be performed, it is verified whether they contain any missing beliefs in their preconditions and if there are any missing beliefs the agent generates active perception desires, then if there are any APD the agent performs active perception plans. The process of committing and selecting plans to an active perception desire is the same as the one previously described (figure 3.10a). After the active perception plans are performed the agent resumes the deliberation process and tries to select a relevant plan to commit with.

The inclusion of the AP mechanism in the plan selection flow can be done in several different ways. Thus, the rest of this section details the possibilities, advantages and disadvantages of each approach.

3.2.2 Distinction of beliefs

The first aspect that is taken into account is how to differentiate regular beliefs from active perception beliefs. For this, two approaches are considered, annotated plans and annotated beliefs. The annotated plans approach **(1)** consists to mark plans as requiring active perception and consider all beliefs in its preconditions as active perception beliefs. Therefore, before executing the plan, it must be checked whether the precondition beliefs are missing beliefs, and for those which are considered as missing an active perception plan must be carried out if one is available, and if there is none it is treated



(a) Without active perception (b) With active perception

Figure 3.10 – Plan selection flowchart

as a regular belief. This is represented in listing 3.3, where the plan *plan1* is marked with *[ap]* implying that *belief1* and *belief2* are active perception beliefs, so they may require to perform an active perception plan.

The annotated beliefs approach **(2)** consists in marking, individually, the beliefs in the preconditions of plans as being active perception beliefs, thus for the ones marked an active perception plan is performed if they can be considered missing. This alternative can be seen in listing 3.4 where only *belief1* is marked, meaning that just this belief is subjected to active perception.

Among the alternatives presented, the marking of beliefs seems to be a more natural and practical approach since the necessity of active perception is based on the characteristics of beliefs, and not plans. Besides, when marking the whole plan the ability to distinguish regular beliefs and active perception beliefs is lost.

Listing 3.3 – Annotated plans

```
plan1[ap] for desire1:
    preconditions = belief1 and belief2
    plan body = action1 and action2 ...
```

Listing 3.4 – Annotated beliefs

```
plan1 for desire1:
    preconditions = belief1[ap] and belief2
    plan body = action1 and action2 ...
```

3.2.3 Analysis of relevant plans

Another designing choice that has to be made is **(1)** if all the active perception plans referring to all the available relevant plans should be performed beforehand, and only then check if the preconditions are satisfied (grouped mode), or **(2)** to start by checking the context of the relevant plans one by one and only perform active perception when it is needed (individual mode).

Taking listing 3.5 as an example, suppose the agent has the *desire1* which has two relevant plans: *plan1* and *plan2*. The grouped mode approach consists to perform the active perception plans for *belief1*, *belief3*, and *belief4* and only then verify if the contexts *(belief1[ap] and belief2)* and *(belief3[ap] and belief4[ap])* are satisfied. The individual mode consists to first execute the active perception plan just for *belief1* and check if the context *(belief1[ap] and belief2)* is fulfilled and only if it fails proceed to perform the active perceptions plans for *belief3* and *belief4* and after that analyze if the context *(belief3[ap] and belief4[ap])* is satisfied.

In this regard, applying active perception for all the relevant plans grouped is similar to the algorithm IAP proposed in (RAFAELI; KAMINKA, 2017), where the authors concluded that IAP guarantees that the best plan is selected, but it is only optimal when all the missing beliefs must be revealed in order to choose the most advantageous plan, or when the active perception plans have no cost. Nonetheless, this conclusion has the assumption that the BDI reasoning cycle algorithm for selecting a relevant plan to commit (*filter selection function* from listing 3.1) guarantees that the best plan is selected, however, it is not uncommon for BDI architectures to have strategies that can not assure that the best plan is selected. Thereby, the grouped mode can only be considered the best choice when all this conditions are met, otherwise it may perform unnecessary active perception plans.

Listing 3.5 – Plans example

```
plan1 for desire1:
    preconditions = belief1[ap] and belief2
    plan body = action1 and action2 ...

plan2 for desire1:
    preconditions = belief3[ap] and belief4[ap]
    plan body = action3 and action4 ...
```

3.2.4 Context verification

Another important aspect to consider is the context verification, this can be done with two distinct approachs. The first method **(1)** is to reveal all the missing beliefs and only then check whether the context is true or false. Using listing 3.5 as an example, in *plan2* the beliefs *belief3* and *belief4* would be revealed and only then the whole context would be checked, in the case that *belief3* is false it is unnecessary to perform an active perception plan for *belief4*.

This problem is mitigated with the second method **(2)**, in the case the relevant plans are being analyzed in the individual mode, which consists in verifying one-by-one each missing belief after they are revealed whether the context can still be true. Using listing 3.5 as an example, when *belief3* is revealed it is verified if it is true or false and if it is false no active perception plans would be performed for *belief4*.

Therefore, using the second method is advantageous because it might avoid performing unnecessary active perception plans, however, the process of verifying the context more frequently could cause an increase in computational complexity or in the required processing power.

3.2.5 Timed active perception belief lifetime

One important characteristic of the model being proposed is that it considers the possibility of a belief being outdated, opposed to the traditional BDI architectures. Therefore, it is essential to consider how a belief would change its state of being known and updated to known and outdated.

To provide this notion, a method is to add a mark into the plan or belief to indicate for how long the belief should be considered updated once revealed **(1)**. In the case of annotated plans, such as in listing 3.6, the whole context should be revealed and verified within the time limit, in this case 1000 milliseconds, otherwise it is considered false. For annotated beliefs, like in listing 3.7, it indicates the amount of time (lifetime) that beliefs should be considered updated after being revealed, in this case *belief1* is considered outdated after 1000 milliseconds.

However, some variations can be applied to the notion of being outdated. Another method **(2)** is to consider after revealing a missing belief that it is updated until all the relevant plans related to the associated desire are checked, or one is selected. In the case that the relevant plans are being checked in the individual mode (section 3.2.3), an alternative **(3)** is to consider the revealed belief as updated until the end of the context verification. All three approaches seem to be valid, but the stricter the time constraint, the greater active perception selection pressure and consequently the AP belief credibility. However, smaller time constraints may result in more active perception plans being performed which can increase the computational cost. Thus, choosing the best alternative is tied to each use case.

Listing 3.6 – Annotated plans with lifetime

```
plan1[ap, 1000] for desire1:
    preconditions = belief1 and belief2
    plan body = action1 and action2 ...
```

Listing 3.7 – Annotated beliefs with lifetime

```
plan1 for desire1:
    preconditions = belief1[ap, 1000] and belief2
    plan body = action1 and action2 ...
```

3.2.6 Revealing order

Another aspect to take into account is the execution order of the active perception plans. A possibility **(1)** is to follow the order in which the active perception beliefs appear, another approach **(2)** is to perform the active perception plans for the AP beliefs with greater lifetime first.

Using listing 3.8 as example, for the first approach the active perception plan for *belief1* would be performed before the AP plan for *belief2*. For the second method, since *belief2* has a greater lifetime it would be revealed before *belief1*.

Listing 3.8 – Revealing order plan example

```
plan1 for desire1:
    preconditions = belief1[ap, 1000] and belief2[ap, 3000]
    plan body = action1 and action2 ...
```

The second method, revealing taking the belief lifetime into consideration, may increase the chances of the context being valid until all beliefs are revealed. Other than this, there is no clear advantages/disadvantages between both approaches.

3.2.7 Options analyses

Considering that there are three different approaches for defining the active perception beliefs lifetime and two distinct ways for the distinction of beliefs, analysis of relevant plans, context verification, and revealing order. If all the options were combined it would result in 48 possibilities. However, there are two combination of options that can not be together: revealing the relevant plans in grouped mode with beliefs lifetime being valid until the end of context, and relevant plans in grouped mode with context verification being done one by one. Hence, there are 28 valid combinations of design options, therefore, it would be exhaustive to analyze all possible combinations. Thus, considering the advantages and disadvantages of each design option presented in the previous sections, Table 3.1 was created with the combinations that look most promising.

Table 3.1 – Active perception mechanism options

Approach	**Annotation**	**Mode**	**Context Verification**	**Belief Expiration**	**Revealing Order**	**AP Selection Pressure**
1	Beliefs (2)	Grouped (1)	All (1)	Relevant Plans (2)	Natural (1)	+
2	Beliefs (2)	Grouped (1)	All (1)	Time (1)	Natural (1)	++
3	Beliefs (2)	Individual (2)	All (1)	Time (1)	Natural (1)	++
4	Beliefs (2)	Individual (2)	One by One (2)	End of Context (3)	Greater lifetime (2)	++
5	Beliefs (2)	Individual (2)	One by One (2)	Time (1)	Greater lifetime (2)	+++

Approach number 1 uses annotated beliefs, it reveals all the relevant plans beforehand, it only checks the context after all the revealing is completed, the beliefs do not expire until all the relevant plans are analyzed, and the active perception plans are applied in the order that they appear. This makes this approach the one with less active perception selection pressure out of the five, since all AP beliefs are revealed beforehand and they last until all relevant plans context are analyzed or one is selected. As a result, active perception plans are performed less frequently, all of them are executed exactly once at the beginning of the process. This method is advantageous when all active perception beliefs would need to be revealed anyway, otherwise, it would execute unnecessary

AP plans, which can be really costly in real world scenarios, and when the environment is not very dynamic, since its AP beliefs credibility is low.

Approach number 5 uses annotated beliefs, it reveals the relevant plans individually, the context is verified after revealing each missing belief, the beliefs expire with time, and the active perception plans are applied for the AP beliefs with greater lifetime first. This results in this approach being the one with more active perception selection pressure out of the five. With this, active perception plans are performed more frequently, since active perception beliefs can expire between checking the relevant plans. This method is advantageous when it is not required that all AP plans are executed. And it proves to be advantageous in more dynamic environments, since its AP beliefs credibility is high.

Approaches 2, 3, and 4 are in between 1 and 5 in relation to the active perception selection pressure. In any case, it is difficult to assess objectively which method is better, since each approach can prove to be more efficient in different situations. It is possible to notice that designing an active perception mechanism does not have an obvious answer and it is not a trivial task.

3.3 IMPLEMENTATION

This section describes an alternative for implementing the proposed model of active perception. It is divided into two parts, one describing the object model for active perception beliefs, and the other the algorithms used do implement the active perception model for the Jason language.

3.3.1 Object model for active perception beliefs

The unified modeling language (UML) class diagram for the proposed implementation can be seen in figure 3.11. Regular beliefs have exactly one predicate that in its turn can have from zero to many terms, e.g. a RB could have a predicate UAV with a couple of terms indicating the battery level, height, current position etc.

Timed beliefs inherit everything from RB, additionally they have a member called *lastUpdated* which indicates the time that the belief was last updated, and a member called *lifetime* that represents the lifetime of that belief, the default value for *lifetime* is zero which means that the belief lifetime is infinite. When TB are not updated for longer than its lifetime they are considered to be outdated. The method *isUpdated* returns *True* when the belief is updated and *False* when it is outdated.

It was opted to represent both regular active perception beliefs and timed active perception beliefs with a single class called active perception belief, inherited from TB. When *lifetime* is set to zero it is a RAPB and when it has another value it is a TAPB. Active perception beliefs may have an associated desire, intention, and plans. An active perception desire is associated when the agent has the desire to apply active perception plan for that belief. An active perception intention is associated when the agent commits to an active perception plan for the APD. An APB can have several active perception plans mapping its APD into API in different ways.

3.3.2 Jason implementation

To better evaluate the advantages and disadvantages of the several different approaches described in the last chapter, a few are implemented and integrated with Jason

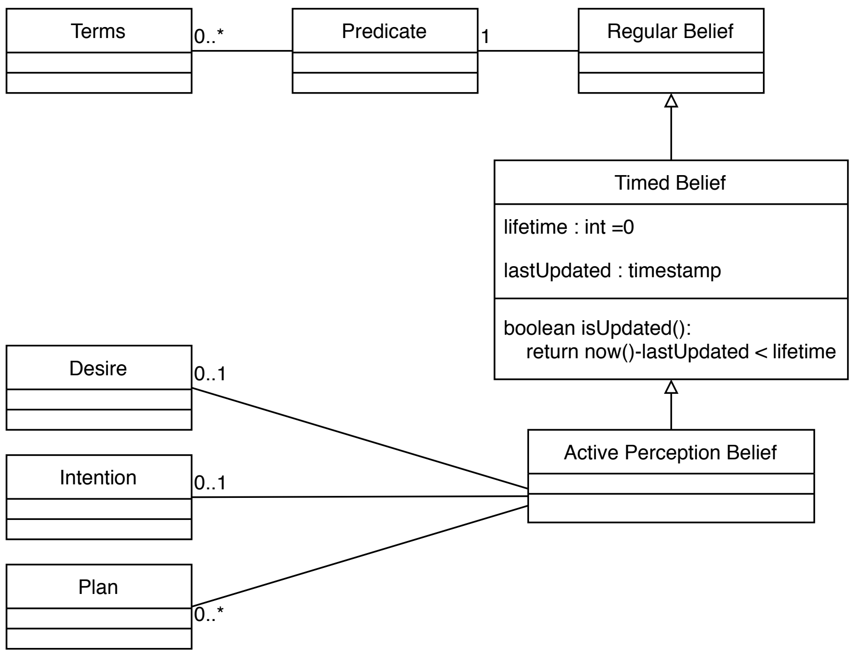


Figure 3.11 – Active perception class diagram

programming language (BORDINI; HÜBNER; VIEIRA, 2005), a BDI agents programming language.

Jason reasoning cycle can be seen in figure 3.12. If the model presented in section 3.1 were to be replicated for Jason it would required that step *Check Context* (7) were modified to include the active perception mechanism. For this, before checking the context of the relevant plans, the agent must perform the active perception plans. However, to do this, the agent generates AP intentions and, in Jason, intentions are only dealt with after step 8, so it would be necessary for the reasoning cycle to be modified to allow the agent to generate AP intentions before step 7. Thus, it is possible to notice that it involves a lot of steps and it would not be a trivial to task to modify it to include active perception.

Since it is not yet clear that the proposed model is satisfactory and which of the approaches presented in section 3.2 is the best one, in a first moment, it was opted to adopt a simpler solution that enables the implementations to be done faster. This enables that different approaches are tested with less effort. And once different approaches are tested and evaluated, it would be interesting to use the Jason architecture modification as a definitive solution.

For this, it was decided to automatically modify the agent's program to consider AP instead of altering Jason architecture. The proposed algorithm can be seen in listing 3.9 and its flowchart in figure 3.13. In summary, before executing the agent logic, the syntactic substitution algorithm checks which plans have active perception beliefs as precondition and with that information as input the change takes place. In Jason this can be done by using directives, according to (BORDINI; HUBNER; WOOLDRIDGE, 2007) "directives are used to pass some instructions to the interpreter that are not related to the language semantics, but are merely syntactical".

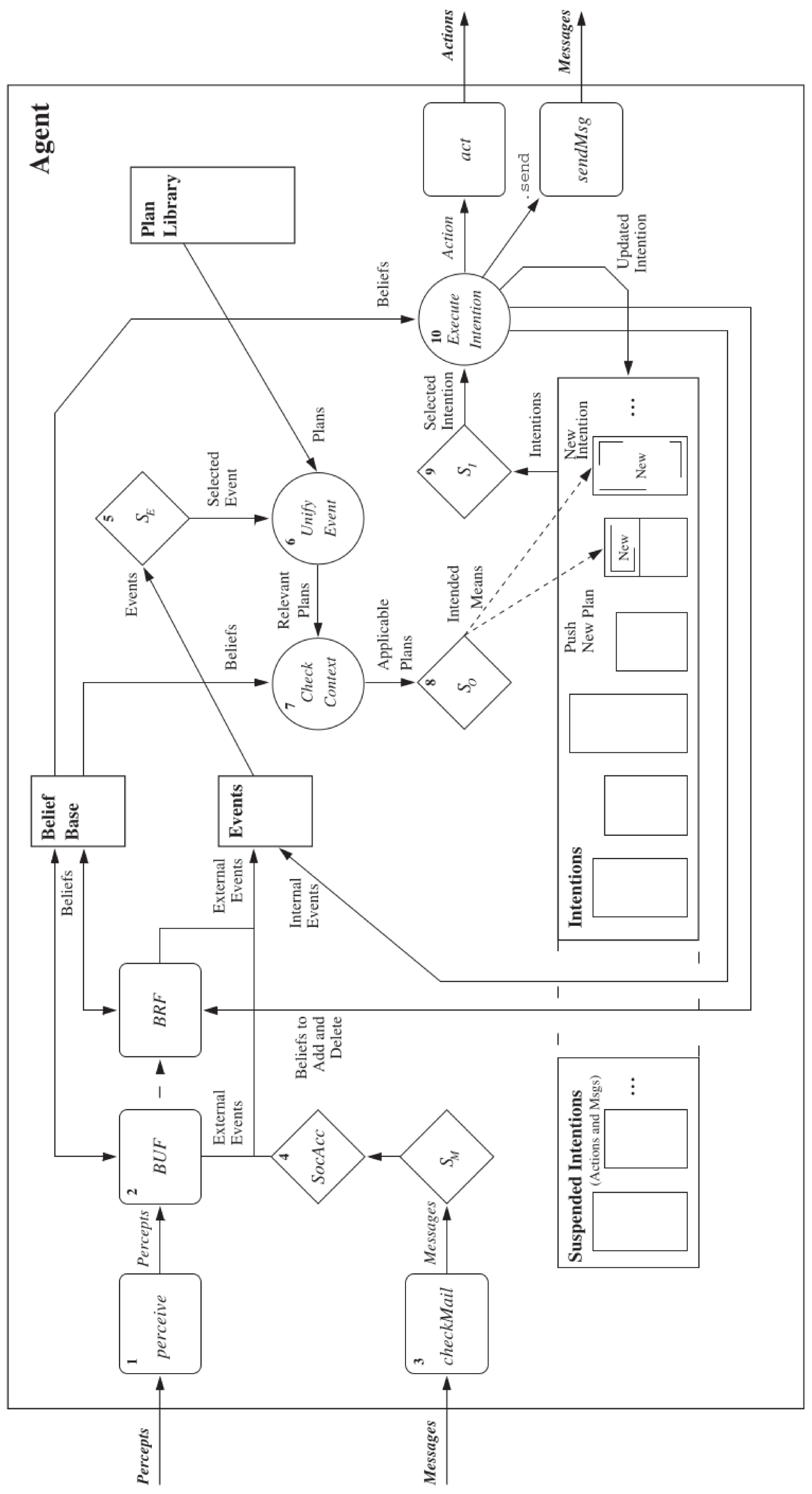


Figure 3.12 – Jason Reasoning Cycle (source (BORDINI; HUBNER; WOOLDRIDGE, 2007))

Listing 3.9 – Syntactic substitution algorithm

```
procedure SyntaticSubstitution(PlanLibrary):

    let PlansWithAp be a list

    for all Plans in PlanLibrary:
        if Plan has ap belief in context:
            PlansWithAp.add(Plan)

    PlanLibrary = Directive(PlanLibrary, PlansWithAp)
    return PlanLibrary
```

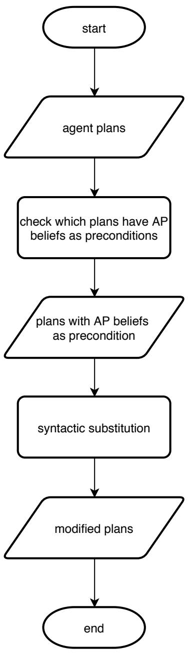

start
agent plans
check which plans have AP beliefs as preconditions
plans with AP beliefs as precondition
syntactic substitution
modified plans
end

Figure 3.13 – Syntactic substitution flowchart

The application of the syntactic substitution algorithm in the Jason program represented in listing 3.10 would identify that one of the relevant plans for the desire *!g* contains active perception beliefs as precondition, then it would perform the substitution for all the plans that map *!g* into an intention, which in this case are the plans with the trigger *+!g*. For each approach discussed in the last section the substitution is different. The active perception plans for the AP beliefs must be explicitly described as an achieve goal *+?* with *ap* as annotation.

For approach 2 from Table 3.1 the algorithm of the directive used is represented in listing 3.11. By applying the syntactical substitution algorithm with Directive2 in the Jason program shown in listing 3.10 it would result in the one presented in listing 3.12. The triggers *+!g* are replaced with *+!g[rp]*. A new plan with trigger *+!g[ap]* is added, it calls the plan *update* for each belief marked as requiring active perception, and then sets up the intention *!g*. The *update* plan is added and it is responsible for initiating the active perception plan, this takes place by triggering an event like *+?b[ap]* for the belief passed as parameter, however, this only occurs when the belief is unknown or outdated, in this case outdated means that the last update occurred before the time limit. To check the

Listing 3.10 – Jason program before syntactical changes

```
!g.

+!g: b[ap(1000)] & d[ap(3000)]
    <-  .print("GOAL G1").

+!g
    <-  .print("GOAL G2!").

+?b[ap]
    <-  .time(HH,MM,SS);
        +b[ap(_),lu(HH,MM,SS)];
        .print("Active perception plan for b").

+?d[ap]
    <-  .print("Active perception plan for d").
```

state of the belief the internal action *active_perception.isUpdated* is used, it returns *true* when the AP belief is updated and *false* when it is not.

For approach 3 from Table 3.1 the directive algorithm is represented in 3.13. When applying the syntactic algorithm with the Directive3 in the program illustrated in listing 3.10 it results in the one in listing 3.14. The event *!g* is replaced with *!g[ap, l_ 1]*, the plans *+!g[ap, l_x]* are added in order to call the *update* plan for the active perception beliefs used as precondition of the respective *+!g*, the plan's triggers *+!g* are changed to *+!g[rp, l_x]*, with x being the number of the plan, and an additional *+!g[rp, l_x]* is added for each existing *+!g[rp, l_x]* in order to call *+!g[rp, l_x+1]*. Also, the *update* plan is added just like the Directive2.

It is possible to conclude that by applying syntactic substitution in Jason it is possible to imitate the behavior of the proposed reasoning cycle to include active perception. This approach is advantageous because it does not require that all the details of Jason's reasoning cycle are known to implement active perception, which facilitates the process. However, since syntactic substitution is not conceptually equivalent to modifying the reasoning cycle, the active perception mechanism is not guaranteed to work correctly for all situations, such as when plans are added or changed dynamically while the agent is running.

All the implementations related to the active perception model discussed in this chapter can be found at https://github.com/Rezenders/jason-active-perception.

Listing 3.11 – Directive2 algorithm

```
procedure Directive2(PlanLibrary, PlansWithAp):

    # e.g: !g becomes !g[ap]
    for all Goals in Initial Goals:
        if Goal has same literal as one of PlansWithAP:
            annotate Goal with ap

    # e.g: +!z <- !g. becomes +!z <- !g[ap].
    for all Plans in PlanLibrary:
        if Plan has any Goal with the same literal as one
           of PlansWithAP:
            annotate Desire with ap

    # e.g: +!g <- ... becomes +!g[rp] <- ...
    for all Plans in PlanLibrary:
        if Plan trigger is equal to one of PlansWithAp:
            annotate Plan trigger with rp

    # e.g: +!g[ap] <- !update(b[ap(1000)]); !g[rp].
    for all distinct PlanTriggers in PlansWithAp:
        NewPlan = { trigger = PlanTrigger[ap]
                    preconditions = None
                    plan body = goal to update all the AP
                       bels and goal to achieve Plan[rp] }

        add NewPlan to Agent PlanLibrary

    # +!update(X[ap(T)])[ap]: not
       active_perception.isUpdated(X[ap(T)]) <- ?X[ap].
    UpdatePlan = {  trigger = UpdateTrigger(X)[ap]
                    preconditions = X is not updated
                    plan body = calls AP plan for X }

    add UpdatePlan to Agent PlanLibrary
    return PlanLibrary
```

Listing 3.12 – Jason program after syntactical changes (Approach 2)

```
!g[ap].

+!g[ap]
    <-  !update(b[ap(1000)])[ap];
        !update(d[ap(3000)])[ap];
        !g[rp].

+!g[rp]: b[ap(1000)] & d[ap(3000)]
    <-  .print("GOAL G1").

+!g[rp]
    <-  .print("GOAL G2!").

+?b[ap]
    <-  .time(HH,MM,SS,MS);
        +b[ap(_),lu(HH,MM,SS,MS)];
        .print("Active perception plan for b").

+?d[ap]
    <-  .print("Active perception plan for d").

+!update(X[ap(T)])[ap]: not
   active_perception.isUpdated(X[ap(T)])
    <- ?X[ap].

+!update(X[ap(T)])[ap].
```

Listing 3.13 – Directive3 algorithm

```
procedure Directive3(PlanLibrary, PlansWithAp):

    # e.g: !g becomes !g[ap,l_first]
    for all Goals in Initial Goals:
        if Goal has same literal as one of PlansWithAP:
            annotate Goal with ap and first plan label

    # e.g: +!z <- !g. becomes +!z <- !g[ap, l_first].
    for all Plans in PlanLibrary:
         if Plan has any Goal with the same literal as one
            of PlansWithAP:
            annotate Goal with ap and first plan label

    # e.g: +!g <- ... becomes +!g[rp,l_x] <- ...
    for all add Plans in PlanLibrary:
        if Plan trigger is equal to one of PlansWithAp:
            annotate Plan trigger with rp and respective
               plan label

    # e.g: -!g <- ... becomes -!g[rp,l_last] <- ...
    for all del Plans in PlanLibrary:
        if Plan trigger is equal to one of PlansWithAp:
            annotate Plan trigger with rp and last plan
               label

    #e.g:+!g[ap,l_x] <- !update(b[ap(1000)]); !g[rp,l_x].
    for all Plans in PlansWithAp:
        NewPlan = { trigger = PlanTrigger[ap,l_x]
                    preconditions = None
                    plan body = goal to update the AP bels
                       in PlanContext and goal to achieve
                       Plan[rp,l_x] }

        add NewPlan to Agent PlanLibrary

    #e.g: +!g[rp,l_x] <- !g[ap,l_x+1].
    for all Plans in PlansWithAp:
        if PlanLibrary has Plan[ap,l_x+1]:
            NewPlan = { trigger = PlanTrigger[rp,l_x]
                        preconditions = None
                        plan body = goal to achieve
                           Plan[ap,l_x+1] }

        add NewPlan to Agent PlanLibrary

    # +!update(X[ap(T)])[ap]: not
       active_perception.isUpdated(X[ap(T)]) <- ?X[ap].
    UpdatePlan = {  trigger = UpdateTrigger(X)[ap]
                    preconditions = X is not updated
                    plan body = calls AP plan for X }

    add UpdatePlan to Agent PlanLibrary
    return PlanLibrary
```

Listing 3.14 – Jason program after syntactical changes (Approach 3)

```
!g[ap, l_1].

+!g[ap, l_1]
    <-  !update(b[ap(1000)])[ap];
        !update(d[ap(3000)])[ap];
        !g[rp, l_1].

+!g[ap, l_2]
    <-  !g[rp, l_2].

+!g[rp, l_1]: b & d
    <-  .print("GOAL G1").

+!g[rp, l_1]
    <-  !g[ap, l_2].

+!g[rp, l_2]
    <-  .print("GOAL G2!").

+?b[ap]
    <-  .time(HH,MM,SS,MS);
        +b[ap(_),lu(HH,MM,SS,MS)];
        .print("Active perception plan for b").

+?d[ap]
    <-  .print("Active perception plan for d").

+!update(X[ap(T)])[ap]: not
   active_perception.isUpdated(X[ap(T)])
    <- ?X[ap].

+!update(X[ap(T)])[ap].
```

4 ARCHITECTURE FOR PROGRAMMING BDI AGENTS FOR ROBOT APPLICATIONS

As discussed in Section 2.4, there are already some related works that use Jason for programming the cognitive part of robotics applications. However, those works provide *ad hoc* solutions regarding the control of the robot's hardware devices, therefore a significant part of the software related to this integration must be always created. In case the device is replaced, again the software must be re-programmed. Thereby, to circumvent this problem, we propose a set of standards for using Jason with ROS, including the creation of a dedicated and configurable ROS node named *HwBridge* that allows the integration of Jason and the robot.

An architecture composed of four ROS nodes is proposed to integrate Jason and ROS, as shown in Figure 4.1. The *Agent* node, as the name suggests, is the agent itself and is implemented with the Jason language. The *HwBridge* node serves as a bridge between the agent and the hardware, it translates the messages and publishes them in the correct topics. The *Hardware Controller* node is the one that manages the hardware. The *Comm* node is responsible for the communication between agents, this can be via Ethernet, wifi etc. An implementation of the proposed integration is available at: https://github.com/jason-lang/jason_ros.

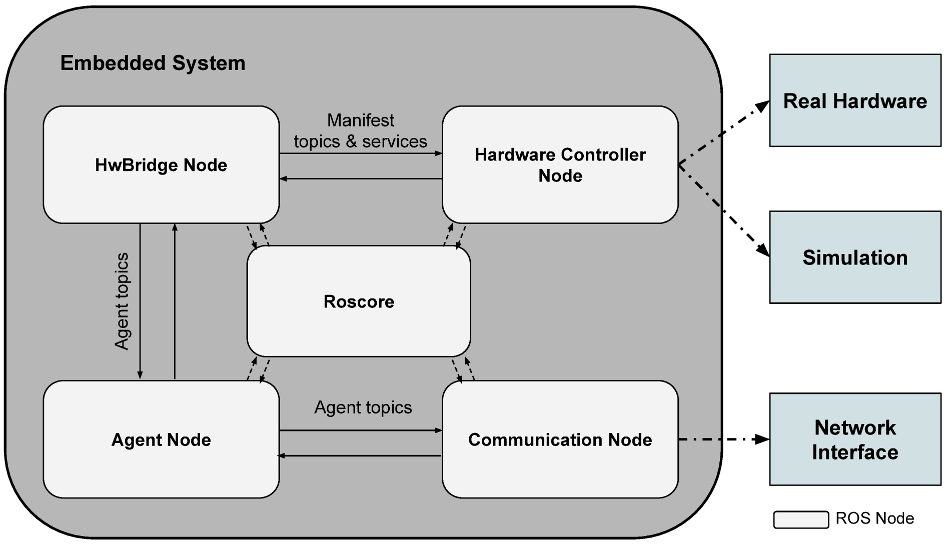


Figure 4.1 – System architecture

4.1 AGENT NODE

To allow the use of ROS by a Jason agent it was necessary (1) to specify and establish standards for ROS topics and messages; (2) to customize the agent architecture to include the functionalities defined in the first step.

4.1.1 Specifications and standards

For the definition of standards, the work of (MORAIS, 2015) was used as an starting point. The resulting specification of ROS topics and their definition can be seen in Table 4.1, also in the graph represented in figure 4.2.

Table 4.1 – Topics used by the agents

Topic Name	**Definition**
/jason/percepts	Subscribes to get new perceptions
/jason/actions	Publishes to send actions it wants to perform
/jason/actions_status	Subscribes to receive the status of an action sent
/jason/send_msg	Publishes to send message
/jason/receive_msg	Subscribes to receive message

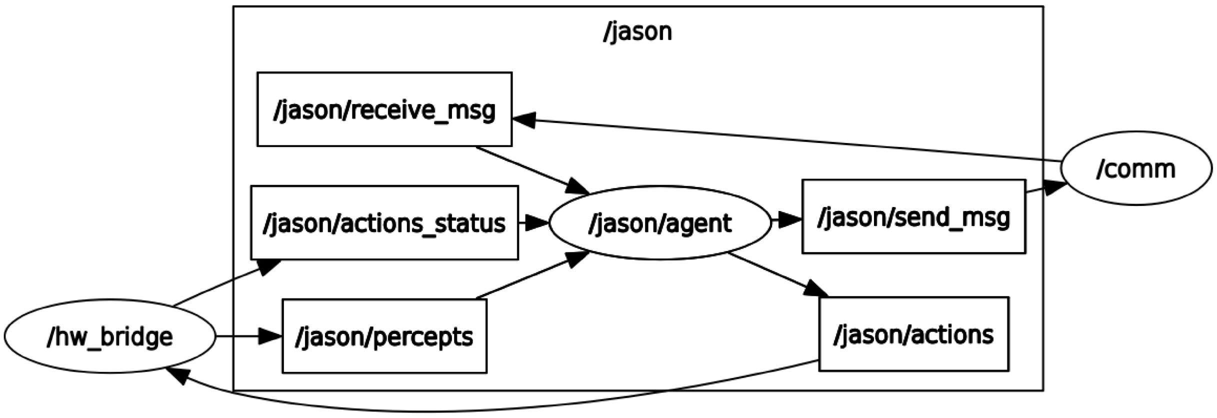


Figure 4.2 – Jason-ROS nodes and topics graph

To handle perceptions, the agent subscribes to the topic */jason/percepts* which uses a custom type of message called Perception (see listing 4.1), that is composed of 4 fields: header, name of the perception, perception parameters, and a boolean called update which indicates if the perception should be added or updated in the belief base.

Listing 4.1 – Perception message

```
Header header
string perception_name
string[] parameters
bool update
```

In order to perform an action, the agent publishes into the */jason/action* topic a message of the type Action (see listing 4.2), which contains 3 fields: header, the action name, and action parameters. Then, the agent subscribes to the topic */jason/actions_-status* to receive information about an action that was previously sent, the message type used is *ActionsStatus* (see listing 4.3), which also contains 3 fields: header, the result of the action, and its unique id.

Listing 4.2 – Action message

```
Header header
string action_name
string[] parameters
```

Listing 4.3 – Action status message

```
Header header
bool result
uint32 id
```

Regarding communication, when an agent wants to send messages to external agents it publishes it into the topic */jason/send_msg* using a custom type of message called Message (see listing 4.4). This message is composed of 2 fields: header and the data. In order to receive messages the agent subscribes to the topic */jason/receive_msg*, which also makes use of the Message type.

Listing 4.4 – Message message

```
Header header
string data
```

4.1.2 Agent architecture customization

With the specification completed and the standards defined, the agent architecture was modified to include the functionalities discussed, which was done by overloading the methods represented in Table 4.2. Jason is interpreted with the Java language thus these methods must be overloaded using Java. To accomplish that, since ROS does not provide support for using Java in its official distribution, a 3rd party Java ROS implementation, *rosjava* was used. It must be emphasized that for the Jason programmer this is all transparent, in other words, a user of this Jason-ROS integration does not need to modify any code in Java.

Table 4.2 – Overloaded methods

Method	**Customization**
init	Initialize a ROS node
act	Send actions via ROS
reasoningCycleStarting	Receive feedback of actions via ROS
perceive	Receive perceptions via ROS
checkMail	Receive msgs via ROS
sendMsg	Sends msgs via ROS
broadcast	Broadcast msgs via ROS

4.2 HWBRIDGE NODE

The *HwBridge* node is the main advantage in relation to the architecture proposed by (MORAIS, 2015), this node has a similar purpose that the ones he calls decomposers and synthesizers. As discussed in Section 2.4, they are used as intermediary nodes to translate the information between the agent and the hardware. The biggest difference here is that instead of requiring that both of these nodes are programmed for each specific use case, depending on the hardware, a general purpose node (*HwBridge* node) is available and the only thing that has to be adjusted for each case is a couple of configuration files.

The communication with the *Agent* node is done via the first three topics defined in Table 4.1, */jason/percepts*, */jason/actions*, and */jason/actions_status*. However, the information flow is in the opposite direction. The *HwBridge* node publishes the perceptions it receives from the *Hardware Controller* into the topic */jason/percepts*, it subscribes to the topic */jason/actions* to get the actions it needs to send to the *Hardware Controller*, and it publishes into the topic */jason/actions_ status* to inform the agent about the status of previously submitted actions.

To communicate with the *Hardware Controller* specific topics and services are used for each different perception and action, which are configured via two configuration files, the perceptions and actions manifest. These files contain all the information required to translate actions sent by the agent into understandable commands by the *Hardware Controller*, and to create perceptions understandable by the agent based on data published by the *Hardware Controller*.

The perception manifest contains the information about which topics the *HwBridge* node must subscribe to get each perception, and how to translate the data into a perception understandable by the agent. An example of perception manifest is shown in listing 4.5. In this case the perception comes from the topic */turtle1/pose* and it results in a perception such as *pose(3.0, 2.0, 0.3)* being sent to the *Agent* node, then the agent replaces in its belief base all the perceptions called pose by this new one, or add it in case none exists.

Listing 4.5 – Perception manifest

```
[pose]
name = /turtle1/pose
msg_type = Pose
dependencies = turtlesim.msg
args = x,y,theta
buf = update
```

The action manifest describes in which topics/services the *HwBridge* node should publish/request to perform each action, and how the data being sent must be set up. An example of action manifest can be seen in listing 4.6. With this configuration when the agent tries to perform an action, as for example *cmd_ vel(1.5, 0.0, 0.0)*, the *HwBridge* node would publish to the topic */turtle1/cmd_ vel* a message of the type Twist with its fields “linear.x=1.5”, “linear.y=0.0”, and “linear.z=0.0”.

Listing 4.6 – Action manifest

```
[cmd_vel]
method = topic
name = /turtle1/cmd_vel
msg_type = Twist
dependencies = geometry_msgs.msg
params_name = linear.x, linear.y, linear.z
params_type = float, float, float,
```

Figure 4.3 illustrates typical sequence of messages exchanged by the system nodes. Firstly, when an Agent sends an action, the *HwBridge* node translates the message and forwards it to the *Hardware Controller* node in the right topic/service, which then executes what is necessary to perform the action, and upon its completion or failure it informs the agent about its status. In another situation, when the *Hardware Controller* node publishes data into a topic associated with a perception, the *HwBridge* node interprets the information and, if the data is different from the last one received, it translates it into a digestible message and forwards it to the *Agent* node.

It is important to highlight that the the arrival of new actions and perceptions are handled with callbacks and in separate threads, which allows that they are processed concurrently, despite what is being shown in the diagram. All message exchanging are asynchronous, except when the *HwBridge* node sends an action to *Hardware Controller*, in this case it is synchronous.

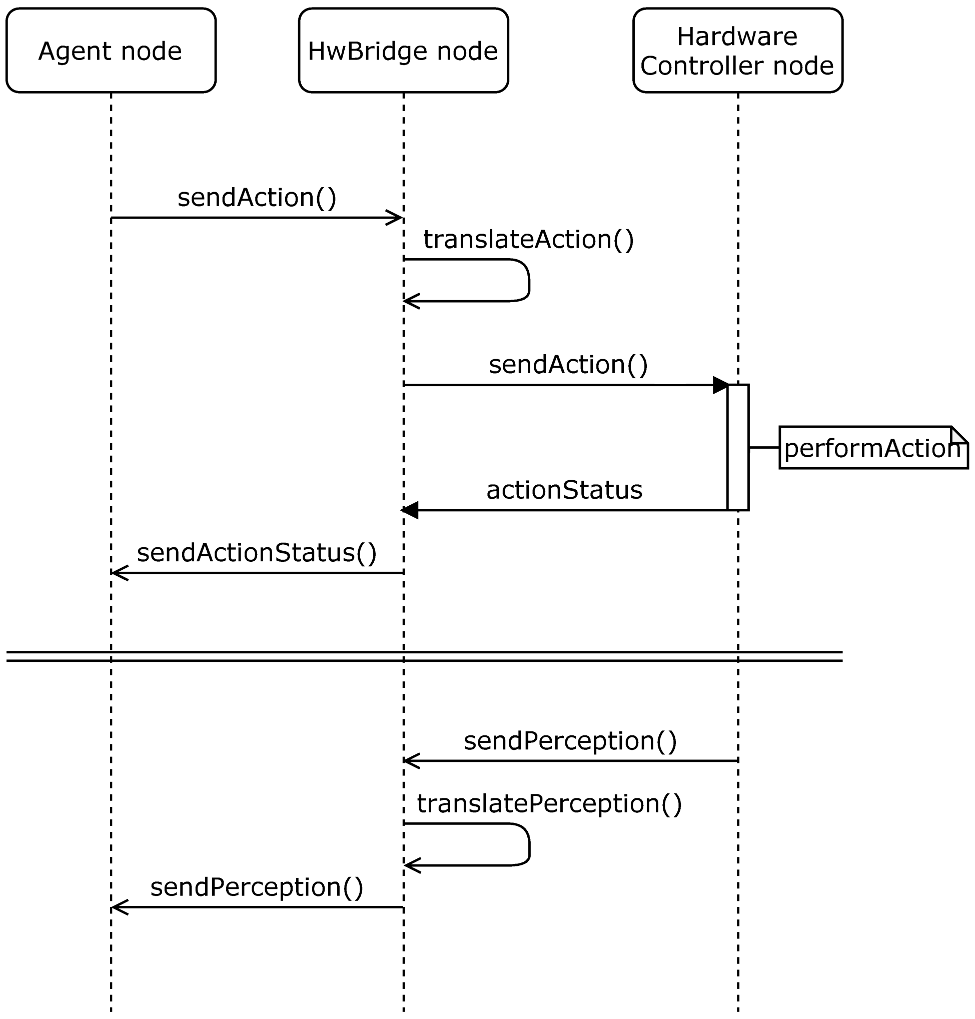


Figure 4.3 – Typical sequence of messages exchanged by the system nodes

It was decided to implement the *HwBridge* node using Python language because: (i) ROS offers native support for Python; (ii) since Python is an interpreted language, it facilitates to deploy the proposed solution into different embedded systems, as no (cross)compilation is needed.

4.3 HARDWARE CONTROLLER NODE

The *Hardware Controller* node is the one that, de facto, controls the hardware. Since a lot of robots nowadays already have a ROS package implemented for interacting with hardware, most of the time, there is no need to implement this node.

The proposed architecture is advantageous since the programmer has to only implement the *Agent* node, set up the perceptions and actions manifest and reuse an existent *Hardware Controller* node. This allows a person that only posses knowledge about Jason to program real robots, even for those with extended knowledge about different programming languages, it reduces the time needed for setting up a robotic system.

4.4 COMM NODE

Given that our proposal works in a distributed way, that is, there exists several ROS-Master nodes, it was created the *Comm* node. It serves as communication interface between agents, given that standard ROS protocols cannot be used within this scenario.

It is left to developers to decide which technology should be used to implement the *Comm* node, attempting that the message Data Field (listing 4.4) must comply with the Jason message specification. It consists in a string with the following format: "<*id*,*sender*,*itlforce*,*receiver*,*data*>". Where *id* is an unique identifier for the message, *sender* is the name of the agent sending the message, *itlforce* is the illocutionary force (SEARLE, 1965), *receiver* is the name of the agent receiving the message, and *data* is the information being sent.

5 EXPERIMENTS

This chapter describes the experiments conducted in this project and the methodology used for its realization and analysis. First, section 5.1 presents the experiments performed to evaluate the proposed architecture for integrating Jason and ROS. Afterwards, section 5.2 details the experiments performed to assess the active perception mechanism proposed in this work, and its impact on mobile robotics applications.

5.1 JASON-ROS EXPERIMENTS

In order to validate and evaluate the proposed architecture for programming intelligent robots based on the cognitive concepts of BDI, a MAS composed of unmanned aerial vehicles (UAVs) serve as testbed. First, all the configurations done in order to perform the experiments are described in section 5.1.1. Then, a simulation with a single UAV is performed an it is outlined in section 5.1.2, the main objective of this experiment is to serve as a proof of concept. Following, a more complex simulation involving multiple UAVs is performed in order to assess the advantages of using BDI agents as a programming paradigm instead of the more traditional approaches, as explained in section 5.1.3. The implementation of the experiments performed can be seen at: `https://github.com/Rezenders/mas_uav`.

5.1.1 Experiments setup

The first step is to enable the proposed architecture to control a single UAV. For that, a proper *Hardware Controller* node must be used. Fortunately, there is already implemented a ROS package called Mavros(MAVROS - ROS WIKI. . ., n.d.) that allows to communicate with flight controllers (FC), avoiding the need to develop a new *Hardware Controller* node. This UAV architecture differs from the one shown in Figure 4.1 with regard to the *Hardware Controller* node, which in this case is Mavros, and the UAV is the actual hardware.

It was necessary to properly configure the perceptions and actions manifests to allow the *Agent* node to command the UAV through Mavros. This enables the *HwBridge* node to communicate with the *Hardware Controller* node through the correct topics and services. The produced perception manifest is presented in Listing 5.1. It results in the following perceptions being sent to the agent:

- state(Mode, Connected, Armed)
  - Mode: indicates the current flight mode, can be one of: [‘RTL’, ‘POSHOLD’, ‘LAND’, ‘OF_LOITER’, ‘STABILIZE’, ‘AUTO’, ‘GUIDED’, ‘DRIFT’, ‘FLIP’, ‘AUTOTUNE’, ‘ALT_HOLD’, ‘LOITER’, ‘POSITION’, ‘CIRCLE’, ‘SPORT’, ‘ACRO’]
  - Connected: indicates if Mavros in connected to Ardupilot (boolean)
  - Armed: indicates if the motors are armed (boolean)
- altitude(Altitude)
  - Altitude: current UAV altitude (float)

- global_pos(Latitude, Longitude)
  - Latitude: current UAV latitude (float)
  - Longitude: current UAV longitude (float)
- home_pos(Latitude, Longitude)
  - Latitude: UAV latitude when motors were armed (float)
  - Longitude: UAV longitude when motors were armed (float)

Listing 5.1 – Mavros perception manifest

```
[state]
name = mavros/state
msg_type = State
dependencies = mavros_msgs.msg
args = mode, connected, armed
buf = update

[altitude]
name = mavros/global_position/rel_alt
msg_type = Float64
dependencies = std_msgs.msg
args = data
buf = update

[global_pos]
name = mavros/global_position/global
msg_type = NavSatFix
dependencies = sensor_msgs.msg
args = latitude, longitude
buf = update

[home_pos]
name = mavros/home_position/home
msg_type = HomePosition
dependencies = mavros_msgs.msg
args = geo.latitude, geo.longitude
buf = update
```

The actions manifest of Listing 5.2 sends the following actions to Mavros:

- set_mode(Mode)
  - Mode: represents the desired flight mode (‘RTL’, ‘POSHOLD’, ‘LAND’, ‘OF_-LOITER’, ‘STABILIZE’, ‘AUTO’, ‘GUIDED’, ‘DRIFT’, ‘FLIP’, ‘AUTOTUNE’, ‘ALT_HOLD’, ‘LOITER’, ‘POSITION’, ‘CIRCLE’, ‘SPORT’, ‘ACRO’)

Listing 5.2 – Mavros action manifest

```
[set_mode]
method = service
name = mavros/set_mode
msg_type = SetMode
dependencies = mavros_msgs.srv
params_name = custom_mode

[arm_motors]
method = service
name = mavros/cmd/arming
msg_type = CommandBool
dependencies = mavros_msgs.srv
params_name = value
params_type = bool

[takeoff]
method = service
name = mavros/cmd/takeoff
msg_type = CommandTOL
dependencies = mavros_msgs.srv
params_name = altitude
params_type = float

[setpoint]
method = topic
name = mavros/setpoint_position/global
msg_type = GeoPoseStamped
dependencies = geographic_msgs.msg
params_name = pose.position.latitude, pose.position.longitude,
    pose.position.altitude
params_type = float, float, float

[land]
method = service
name = mavros/cmd/land
msg_type = CommandTOL
dependencies = mavros_msgs.srv
```

- arm_motors(Value)
  - Value: arm or unarm the motors (boolean)
- takeoff(Altitude)
  - Altitude: desired altitude after taking off (float)
- setpoint(Latitude, Longitude, Altitude)
  - Latitude: set a desired latitude (float)
  - Longitude: set a desired longitude (float)
  - Altitude: set a desired altitude (float)
- land
  - Lands at the initial position of the UAV

With the manifests configured, the experimented system can be be launched. When the system is up and running the ROS package *rqt_graph* can be used to visualize the nodes and topics graph, the resulting graph can be seen in figure 5.1. It is possible to verify that the *HwBridge* node is subscribing to Mavros topics according to what is described in the perception manifest (see Listing 5.1). The topics and services used for the actions cannot be seen in the figure because, when implementing the *HwBridge* node, it was decided to create the publisher and service proxies dynamically as the actions are called. The *Comm* node will only be running when the application requires external communication.

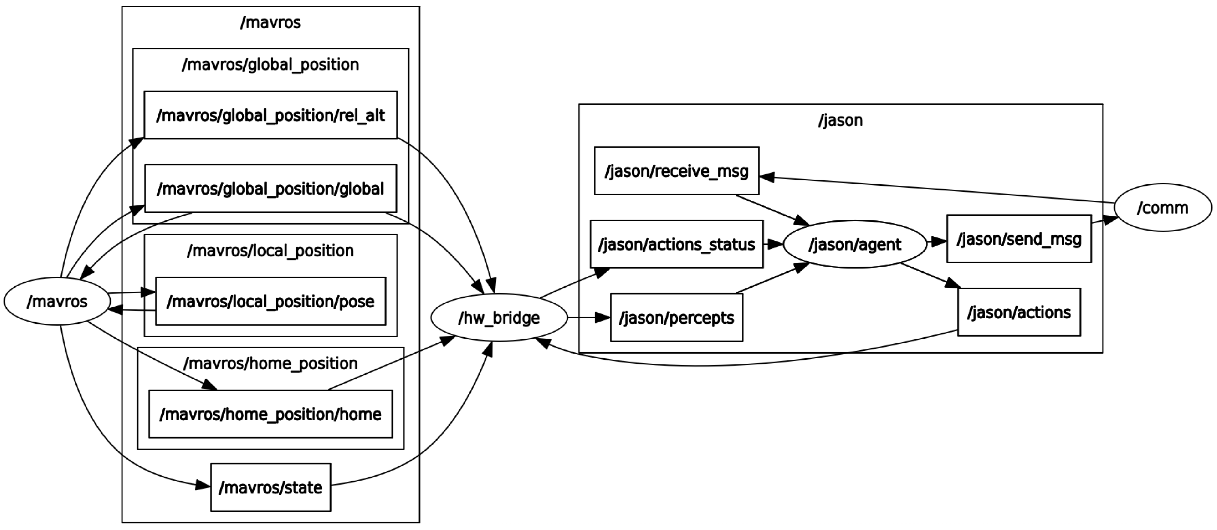
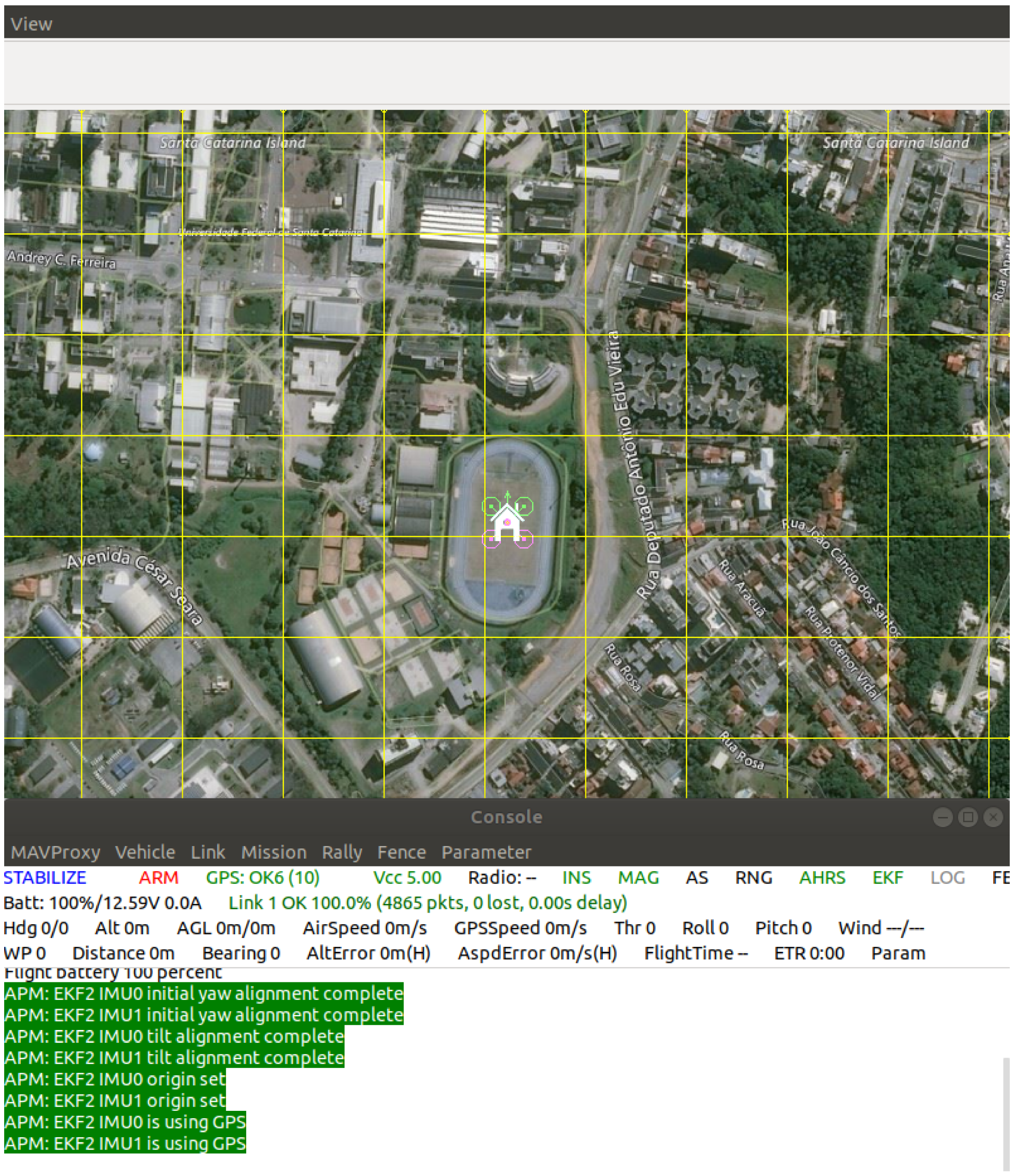


Figure 5.1 – Single UAV mission nodes and topics graph

Given that performing programming tests in real UAVs is very risky and unsafe, the present proposal was tested using simulated UAVs within a software-in-the-loop (SITL) environment. In such solution, the proposed agents execute within an embedded computing platform (one platform represents one UAV) and the UAV itself is simulated. For this, the ArduPilot simulator was used (see Figure 5.2). It emulates the FC and UAV dynamics, and communicates with Mavros in the same way as a real FC.

Two different embedded computing platforms were used on the present tests, as shown in Table 5.1.


View
Santa Catarina Island
Santa Catarina Island
Universidade Federal de Santa Catarina
Andrey C. Ferreira
Rua Deputado Antonio Edu Vieira
Avenida César Seara
Rua João Cancio dos Santos
Rua Aracuã
Rua Protenor Vidal
Rua Rosa
Rua Rosa
Console
MAVProxy Vehicle Link Mission Rally Fence Parameter

```
STABILIZE    ARM    GPS: OK6 (10)    Vcc 5.00    Radio: --    INS    MAG    AS    RNG    AHRS    EKF    LOG    FE
Batt: 100%/12.59V 0.0A    Link 1 OK 100.0% (4865 pkts, 0 lost, 0.00s delay)
Hdg 0/0    Alt 0m    AGL 0m/0m    AirSpeed 0m/s    GPSSpeed 0m/s    Thr 0    Roll 0    Pitch 0    Wind ---/---
WP 0    Distance 0m    Bearing 0    AltError 0m(H)    AspdError 0m/s(H)    FlightTime --    ETR 0:00    Param
Flight battery 100 percent
APM: EKF2 IMU0 initial yaw alignment complete
APM: EKF2 IMU1 initial yaw alignment complete
APM: EKF2 IMU0 tilt alignment complete
APM: EKF2 IMU1 tilt alignment complete
APM: EKF2 IMU0 origin set
APM: EKF2 IMU1 origin set
APM: EKF2 IMU0 is using GPS
APM: EKF2 IMU1 is using GPS
```

Figure 5.2 – ArduPilot SITL view

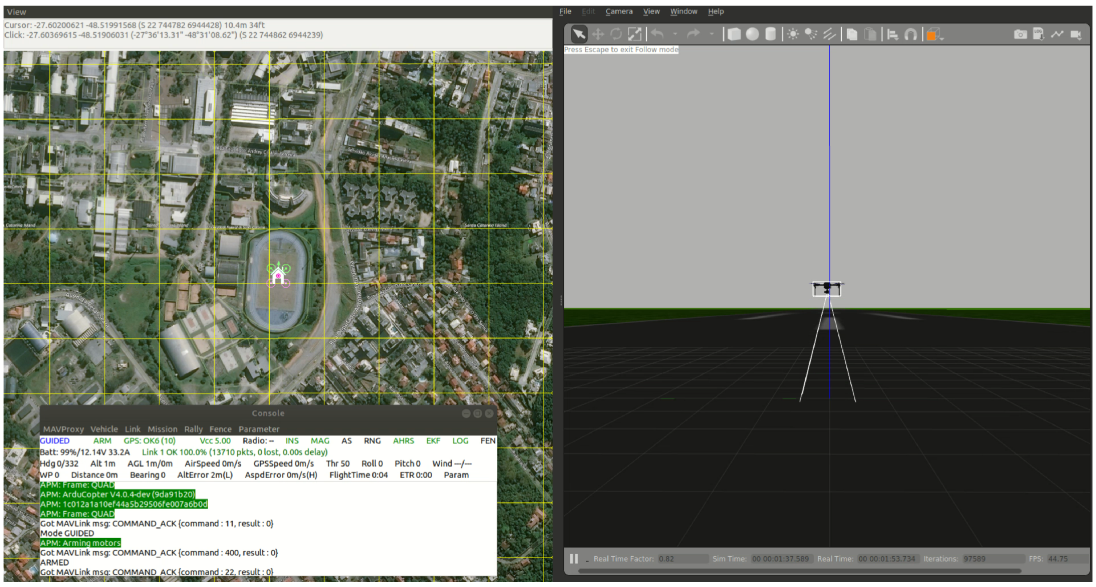
Table 5.1 – Adopted embedded computing platforms

	Beaglebone Black	Raspberry Pi 3 B+
CPU	ARM Cortex-A8 1GHz + 2x200-MHz PRUs + ARM Cortex-M3	4 Core ARM Cortex-A53 1.4GHz
GPU	200 MHz PowerVR SGX530	400 MHz VideoCore IV
SDRAM	512MB DDR3	1GB LPDDR2

5.1.2 Single UAV Mission

For this experiment a simple mission was performed: the UAV had to (1) takeoff; (2) fly to a predefined waypoint; (3) return to home; and finally (4) land.

The Beaglebone was initially used as embedded device. However, the application did run out of memory every time it was executed, not being able to complete the mission. Observing the experiments reported in (MENEGOL; HÜBNER; BECKER, 2018), which used a Jason solution without ROS in the same Beaglebone, the maximum memory utilization reached 85%. Considering that the proposed architecture uses ROS and that the



```
Cursor: -27.60200621 -48.51991568 (S 22 744782 6944428) 10.4m 34ft
Click: -27.60369615 -48.51906031 (-27°36'13.31" -48°31'08.62") (S 22 744862 6944239)
```

```
GUIDED      ARM    GPS: OK6 (10)      Vcc 5.00    Radio: --    INS    MAG    AS    RNG    AHRS    EKF    LOG    FEN
Batt: 99%/12.14V 33.2A    Link 1 OK 100.0% (13710 pkts, 0 lost, 0.00s delay)
Hdg 0/332    Alt 1m    AGL 1m/0m    AirSpeed 0m/s    GPSSpeed 0m/s    Thr 50    Roll 0    Pitch 0    Wind ---/---
WP 0    Distance 0m    Bearing 0    AltError 2m(L)    AspdError 0m/s(H)    FlightTime 0:04    ETR 0:00    Param
APM: Frame: QUAD
APM: ArduCopter V4.0.4-dev (9da91b20)
APM: 1c012a1a10ef44a5b29506fe007a6b0d
APM: Frame: QUAD
Got MAVLink msg: COMMAND_ACK {command : 11, result : 0}
Mode GUIDED
APM: Arming motors
Got MAVLink msg: COMMAND_ACK {command : 400, result : 0}
ARMED
Got MAVLink msg: COMMAND_ACK {command : 22, result : 0}
```

Figure 5.3 – ArduPilot SITL + Gazebo view

Beaglebone only has 512MB of RAM, such behavior was not a surprise

To overcome this drawback, the same tests were repeated using the Raspberry Pi 3, which has twice the memory size. Thereby the UAV successfully completed the mission every time the application was executed. This shows the feasibility of integrating Jason and ROS within an embedded device with more than 512MB of RAM.

Afterwards, in another experimentation round, the Jason agent was replaced by a Python program in charge of performing the same mission. It was decided to maintain the same *HwBridge node* to keep everything else similar in the experiment, but the *Agent node* (Jason or Python). The Python program was also able to successfully complete the mission.

Info from CPU and memory usage collected during the execution of both agents are shown in Table 5.2. Just like expected, the Jason implementation requires more computational resources than the Python one, 256% more CPU and 231% more memory.

Table 5.2 – Results for Single-UAV mission

Approach	**CPU [%]**			**Memory Usage [%]**		
	Mean	**Std**	**Max**	**Mean**	**Std**	**Max**
Python	1.65	0.20	2.61	17.45	0.00	17.45
Jason	4.22	2.25	14.59	40.37	0.06	40.51

Another point of comparison between both agents is qualitative, and regards the easiness of programming the corresponding mission in Jason versus Python. To support this subjective analyses some parameters were targeted, such as the size of the programs, their number of lines, and their number of words. For the measurement of the source files size they were compressed using *gzip* to reduce the influence of line breaks and blank spaces. As can be seen in Table 5.3, the size of the source file, number of lines, and number of words of the Jason program is smaller than the one in Python. This may be considered as an indicator that the Jason approach has better readability and maintainability, and one can also infer that it is easier to program.

However, it should be noted that the Jason approach requires the perception and action manifests to be properly set up. Besides, *rosjava* needs to be installed and configured. This results in development and execution overhead in the side of the proposed architecture, which must be properly balanced in too simple applications.

In order to better assess the complexity gap (difficulty) in between programming using Jason versus Python, a more elaborate experiment was developed, as follows.

Table 5.3 – Programs metrics in S-UAV mission

Approach	**Size (bytes)**	**# of lines**	**# of words**
Python	518	49	112
Jason	344	26	64

5.1.3 Multiple UAV Mission

To better understand and evaluate the usage of Jason in more complex tasks, it was chosen to design an application that is already being explored in the real world, a search-and-rescue (S&R) mission where UAVs are being used to find victims in floods and then deliver them buoys.

In the context of S&R missions, it is really useful to have more than one UAV collaborating since when vehicles are equipped with buoys their flight autonomy time is reduced due to the increased payload. Hence, a good strategy to adopt is to have two types of UAVs working together: (i) the *Scouts* which are equipped with cameras and (ii) the *Rescuers* that are in possession of buoys, using the former to find victims and inform the latter about their location, which then deliver the buoys.

Thus, an application was designed to mimic a S&R mission that uses one *Scout* and two *Rescuers* agents working in cooperation. Firstly, the Scout takes off and flies over an area looking for victims. When a victim is located the agent informs the rescuers about the victim's position. When the rescuers receive information about a victim's location they negotiate to decide which one will deliver the buoy. The one that ends up in charge of the rescue takes off, flies to the designated position, drops a buoy, and then returns to the landing area to recharge and replace the buoy. For the sake of simplicity, scouts are only in charge to locate victims and the rescuers to drop buoys.

In this experiment each agent will be embedded in a distinct Raspberry device, in total 3, and the simulation will be running in a separate desktop computer. Another simplification done in this experiment is that the connection between the Raspberrys' is considered to be constant and without losses. The Raspberrys and the desktop were connected with each other via Ethernet. The *Comm* node was implemented to send/receive messages to/from other devices via UDP.

Like in the single UAV experiment, the agents performed the same mission using both Jason and Python. During the execution of both methods the CPU and memory usage were monitored and the data collected can be seen in Table 5.4. As expected, Jason uses more CPU and memory than Python, 2.31 and 2.30 times respectively. Again, this is not a problem for embedded platforms such as Raspberry Pi 3.

Table 5.4 – Results for Multi-UAVs mission

Approach	**Agent**	**CPU [%]**			**Memory Usage [%]**		
		Mean	**Std**	**Max**	**Mean**	**Std**	**Max**
Python	Scout	1.73	0.17	2.30	19.78	0.00	19.78
	Rescuer 1	1.78	0.42	4.02	19.61	0.00	19.64
	Rescuer 2	1.76	0.27	3.67	17.99	0.00	18.01
	ALL	**1.76**	**0.31**	**4.02**	**18.93**	**0.85**	**19.78**
Jason	Scout	4.68	1.56	8.61	43.81	0.06	43.87
	Rescuer 1	3.95	1.57	16.00	44.00	0.06	44.15
	Rescuer 2	3.94	1.08	8.86	43.14	0.03	43.18
	ALL	**4.07**	**1.42**	**16.00**	**43.64**	**0.40**	**44.15**

Regarding the easy (facility) of programming, considering the subjective analysis of the authors, in this experiment it was undoubtedly easier to program the behaviour logic using Jason. But still, in order to support this statement, for each approach the size of the source files, number of lines, and number of words of all agents were measured and its sum can be seen in Table 5.5. It can be noted that the Jason program is smaller, and contains less lines and words than the one in Python, which is an indicative of the Jason approach being easier to program. Another metric that can be used is that in the Python approach it was necessary to use multi-threading and locks, which made the programming more complex.

Table 5.5 – Programs metrics in M-UAVs mission

Approach	**Sum Size (bytes)**	**Sum # of lines**	**Sum # of words**
Python	3668	384	928
Jason	2473	260	569

5.2 ACTIVE PERCEPTION EXPERIMENTS

This section describes the experiments performed to test and evaluate the proposed implementations of active perception. Three distinct modifications of the search and rescue mission described in section 5.1.3 are carried out. The first consists of adding active perception to the action of dropping a buoy, the second of adding active perception to the communication between UAVs, and the third is a combination of the last two. The code of the experiments performed can be found at `https://github.com/Rezenders/active-perception-experiments`

5.2.1 Experiments setup

For the active perception experiments it was decided to use a more realistic scenario of the search-and-rescue mission described in section 5.1.3. For this, Gazebo is used to simulate a virtual world with victims (figure 5.4), where the scouts cover an area looking for victims in a boustrophedon path (figure 5.5), and the rescuers drop buoys for the victims (figure 5.6). The flight controller emulator used in the simulations presented in the previous section (ArduPilot) was replaced for the PX4, given the fact that it provides a better integration with Gazebo. Additionally, this modification also allows testing the Jason-ROS integration in a larger number of scenarios. Mavros is still used as *Hardware Controller* node to communicate with the FC unit.

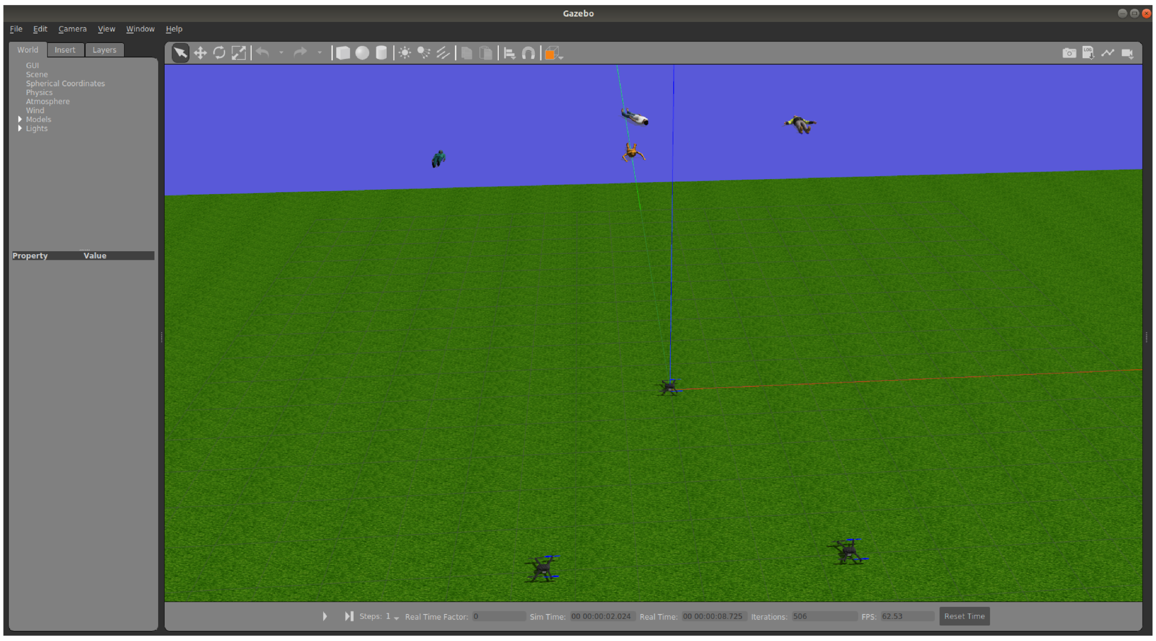


Figure 5.4 – Gazebo search-and-rescue world

The scout UAV uses the ROS package *boustrophedon_planner* (`https://github.com/Greenzie/boustrophedon_planner`) to generate the flight path. It takes as input a polygon of the area to be covered and the initial position of the robot, and its output is a list of setpoints of the calculated path. The aim here is to cover an entire search area, like depicted in the example of figure 5.5. This package is based on the algorithm "Convex Decomposition for a Coverage Path Planning for Autonomous Vehicles: Interior Extension of Edges" (L. D. NIELSEN; SUNG; P. NIELSEN, 2019).

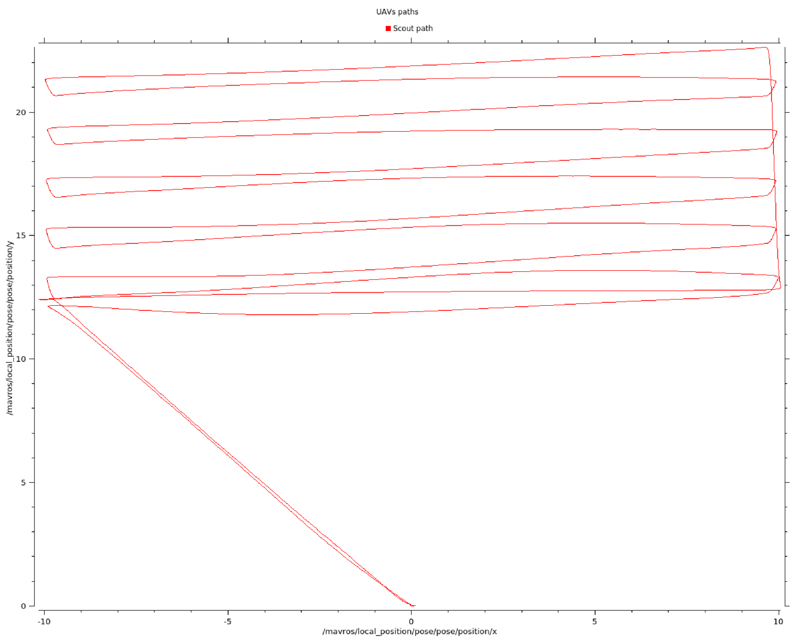


Figure 5.5 – Scout path

To detect victims a ROS node was created to mock the behaviour of using a camera and an object detection module. When the UAV flies near a victim the mock camera publishes a message informing the position of the victim in gazebo-coordinates. The agent subscribes to this topic to get the perception of the victims position. To drop buoys a gazebo-plugin was implemented, it subscribes to a ROS topic to get the position where a buoy should be spawned. The agent publishes to this topic in order to perform the action of dropping a buoy (figure 5.6).

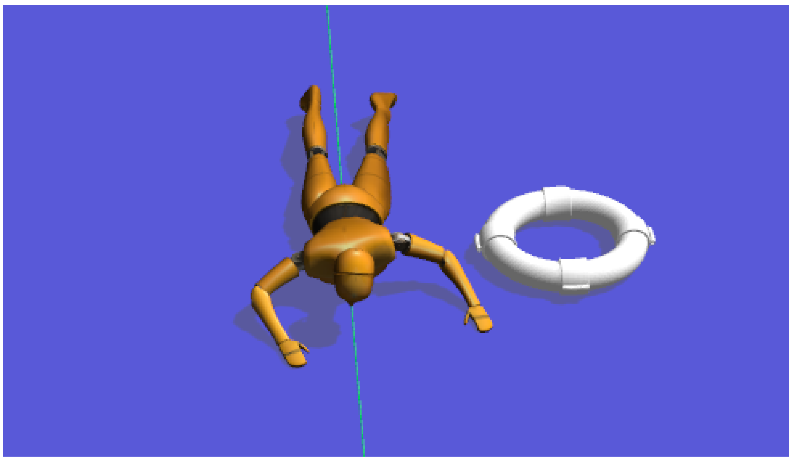


Figure 5.6 – Lifebuoy

5.2.2 First scenario

The first scenario is almost the same as the search-and-rescue mission described in section 5.1.3. The difference here is that the victims drown after a predetermined amount of time. To measure the impacts of using AP, search-and-rescue missions with only passive perception and others including active perception are performed and compared.

The active perception is inserted into this mission by adding as precondition a timed active perception belief (TAPB) of the victim's position to the action of dropping buoys. In this way, before dropping a buoy, the agent checks whether the belief of the victim´s position is a missing belief and, if it is, an active perception plan is executed to reveal the missing belief. In this case, the AP plan consists in turning the camera on and checking whether the victim is still in the informed position.

In terms of implementation, to add active perception capabilities to the agent it is necessary to include into the project one of the directives described in section 3.3, besides adding to the AgentSpeak(L) code an AP plan for each AP belief.

To understand how the dynamism of the environment influences in the comparison between using only passive perception and using active perception, the missions are performed for the two different victims drowning times represented in table 5.6. The times of the first configuration were chosen in order to leave the environment dynamic enough for some victims to drown during the execution of the mission, while the times of the second configuration were chosen so that the mission is completed before any victims drown. Resulting in the first configuration being more dynamic and the second less dynamic. The victims distribution in the scene are illustrated in figure 5.7.

To properly analyze the impact of using active perception in this search and rescue scenario, the experiments are performed and the number of victims successfully rescued is collected. Since the simulation is non deterministic, the results may vary for each run,

Table 5.6 – Victims drowning times

Setup	Drowning Time [ms]					
	Victim 1	Victim 2	Victim 3	Victim 4	Victim 5	Victim 6
1	70000	45000	48000	80000	60000	85000
2	100000	100000	100000	100000	100000	100000

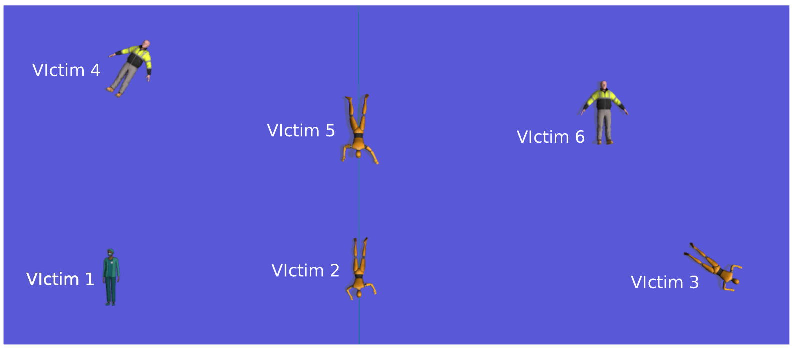


Figure 5.7 – Victims distribution

thus each experiment setup is repeated five times and the statistical results are presented. The efficiency is calculated by means of the number of victims rescued and number of buoys dropped. For the active perception mission, two different times are used for the lifetime of the timed active perception belief (TAPB), a shorter and a longer time, 5000 and 25000 milliseconds, respectively. Also, it is performed for both AP approaches from table 3.1, which the implementations are described in section 3.3. Obtained results with the respective configurations are presented in table 5.7.

Table 5.7 – Scenario 1 results

Experiment	Setup	AP	TAPB lifetime [ms]	Victims rescued			Buoys dropped			Efficiency w/ mode
				Mean	Std	Mode	Mean	Std	Mode	
1	1	NO	-	2.80	0.45	3	6.00	0.00	6	0.5
2	1	Approach 3	25000	2.80	0.45	3	4.00	0.71	4	0.75
3	1	Approach 3	5000	3.80	0.45	4	4.00	0.71	4	1
4	1	Approach 2	5000	3.60	0.89	4	3.80	0.45	4	1
5	2	NO	-	6.00	0.00	6	6.00	0.00	6	1
6	2	Approach 3	5000	6.00	0.00	6	6.00	0.00	6	1

For the first victims setup, experiments 1 to 4, it is possible to notice that experiments 1 and 2 had a worse result than 3 and 4, with most of the time 3 and 4 victims being rescued, respectively. In the second victims setup, experiments 5 and 6, there are no difference in the number of victims rescued between the experiment with active perception and the one with only passive perception.

In experiment 1, UAVs do not perform active perception before dropping the buoy. With this, the rescuers sometimes drop a buoy for a victims who have already drowned and, when a buoy is dropped, the UAV must return to its initial position and land to load a new buoy. This consumes a precious time that could be used to rescue a victim who has not yet drowned.

Experiment 2 has a similar problem, since the TAPB lifetime is long, the active perception pressure is not enough and sometimes the rescuers do not perform AP when necessary. Again, time is wasted dropping buoys for victims who have already drowned.

In experiment 3, the TAPB lifetime is shorter and consequently the active perception pressure is higher. With this, AP is always carried out before throwing a buoy which ensures that no buoy will be thrown in vain, increasing the efficiency of the mission. In this specific case, constantly performing active perception plans does not affect the agent's performance, since the AP plan has almost no cost, it consists only of turning the camera on and off and waiting for 1 second.

For experiment 4, active perception is replaced from approach 3 to approach 2, which has no impact on the mission result. This is expected, since the agent has only one relevant plan to drop the buoy that has a TAPB as a precondition, with the only difference between approaches 2 and 3 being how the relevant plans are revealed.

What happens in experiments 5 and 6 is that the environment is not changing fast enough for the active perception to have any impact on the mission, and since performing the AP plan has almost no cost it does not bring any disadvantages for the mission.

5.2.3 Second scenario

The second scenario is also a modification of the search-and-rescue mission detailed in section 5.1.3, but now the change is that instead of assuming that the network connection between the agents is constant and without losses the connection is limited to a predefined distance. Again, to measure the impacts of active perception on this scenario a mission with AP and one with only passive perception are performed.

To add AP into this scenario, a TAPB indicating if the UAV is in communication range is added as communication precondition. Thus, before sending any messages the agent checks if the communication range belief is a missing belief and, if it holds, the agent performs an AP plan to reveal it. The AP plan consists of flying to a predefined position where the UAV has communication range with most of the mission area.

In this experiment, the victims are distributed as shown in figure 5.7. The communication range is limited to 10m and the communication range TAPB lifetime is 1000ms in the scout agent. The AP mission is repeated for different TAPB lifetimes in the rescuers agents, and for both active perception approaches implemented.

To assess the results of the missions, the main aspect taken into account is the number of victims rescued. After that, it can be analyzed the number of buoys thrown and the ratio between the former and the latter, which indicates the efficiency of the system. Sometimes communication fails even with active perception, causing rescuers to drop a buoy for repeated victims. The setups and results of each experiment can be seen in table 5.8.

Table 5.8 – Scenario 2 results

Experiment	**AP**	**Rescuer range lifetime [ms]**	**Victims rescued**			**Buoys dropped**			**Efficiency w/ mode**
			Mean	**Std**	**Mode**	**Mean**	**Std**	**Mode**	
1	NO	-	1.80	0.45	2	1.80	0.45	2	1.00
2	Approach 2	5000	6.00	0.00	6	6.40	0.55	6	1.00
3	Approach 3	5000	6.00	0.00	6	6.60	0.89	6	1.00
4	Approach 3	1000	6.00	0.00	6	6.20	0.45	6	1.00
5	Approach 3	10000	6.00	0.00	6	7.40	1.14	7	0.86

Experiment number 1 relates with using passive perception only. It is possible to observe that with this configuration most of the time only 2 victims are rescued (mode), which is the smallest number among the 5 experiments. This happens because when the scout sends the location of victims 3 to 6 it is already out of the communication range.

Also, it can be noted that the efficiency calculated with the mode is 1, which means that when the mission is successfully performed no redundant buoy is dropped.

For all the experiments performed using active perception, 2 to 5, all the victims were rescued in all executions, since the mean is 6 and the standard deviations is 0. The only difference is regarding the efficiency of the system, in experiment 5 most of the time one unnecessary buoy is dropped, this is due to the larger TAPB lifetime which leads to an insufficient active perception pressure. Consequently, the rescuers sometimes do not perform active perception before communicating when it would be necessary, failing to inform that a victim was already rescued by them.

An interesting aspect to be highlighted in this scenario is that, although active perception has a cost related to flying to predefined positions, executing AP constantly does not bring disadvantages to the mission, as the environment is static.

5.2.4 Third scenario

The third scenario is a combination of the last two, the victims drown after a predefined amount of time and the network connection is limited to a certain distance. One more time, some missions with only passive perception and some with AP are performed to measure the impacts of using the latter.

The AP is included in the same way as the last scenarios, so in this case there are two timed active perception beliefs instead of only one. In this experiment, the victims are distributed as shown in figure 5.7, and the victims drowning times are represented in table 5.6. The communication range is limited to 10m, and for the AP missions the victim's position and the scout communication range TAPB lifetime are set to 5000ms and 1000ms, respectively. The AP approach, the victims drowning times, and the rescuer communication range TAPB lifetime are varied to verify their impact on the mission results. The experiments configurations and results are presented in table 5.9.

Table 5.9 – Scenario 3 results

Experiment	Setup	AP	Rescuer range lifetime [ms]	Victims rescued			Buoys dropped			Efficiency w/ mode
				Mean	Std	Mode	Mean	Std	Mode	
1	1	NO	-	1.60	0.55	2	2.00	0.00	2	1.00
2	1	Approach 2	5000	1.20	0.45	1	1.20	0.45	1	1.00
3	1	Approach 3	5000	1.20	0.45	1	1.20	0.45	1	1.00
4	1	Approach 3	10000	1.60	0.55	2	1.60	0.55	2	1.00
5	1	Approach 3	1000	1.00	0.00	1	1.00	0.00	1	1.00
6	2	NO	-	2.00	0.00	2	2.00	0.00	2	1.00
7	2	Approach 3	5000	4.40	0.55	4	5.20	0.45	5	0.80
8	2	Approach 3	10000	4.20	0.45	4	4.60	0.55	5	0.80
9	2	Approach 3	1000	3.60	0.55	4	4.20	0.45	4	1.00

Experiment 1 and 6 have only passive perception and their behavior is similar to the first experiment in scenario 2. The only difference is that in experiment 1, in some executions, the victims drown before being rescued, however, in most cases, 2 victims are rescued.

For the first victim's setup, the active perception experiments 2, 3, and 5 have worse results, only experiment 4 has the same performance as the experiment with only passive perception. This happens because the active perception plans for the communication range TAPB take a considerable amount of time to be performed, so before they have time to complete, the victims drown. Thus, it can be concluded that the active perception pressure for experiments 2, 3, and 5 is too high for an environment with this amount of dynamism.

For the second victim's setup, most of the time the active perception experiments 7, 8, and 9 have the same result, 4 victims rescued, which is better than experiment 6, 2 victims rescued. However, experiment 7 has a slightly higher average of victims rescued, which is an indicative that it is better suited for this victim's setup. It is interesting to notice that experiment 7 has an intermediary active perception pressure in comparison with experiment 8 and 9.

After analyzing the results from both victim's setup, one can conclude that the amount of active perception pressure that can be applied in order to improve the mission results depends directly on the dynamism of the environment and the cost, e.g. time, of the active perception plans. The lower the dynamism of the environment, the less the need for active perception, as beliefs maintain their credibility for longer. In more dynamic environments, the credibility of beliefs diminishes quickly, which requires active perception to be carried out more often. The problem of performing actions more constantly is that when it has a medium/high cost it can impact negatively on the performance of the mission. As an example, in this scenario the mission has a time restriction to be completed and the active perception requires an amount of time to be completed, thus, if the AP is performed too often it can reduce the performance of the mission, so it can be more advantageous to have a lower credibility than to spend time performing AP.

The experiments performed in this section demonstrate that the proposed active perception (AP) model can be an asset in some application scenarios, such as in search and rescue applications like scenario 2. Additionally, it is important to notice that the mechanism proposed in this Thesis allows adding AP to BDI agents in a simple manner, just by adding an annotation indicating that a belief is an active perception belief and another annotation expressing the belief lifetime, not requiring any code related to the active perception process to be programmed.

6 CONCLUSIONS

Conducting this work allowed me to conclude that it is not trivial to define an active perception (AP) model for BDI agents and to transform such model into a concrete implementation. It required several concepts related with BDI agents to be expanded and several new ones to be created. In addition, this could have been done in several different ways, leaving countless possibilities to be explored. Throughout this work it was detailed how and why some decisions were made to arrive at possible approaches to model and implement an AP mechanism integrated with the reasoning cycle of BDI agents.

With the experiments carried out, it could be noted that active perception is important in some scenarios, where through its inclusion the efficiency of the system is improved. This is due to the fact that in the proposed model of active perception, the agent does not consider that all the necessary information for the reasoning cycle is available or updated, trying to actively perceive what is necessary for its deliberation process. This results in more assertive decisions, since the deliberation process depends directly on the quality of the information that the agent has.

Also, it can be noticed that it is possible to use the proposed model and implementations of the active perception mechanism with BDI agents programmed with Jason to create complex behaviours for mobile robots. Of course, the same behaviors can be programmed without the proposed AP mechanism, but it would require the programmer to code everything related to the AP for each Jason program created, which requires more labor and is also more error-prone. In other words, our proposal simplifies the programming of AP for Jason.

In addition, it is possible to notice that an impactful aspect that must be taken into account when including AP is the trade off between the AP beliefs credibility and the associated cost of performing AP. This balance is regulated by the active perception pressure. The greater the AP pressure, the more often active perception is carried out and, consequently, the greater the credibility of beliefs. However, it should be noted that taking actions, mainly in the real world, has an associated cost, e.g. time, battery consumption.

Regarding the Jason-ROS integration, it can be concluded that it is possible to use Jason agents with ROS interface, even within an embedded platform. And that by using ROS it is possible to take advantage of existing ROS packages to simplify the development of complex tasks needed to program robots.

Also, the experiments shows that the use of BDI agents approach simplifies the development when compared to the conventional imperative programming, and has better readability and maintainability. Such conclusion is supported by metrics such as number of lines from the respective programs, their binary size, number of words, and code complexity in terms of the need to use multi-threading and locks. Another interesting point to highlight is that given the nature of the *Comm* node, it is possible to create a multi-robot system with heterogeneous agents, i.e., where some agents may be programmed using Jason and others using Python. A disadvantage of the proposed approach is that it uses more computational resources.

6.1 FUTURE WORKS

As future work, in the Jason-ROS integration, it is necessary to implement a way to get the responses of actions performed using a service, one way to do this would be to get the response via a *rostopic* and treat it as a perception. Also, it should be explored if the perception and actions manifests can be replaced by *rosparams* in order to make it even more compliant with ROS.

The Jason-ROS integration developed in this work uses The Robot Operating System (ROS) as robotics framework, this allows Jason to be used to program different types of robots and to leverage a lot of existing ROS packages. ROS was developed with the intention to ease the process of programming robots, and its fair to say that it accomplished its goal since it is one of the most used robotics framework today. The development of ROS adopted some characteristics such as: running a single robot, workstation-class computational resources on board, no real-time requirements, excellent network connectivity, etc. However, there are a lot of use cases for robots that do not comply with this characteristics. Therefore, ROS 2 is being developed to include the following use cases: teams of multiple robots, small embedded platforms, real-time systems, non-ideal networks, production environments, and prescribed patterns for building and structuring systems. Thereby, it would be interesting to also develop the Jason-ROS integration for ROS 2 in order to comply with future robots and technologies.

For the active perception, one topic that can be further addressed in the future is the concept of an belief being updated or outdated. In this work, a belief changes its state from updated to outdated when the time elapsed since the last update is longer than the belief lifetime, which is abrupt. Instead of having this binary changes it can be explored if it makes senses for the beliefs to have a fuzzy transition, having some kind of credibility (as addressed in section 3.1.1.6) that decreases over time, ranging from 0 (outdated) to 1 (updated). With this, more complex behaviours for the active perception mechanism can be designed, such as doing a fuzzy combination of all the beliefs in a plan preconditions in order to decide if active perception plans should be performed or not.

Regarding the implementation of the AP mechanism, more of the approaches proposed in section 3.2.7 should be implemented and tested. After that, it should be analyzed the viability of modifying the Jason architecture to include the active perception mechanism as a definitive solution, and if it would bring any advantages.

Lastly, the experiments performed in this work could be incremented to become more realistic. For example, a camera can be added to UAVs and actually perform object detection, currents can be added into the water to make victims move, victims can simulate to be swimming, etc. This could help understanding the impacts of the environment in the active perception. Also, this simulation could be used by other work that use a search and rescue scenario.

REFERÊNCIAS

BAJCSY, Ruzena; ALOIMONOS, Yiannis; TSOTSOS, John K. Revisiting active perception. **Autonomous Robots**, 2018. ISSN 15737527. DOI: 10.1007/s10514-017-9615-3. Cit. on p. 23.

BEST, Graeme; CLIFF, Oliver M, et al. Dec-MCTS: Decentralized planning for multi-robot active perception. **The International Journal of Robotics Research**, SAGE PublicationsSage UK: London, England, p. 027836491875592, Mar. 2018. ISSN 0278-3649. DOI: 10.1177/0278364918755924. Address: <http://journals.sagepub.com/doi/10.1177/0278364918755924>. Cit. on p. 27.

BEST, Graeme; FAIGL, Jan; FITCH, Robert. Online planning for multi-robot active perception with self-organising maps. **Autonomous Robots**, 2018. ISSN 15737527. DOI: 10.1007/s10514-017-9691-4. Cit. on p. 27.

BORDINI, Rafael H.; HÜBNER, Jomi F.; VIEIRA, Renata. Jason and the Golden Fleece of Agent-Oriented Programming. In: [**sineloco**]: Springer, Boston, MA, 2005. pp. 3–37. DOI: 10.1007/0-387-26350-0{_}1. Address: <http://link.springer.com/10.1007/0-387-26350-0_1>. Cit. on pp. 23, 46.

BORDINI, Rafael H.; HUBNER, Jomi Fred; WOOLDRIDGE, Michael. **Programming Multi-Agent Systems in AgentSpeak using Jason**. Chichester, UK: John Wiley & Sons, Ltd, Oct. 2007. (Wiley Series in Agent Technology). ISBN 9780470061848. DOI: 10.1002/9780470061848. Address: <http://doi.wiley.com/10.1002/9780470061848>. Cit. on pp. 23, 46, 47.

BRATMAN, Michael. **Intentions, Plans, and Practical Reason**. [**sinelocosinenomine**], 1987. ISBN 0-674-45818-4. Cit. on p. 25.

BRATMAN, Michael E.; ISRAEL, David J.; POLLACK, Martha E. Plans and resource-bounded practical reasoning. **Computational Intelligence**, John Wiley & Sons, Ltd (10.1111), vol. 4, no. 3, pp. 349–355, Sept. 1988. ISSN 0824-7935. DOI: 10.1111/j.1467-8640.1988.tb00284.x. Address: <http://doi.wiley.com/10.1111/j.1467-8640.1988.tb00284.x>. Cit. on p. 25.

J-M, Auberlet et al. **Improved Road Crossing Behavior with Active Perception Approach**. [**sineloco**], 2012. Address: <http://perso.lcpc.fr/roland.bremond/documents/TRB12_Ketenci.pdf>. Cit. on p. 27.

MAVROS - ROS WIKI. [**sinelocosinenomine**]. Address: <http://wiki.ros.org/mavros>. Cit. on p. 61.

MENEGOL, Marcelo S.; HÜBNER, Jomi F.; BECKER, Leandro B. Evaluation of Multi-agent Coordination on Embedded Systems. In: [**sineloco**]: Springer, Cham, June 2018. pp. 212–223. DOI: 10.1007/978-3-319-94580-4{_}17. Address: <http://link.springer.com/10.1007/978-3-319-94580-4_17>. Cit. on pp. 29, 65.

MORAIS, Márcio Godoy. Integration of a multi-agent system into a robotic framework : a case study of a cooperative fault diagnosis application. Pontifícia Universidade Católica do Rio Grande do Sul, Mar. 2015. Address: <http://tede2.pucrs.br/tede2/handle/tede/6396>. Cit. on pp. 29, 55, 57.

NIELSEN, Lasse Damtoft; SUNG, Inkyung; NIELSEN, Peter. Convex decomposition for a coverage path planning for autonomous vehicles: Interior extension of edges. **Sensors (Switzerland)**, MDPI AG, vol. 19, no. 19, Oct. 2019. ISSN 14248220. DOI: 10.3390/s19194165. Cit. on p. 69.

PANTOJA, Carlos Eduardo et al. ARGO: An Extended Jason Architecture that Facilitates Embedded Robotic Agents Programming. In: [**sinelocosinenomine**], 2016. pp. 136–155. DOI: 10.1007/978-3-319-50983-9{_}8. Address: <http://link.springer.com/10.1007/978-3-319-50983-9_8>. Cit. on p. 29.

RAFAELI, Niv; KAMINKA, Gal A. **Active Perception at the Architecture Level (Extended Abstract)**. [**sineloco**], 2017. Address: <www.ifaamas.org>. Cit. on pp. 27, 28, 37, 42.

RICCI, Alessandro et al. Environment Programming in CArtAgO. In: MULTI-AGENT Programming. [**sineloco**]: Springer US, 2009. pp. 259–288. DOI: 10.1007/978-0-387-89299-3{_}8. Cit. on p. 29.

SEARLE, John R. What is a Speech Act? **Perspectives in the philosophy of language: a concise anthology**, vol. 2000, pp. 253–268, 1965. Address: <https://pdfs.semanticscholar.org/a6c7/56a24ea621d3882d9b2baa8eb5352105a2cd.pdf>. Cit. on p. 60.

SO, Raymond; SONENBERG, Liz. The Roles of Active Perception in Intelligent Agent Systems. In: [**sineloco**]: Springer, Berlin, Heidelberg, 2009. pp. 139–152. DOI: 10.1007/978-3-642-03339-1{_}12. Address: <http://link.springer.com/10.1007/978-3-642-03339-1_12>. Cit. on pp. 23, 27, 31, 32.

UNTERHOLZNER, Alois; HIMMELSBACH, Michael; WUENSCHE, Hans-Joachim. Active perception for autonomous vehicles. In: 2012 IEEE International Conference on Robotics and Automation. [**sineloco**]: IEEE, May 2012. pp. 1620–1627. ISBN 978-1-4673-1405-3. DOI: 10.1109/ICRA.2012.6224879. Address: <http://ieeexplore.ieee.org/document/6224879/>. Cit. on pp. 27, 28.

VERBEEK, Marko. **3APL as Programming Language for Cognitive Robots**. [**sineloco**], 2003. Cit. on p. 29.

WESZ, Rodrigo Buenavides. Integrating robot control into the Agentspeak(L) programming language. Pontifícia Universidade Católica do Rio Grande do Sul, Mar. 2015. Address: <http://tede2.pucrs.br/tede2/handle/tede/6941>. Cit. on p. 29.

WEYNS, Danny; STEEGMANS, Elke; HOLVOET, Tom. Towards Active Perception In Situated Multi-Agent Systems. **Applied Artificial Intelligence**, Taylor & Francis Group, vol. 18, no. 9-10, pp. 867–883, Oct. 2004. DOI: 10.1080/08839510490509063. Address: <http://www.tandfonline.com/doi/abs/10.1080/08839510490509063>. Cit. on p. 27.

WOOLDRIDGE, Michael. Intelligent agents. **Multiagent systems**, MIT Press London, vol. 35, no. 4, p. 51, 1999. Cit. on pp. 23, 25, 26, 38.