Preprints of the 21st IFAC World Congress (Virtual)
Berlin, Germany, July 12-17, 2020

Embedded Architecture Composed of Cognitive Agents and ROS for Programming Intelligent Robots

Gustavo R. Silva [*] **Leandro B. Becker** [*] **Jomi F. Hübner** [*]

[*] *Department of Automation and Systems, Universidade Federal de Santa Catarina (UFSC), SC, Brazil.(e-mail: gustavorezendesilva@hotmail.com, leandro.becker@ufsc.br, jomi.hubner@ufsc.br).*

Abstract: This paper proposes and evaluates an embedded architecture aimed to promote the utilization of cognitive agents in cooperation with the Robotic Operating System (ROS), serving as an alternative for programming intelligent robots. It promotes the programming abstraction level in two directions. The first direction regards using cognitive agents facilities for programming the robots intelligence, consisting of its perceptions and related actions. The second direction exploits the facilities of using ROS layers for programming the robot interaction with its sensors and actuators. The paper reports experiments of using agents to command simulated UAVs while measuring performance metrics that allowed us to evaluate the benefits of the proposed architecture.

Keywords: BDI Agents, Robotics, UAVs, ROS, Jason

1. INTRODUCTION

When designing robots, one of the difficulties is to develop autonomous software that is capable to perceive the environment, reasoning about what it knows, and then choosing 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 et al., 2005, 2007). Among the benefits, agents can properly balance reactivity and pro-activeness, specially those agents built on top of the BDI (Belief, Desire, Intention) model.

This work proposes an architecture for programming intelligent robots based on the cognitive concepts of BDI. Experiments were performed to evaluate the feasibility of the developed architecture: whether it runs in embedded devices and can be practically used to operate robots. The paper also discusses the advantages and limitations of using BDI agents to program robots when compared to traditional imperative programming.

The reminder parts of this paper are organized as follows. Section 2 provides a background for what is discussed in this paper; Section 3 presents the related work that served as reference to tailor the proposed architecture; Section 4 contains the description of the proposed architecture used to integrate BDI agents and hardware; Section 5 details the performed experiments; and Section 6 outlines the conclusions and future works.

⋆ The authors acknowledge the support from CAPES and Pró-Alertas.

2. BACKGROUND

Multi-agents systems can be defined as systems that are composed of one or more intelligent agents. As Wooldridge (1999) stated, 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 definition: "An intelligent agent is one that is capable of flexible autonomous action in order to meet its design objectives", flexible means that it posses reactivity, pro-activeness, and social ability. *Reactivity* is the ability to perceive the environment and promptly react according to what is perceived; *pro-activeness* is the capability of taking the initiative to perform actions in order to achieve goals, producing a goal-driven behaviour; *social ability* is the capacity of interacting with other agents.

An important characteristic of intelligent agents is the balance between reactivity and pro-activeness Wooldridge (1999). If an agent is only reactive, it is difficult to envision how to perform actions to achieve long term goals, the agent will be simply reacting to the environment which likely 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 rarely check if the conditions that led it to commit to those goals still stands, which may result in an agent pursuing a goal that is no longer possible or relevant.

According to Bratman et al. (1988) an ideal but unrealistic solution to this problem would be to compute at each instant of time which is the best possible course of actions. 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), which in short is the combination, in the right amount, of reactivity and pro-activeness. *Beliefs* are the representation of the information the agent has about the world and itself, *desires* are world states the agent wants to achieve, and *intentions* are the desires that the agent decided and committed to accomplish.

Due to the characteristics of intelligent agents, especially BDI agents, they seem to be a good approach to develop complex systems composed of several entities (Bordini et al., 2005, 2007). With that in mind, the authors developed an agent-oriented programming (AOP) language called Jason, which is an extension of AgentSpeak(L) (Rao, 1996), to allow the development of MAS with real-world applications in an elegant manner and with a rigorous formal basis. Thus, Jason will be used as AOP the language in this work.

3. RELATED WORKS

The use of BDI agents to control robots is already being explored in related words (Verbeek, 2003; Morais, 2015; Pantoja et al., 2016; Menegol et al., 2018). For instance, Menegol et al. (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 this proposal keeps the hardware details transparent for the agent programmer, the integration works in an *ad hoc* manner. The high-level (BDI agent) and the low-level layers (robot hardware) are connected by specific protocols and ports – no standardization has been used or defined for using agents to control hardware. As consequence, if the hardware is exchanged a considerable part of the architecture must be reprogrammed. Therefore, it is not trivial to reuse the referred architecture. Also, it does not provide any interface for (re)utilizing robotic software developed by the roboticist community, for navigation, localization, and control purposes.

Wesz (2015) proposed JaCaROS, an architecture composed of Jason, CArtAgO (Ricci et al., 2009), and The Robot Operating System (ROS)(Quigley et al.) 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, in this paper 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.

4. JASON-ROS ARCHITECTURE

As discussed in Section 3, 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 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`.

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.

Specifications and standards For the definition of standards, the work of Morais (2015) was used as an starting

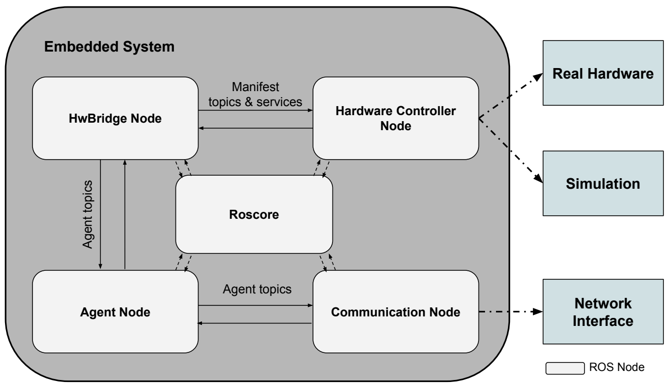

Fig. 1. System architecture

point. The resulting specification of ROS topics and their definition can be seen in Table 1.

Table 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

To handle perceptions, the agent subscribes to the topic */jason/percepts* which uses a custom type of message called Perception (see listing 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 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 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 3), which also contains 3 fields: header, the result of the action, and its unique id.

Listing 2. Action message
```
Header header
string action_name
string[] parameters
```

Listing 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). 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. Message message
```
Header header
string data
```

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 2. 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 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 3, 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 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 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 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 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 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 2 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.

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.

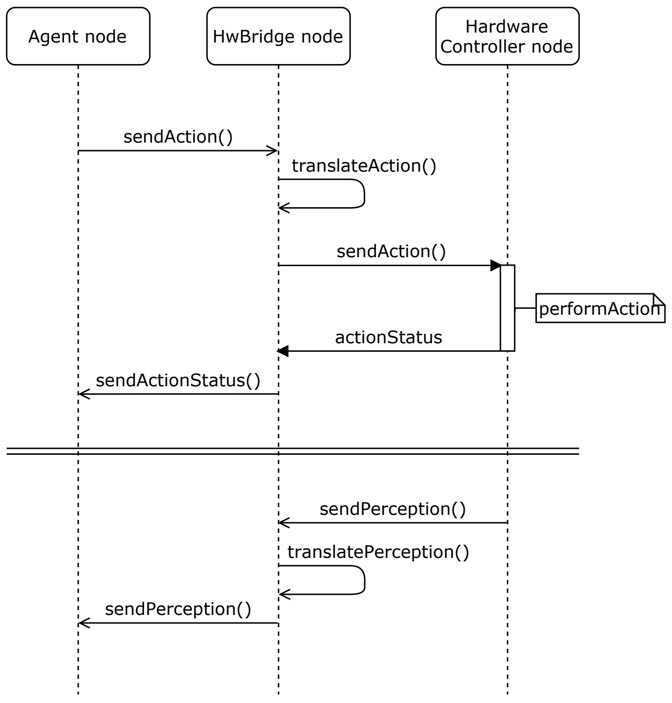

Fig. 2. Typical sequence of messages exchanged by the system nodes

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) 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

In order to validate and evaluate what is being proposed, a MAS composed of unmanned aerial vehicles (UAVs) will serve as a testbed. Firstly, the overall system architecture will be embedded in a board (e.g., beaglebone, raspberypi) and used to control a simulation with a single UAV, the main objective of this experiment is to serve as a test of concept. Then, 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. The implementation of the experiments performed can be seen at: `https://github.com/Rezenders/mas_uav`.

5.1 Single UAV Mission

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* that allows to communicate with flight controllers (FC), avoiding the need to develop a new Hardware node. This UAV architecture differs from the one shown in Figure 1 with regard to the *Hardware Controller* node, which in this case is *mavros*, and the UAV is the actual hardware.

This proposal was tested in a simulated environment (ArduPilot SITL) with a single UAV. The simulation did run in a desktop and all the other applications executed in the embedded device. 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.

A Beaglebone black 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. Considering that the proposed architecture uses ROS, such behavior was not a surprise if compared to results reported in Menegol et al. (2018). In such experiments the Jason solution (without ROS) reached a maximum memory usage of 85% using the same embedded device.

The same tests were performed once again using a Raspberry Pi 3 embedded device, and 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.

Afterwards, the Jason agent was replaced by a Python program in charge of performing the same mission. It was opted to maintain the same *HwBridge node* to keep everything else similar in the experiment, but the *Agent node* (Jason or Python). CPU and memory usage info collected collected during the execution of both agents are shown in Table 3. It is possible to note that Jason requires more computational resources than Python, 256% more CPU and 231% more memory. However, both agents can properly execute in the Raspberry Pi 3 device.

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 the size of the programs, their number of lines, and number of words were measured.

Table 3. 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

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 4, 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 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 4. Programs metrics in S-UAV mission

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

5.2 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.

As well as in the first experiment the agents will be embedded in three distinct Raspberry Pi 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. 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. 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 ease 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 6. 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 6. Programs metrics in M-UAVs mission

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

6. CONCLUSIONS AND FUTURE WORKS

The present work shows that it is possible to use Jason agents with ROS interface within an embedded platform. It also addresses the advantages of using such architecture to program intelligent robots.

Experiments using Hardware-in-the-Loop simulations shows that the use of BDI agents approach simplifies the development when compared to the conventional imperative programming. 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. However, this is not prohibitive in embedded platforms like Raspberry Pi 3.

As future work, it should also be possible to use ROS actions in addition to topics and services. 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.

We also intend to further explore the *Comm Node* (Section 4.4). It should be analyzed the possibility to implement it together with the *HwBridge Node*. Furthermore, it can be analyzed the possibility of deploying the *Comm Node* as a ROS package that already implements communication using Ethernet or Wifi networks.

REFERENCES

Bordini, R.H., Hübner, J.F., and Vieira, R. (2005). Jason and the Golden Fleece of Agent-Oriented Programming. 3–37. Springer, Boston, MA. doi:10.1007/0-387-26350-0_1.

Bordini, R.H., Hübner, J.F., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. John Wiley & Sons, Ltd, Chichester, UK. doi:10.1002/9780470061848.

Bratman, M.E., Israel, D.J., and Pollack, M.E. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(3), 349–355. doi:10.1111/j.1467-8640.1988.tb00284.x.

Menegol, M.S., Hübner, J.F., and Becker, L.B. (2018). Evaluation of Multi-agent Coordination on Embedded Systems. 212–223. Springer, Cham. doi:10.1007/978-3-319-94580-4_17.

Morais, M.G. (2015). Integration of a multi-agent system into a robotic framework : a case study of a cooperative fault diagnosis application. URL `http://tede2.pucrs.br/tede2/handle/tede/6396`.

Pantoja, C.E., Stabile, M.F., Lazarin, N.M., and Sichman, J.S. (2016). ARGO: An Extended Jason Architecture that Facilitates Embedded Robotic Agents Programming. 136–155. doi:10.1007/978-3-319-50983-9_8.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. (????). ROS: an open-source Robot Operating System. Technical report. URL `http://stair.stanford.edu`.

Rao, A.S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. 42–55. Springer, Berlin, Heidelberg. doi:10.1007/BFb0031845.

Ricci, A., Piunti, M., Viroli, M., and Omicini, A. (2009). Environment Programming in CArtAgO. In *Multi-Agent Programming*, 259–288. Springer US. doi:10.1007/978-0-387-89299-3_8.

Searle, J.R. (1965). What is a Speech Act? *Perspectives in the philosophy of language: a concise anthology*, 2000, 253–268. URL `https://pdfs.semanticscholar.org/a6c7/56a24ea621d3882d9b2baa8eb5352105a2cd.pdf`.

Verbeek, M. (2003). 3APL as Programming Language for Cognitive Robots. Technical report.

Wesz, R.B. (2015). Integrating robot control into the Agentspeak(L) programming language. URL `http://tede2.pucrs.br/tede2/handle/tede/6941`.

Wooldridge, M. (1999). Intelligent agents. *Multiagent systems*, 35(4), 51.