ARGO: An Extended Jason Architecture that Facilitates Embedded Robotic Agents Programming

Carlos Eduardo Pantoja (✉)[1,4], Márcio Fernando Stabile Junior[2], Nilson Mori Lazarin[1], and Jaime Simão Sichman[3]

[1] Centro Federal de Educação Tecnológica (CEFET/RJ), Brazil
`{pantoja, nilson.lazarin}@cefet-rj.br`
[2] Instituto de Matemática e Estatística, Universidade de São Paulo, Brazil
`mstabile@ime.usp.br`
[3] Escola Politécnica, Universidade de São Paulo, Brazil
`jaime.sichman@poli.usp.br`
[4] Universidade Federal Fluminense, Brazil

Abstract. This paper presents ARGO, a customized Jason architecture for programming embedded robotic agents using the Javino middleware and perception filters. Jason is a well known agent-oriented programming language that relies on the Belief-Desire-Intention model and implements an AgentSpeak interpreter in Java. Javino is a middleware that enables automated design of embedded agents using Jason and it is aimed to be used in the robotics domain. However, when the number of perceptions increases, it may occur a bottleneck in the agent's reasoning cycle since an event is generated for each single perception processed. A possible solution to this problem is to apply perception filters, that reduce the processing cost. Consequently, it is expected that the agent may deliberate within a specific time limit. In order to evaluate ARGO's performance, we present some experiments using a ground vehicle platform in a real-time collision scenario. We show that in certain cases the use of perception filters is able to prevent collisions effectively.

1 Introduction

Agents are autonomous and pro-active entities situated in an environment and are able to reason about what goal to achieve, based on its perceptions about the world [22]. In robotics, an agent is a physical entity composed of hardware, containing sensors and actuators, and software that is responsible for its reasoning. The Belief-Desire-Intention model (BDI) [3] is a cognitive approach for reasoning based on how information from the environment and the goals an agent has can activate predefined plans in order to try to achieve these goals. Jason [2] is an Agent-Oriented Programming Language (AOPL) that implements an AgentSpeak interpreter in Java, adopting the BDI cognitive architecture. However, programming robotic agents using Jason is a difficult task because a bottleneck can occur in the agent's reasoning cycle when the robot updates its belief base with perceptual information.

Javino [14] is a middleware that enables automated design of embedded agents using Jason. It allows agents to communicate with microcontrollers in hardware devices, e.g. Arduino. Both Javino and Jason can run embedded in a single-board computer such as Raspberry Pi (connected with n devices). However, when using several sensors, the agent's belief base generates events for each perception, which may compromise the robot execution time. In [20], perception filters were used to minimize the cost effects of processing all perceptions in simulation systems using Jason. The results showed that filters are able to improve agent's performance significantly.

Thus, in this paper, we present a customized Jason architecture for programming embedded robotic agents named ARGO[1], which uses a layered robot architecture separating the hardware from the reasoning agency. In ARGO, Javino enables processing data coming from sensors as perceptions in ARGO's agent reasoning cycle. Then, one can restrict the list of perceptions delivered by Javino based on filters designed by the agent's programmer. The main contribution of ARGO is to enable the use of perception filters for programming robotic agents, which reduces the cost of processing perceptions in BDI. Moreover, ARGO allows an agent to decide when to start or to stop perceiving, to fix the interval between each perception and to control the perceptual behavior by using Jason internal actions to filter perceptions at runtime.

In order to evaluate ARGO's performance, we also present some experiments using a ground vehicle platform in a real-time collision scenario constructed. We applied the experimental design methodology described by [12] to test and to statistically verify that in certain cases the use of perception filters reduces BDI processing time, thus preventing collisions effectively.

The rest of the paper is structured as follows. We briefly present in section 2 the Jason framework and the Javino middleware, and then explain how we can construct embedded robotic agents with these frameworks. In the sequence, perception filters are discussed in section 3. We then present ARGO architecture and its implementation in section 4. Our experiments, including the case study, the experimental design and our results, are presented in section 5. In section 6, we discuss related work. Finally, in section 7 we present our conclusions and further research.

2 Programming BDI agents

2.1 Jason

Jason [2] is an interpreter for an extended version of AgentSpeak [17], which is an abstract AOPL based on a restricted first-order language with events and actions. Created to allow the specification of BDI agents, Jason implements the

[1] Download available at http://argo-for-jason.sourceforge.net.

operational semantics of AgentSpeak and provides a platform for the development of multi-agent systems.

A Jason agent operates by means of a reasoning cycle that is analogous to the BDI decision loop [2]. First, the agent receives a list of literals representing the current state of the environment. Then, the belief base is updated based on the perceptions received. Each change in the belief base generates an event that is added to a list to be used in a posterior step. The interpreter checks for messages that might have been delivered to the agent's mailbox. These messages go through a selection process to determine whether they can be accepted by the agent or not. After that, one of the generated events is chosen to be dealt with and when it is selected, all the plans related to that event are selected. From these plans, a new selection is made to separate which of them can be executed given the current state of the environment. If more than one plan can be executed, a function selects which one will be executed. If the agent has many different foci of attention, a function chooses one intention among those for execution. The final step is to execute the first non-executed action from the selected intention.

2.2 Javino

Javino is a library for both hardware and software that implements a protocol for exchanging messages between the low-level hardware (microcontrollers) and the high-level software (programming language) with error detection over serial communication [14]. There are some communicating libraries in the literature, such as RxTx Library and JavaComm, based on serial ports. However, these libraries do not provide error detection and they use byte-to-byte communication. In both cases, the programmer needs to implement a message controller on the hardware layer in order to avoid losses.

The format of a message used in a communication by Javino is composed of 3 fields: preamble, size and message content. The preamble (2 bytes) identifies the beginning of a message that arrived through a serial port. The size field (1 byte) is calculated before any transmission informing the size of the message. The field message content (up to 255 bytes) carries the message to be sent.

Both the preamble and size fields identify errors in case of loss or collision of information during the message transmission. When a message arrives on the serial port, the receiver (either software-side or hardware-side) verifies the preamble. If it is correct, the receiver then counts the size of the message content field and compares it with the value of size field: if they don't match, the message is discarded. In the case of incomplete messages, the receiver also discards the message. Javino provides three different operation modes:

– the **Send** Mode assumes a simplex message transmission by software to hardware. It uses the *sendCommand(port, msg)* method to send a message to the hardware-side. This method returns a boolean value which gives a feedback about the successful transmission to the microcontroller. This feedback is necessary because the port serial can be locked by other concurrent transmissions. The software-side do not wait for answers from the hardware;

- the **Request** Mode assumes a half-duplex transmission between software to hardware, where the hardware sends an answer message. It uses the *requestData(port, msg)* method, that sends a message to the hardware-side through a serial port and returns a boolean value which checks if there is any answer sent by the hardware-side. The user is supposed to implement an answer message in the hardware-side using the *availableMsg()* method, that verifies if it exists a valid message from software-side, the *getMsg()* method, that gets the message sent by software-side and the *sendMsg(msg)* method, that sends a message to software-side;
- the **Listen** Mode assumes a simplex transmission by hardware to software. It uses the *listenHardware(port)* method to check if there is any message sent by the hardware-side. The Request and Listen modes get messages from hardware using the *getData()* method.

The Javino's protocol aims to be multi-platform and can be implemented using any programming language. The hardware-side library may be used in microcontrollers such as ATMEGA, PIC or Intel families. The software-side library may be coded in Java or in another programming language. In [14], it was developed a Java library for the software-side and an Arduino library for hardware-side. In this case, Javino requires both Python and pySerial installed to manage the serial port of an operational system.

2.3 Embedding robotic agents

Some previous research have tried to integrate robotic reasoning into hardware by using BDI agents. In [9], a framework was presented to provide a way of programming agents using AgentSpeak in Unmanned Aerial Vehicle in a simulator. The authors in [4] proposed an aquatic robot which uses Arduino together with BeagleBoard who could move from point-to-point deviating from obstacles. However, the reasoning was centralized on a computer using a Wi-Fi communication with the robot. All the decisions were sent to BeagleBoard and retransmitted, by serial communication, to Arduino, which held sensors and actuators. Another work published in [1] presented a grounded vehicle, which used Arduino and Jason to control sensors and actuators using a Java library for communication between the hardware and the Jason's environment. However, the agent's reasoning was still running on the computer. The messages to the hardware-side were sent from an Arduino connected to a USB port computer to another Arduino embedded on the robot using radio transmitters.

The work in [19] showed that it was possible to use BDI agents on embedded systems employing single-board computers. However, it was not presented an infrastructure to integrate BDI agents in a robot. Therefore, they simulated the environment on a computer to execute the decisions taken by the BDI agent.

Finally, a robotic agent platform using both Javino and Jason framework was presented in [14], which was an improvement of the platform presented in [1]. The authors used Raspberry Pi and Arduino together to provide a fully embedded BDI agent reasoning on a robot. In this case, Javino was integrated into the agent's simulated environment and the agent used a Jason external action to request the perceptions and a Jason internal action to control the actuators. In this architecture, the agent is responsible for controlling both sensors and actuators that are connected to the Arduino board and it is embedded in Raspberry Pi. The Arduino boards are connected to the USB ports of Raspberry Pi, thus, the agents use Javino to get perceptions from sensors and act with the actuators plugged in Arduino. The architecture worked in embedded robotic agents. However, according to the authors, when using too many sensors or plans in Jason code the agent's reasoning suffered a delay due to the cost of processing perceptions in Jason. We believe that using filters to overcome this issue could reduce the time employed in perceptions processing in BDI.

3 Perception filters

3.1 Filtering perceptions

Filtering perceptions is a widely discussed topic in MAS and Robotics. Some works try to provide an agent vision mechanism, which limits the agent range of vision simulating the human eye behavior such as [13]. In classical robotics, Kalman filters are often used to provide robot vision, playing an important role in the development of robotic platforms [5].

In [13], the authors present a technique for perceiving objects using Multi-agent Based Simulation (MABS), when agents are situated in open environments. The agents do not have access to all perceptions available in the simulated system. Adversely, they only have access to partial information about the environment determined by their vision sensor range area (modeled as a cone like the human eyes range of view). So, an agent can perceive only what is within the cone area in front of it and its decisions are based on what it can percept in the simulated system. The agent vision algorithm eliminates unseen items that are not in the sensor area and detects visually obstructed objects (objects that are completely behind another object). The algorithm verifies if an object is too far from the agent position: if this distance is less than a pre-defined range, the object is perceived, otherwise, it is not processed as a perception. The algorithm then verifies if part of an object is within the vision cone. In this case, the object is perceived. Finally, the algorithm verifies if the object is witinh the agent's vision cone and it is not obstructed to be perceived. If it is obstructed, the object is not perceived. The algorithm has as an input the environment's objects, and it returns the ones perceived by the agent. The work is specific for MABS where all the objects are pre-defined in the simulated system. So, if it is desirable to extend the solution to a real robotic domain, one needs to identify objects in the real world using a camera. The camera image can be considered as the agent's vision sensor. However, in order to use thise mechanism coupled with a BDI agent, like Jason, it would be necessary to transform the objects perceptions in Jasons beliefs.

Kalman filters variants are used for many problems in the robotic domain such as robot controlling, object tracking, data estimation and prediction, simultaneous localization and mapping (SLAM), visual navigation, among others. In robot vision for object detection and tracking, a Kalman Filter can be used to identify an object and track it based on a series of images, for instance captured from a camera. Path following can be obtained in a static road segment detecting the distance and the angle between the robot and a line using Kalman filters [5]. Since Kalman filters are based on mathematical approaches, they could also be used along with Jason in internal actions or in Jason's environment. In the same way, the objects perceived in the environment have to be transformed into Jason's beliefs.

3.2 Perception filters in Jason

In order to identify the critical points for performance in the Jason reasoning cycle, the work in [20] used a profiling tool to analyze a piece of Jason code. By measuring memory and CPU usage, the authors verified that two sections of the code were more time-consuming: the Belief Update Function (BUF) and the method responsible for the *unification* of variables in the plans and rules. These two methods generated a bottleneck, and depending on the specification of the agent, those methods could take up to 99% of reasoning time.

Given that Jason's default implementation assumes that everything that an agent can perceive in the environment will be part of its perception list, they proposed the inclusion of a perception filter between the perceive function and the update of beliefs before starting the reasoning cycle. This filter is responsible for analyzing the perception list received and for removing from the list those literals that are not interesting for the agent. This is done through filters defined by the agent designer which are described in XML format files and define restrictions on the predicate, variables and annotations of the beliefs.

Let us suppose a robotic agent that represents his beliefs about the environment by predicates like $p(d, v)$, where predicate p identifies the sensor, d the side of the robot where the sensor is located and v the value acquired by perception. An example of such a perception is shown in Figure 1.

```
temperature(right,36)
temperature(back,38)
light(left,143)
distance(front,227)
distance(right,30)
```

Fig. 1. Example of perception list represented as beliefs.

An example of filter that is used in the experiments section 5.1 is shown in Figure 2. This filter would remove all the perceptions originated from the temperature and light sensors and would also remove the perceptions from the distance sensors that are not in the front of the robot.

```
<?xml version="1.0"?>
<PerceptionFilter>
  <filter>
     <predicate>temperature</predicate>
  </filter>
  <filter>
     <predicate>light</predicate>
  </filter>
  <filter>
     <predicate>distance</predicate>
     <parameter operator="NE" id="0">front
     </parameter>
  </filter>
</PerceptionFilter>
```

Fig. 2. Example of perception filter.

Since the agent's intentions may change, the perceptions that are relevant for the agent may also change. To reflect these changes, a new Jason internal action called *change_filter* was also proposed in [20]. This action receives as a parameter the name of an XML file with the specific rules for the perceptions, and sets it as the current filter so that in the next reasoning cycle, the agent receives perceptions according to its new interests.

4 ARGO

4.1 Overview of a Robot's Architecture using Javino

A robotic agent is an embedded system where software and hardware components are integrated to provide sensing and operating abilities in real-time environments. For this, it is necessary to employ an architecture capable of facilitating the robot construction and programming. Hence, we propose an architecture for programming robotic agents where it is possible to design the robot platform independently from the reasoning agency, and then to integrate them using a protocol for serial communication.

The robot platform must be composed of sensors and actuators coupled to microcontrollers, where all the desired actions that the robot can perform in the environment and the percepts it can capture from sensors are programmed. In this case, our architecture translates raw data into a format for high-level programming language in the firmware, resulting in a performance gain for the agent's reasoning. Javino's protocol is responsible for sending these percepts using the serial port of the microcontroller. In this architecture, it is possible to use any kind of microcontrollers whereas it employs a library compliant with Javino's protocol. Afterwards, a MAS programming language is employed to allow the cognitive control of the robot platform. The chosen program language should be able to host the existing versions of Javino's protocol or to implement a new one. An overview of the architecture is shown in Figure 3.

The architecture is composed of three layers: hardware, firmware and reasoning. The hardware layer is responsible for mounting the robot platform, sensors, actuators and connecting them with respective microcontrollers employed. A single-board computer is used to connect all microcontrollers using USB and will be responsible for hosting the MAS. The firmware layer provides all actions that a robot can execute including procedures for both sensors and actuators and they are programmed directly in the microcontroller. Basically, these procedures send prepared raw data as percepts for the reasoning layer and receive agent's messages to perform some action, both using serial communication and the hardware-side of Javino's protocol.

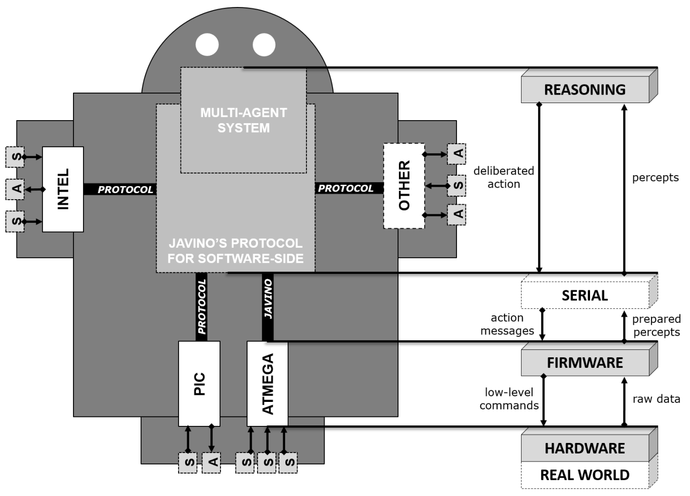

Fig. 3. Overview of a robot's architecture using Javino.

The reasoning layer represents the MAS's programming using a high-level language. The middleware in software-side transmits received percepts from serial port to the agent and sends action messages to the firmware layer. Depending on the AOPL chosen, it is possible to integrate received percepts directly into the agent's reasoning cycle or to use some structure to control the perception flow. As the architecture allows many microcontrollers in a robot platform, a strategy for capturing those percepts should be implemented. For example, it is possible to read all available serial ports one by one and after that to update the agent's percepts or to allow the agent decide which serial port it desires to use at a particular moment. Note that an agent cannot access more than one serial port at a time and more than one agent cannot access the same serial port at the same time.

In most of the commercial platforms, programmers do not have access to implementation details or they have to use an interface as a middleware for controlling the robot; on the other hand, these platforms also present a suite of functions to help in robot motion and planning. Our approach aims to be an architecture for open robot design to be used in cases where the programmer needs freedom to build his own prototype, using open platforms such as Arduino. The architecture is not bound neither to the MAS programming language, which can be interchanged, nor to the hardware adopted. However, it is necessary to adjust the raw data translation to percepts in the firmware layer, if the AOPL is changed.

4.2 ARGO Architecture

In the reasoning layer of our proposed robot architecture, it is necessary to adopt an AOPL which will be responsible for the cognitive reasoning of the robot platform. For this, we propose a customized Jason's architecture named ARGO employing perceptions filters and Javino integrated into Jason agent's reasoning cycle.

The BDI in Jason implies a high cost of processing the perceptions since for each one of the received literals an event is generated. In complex codes, plans may be added in running time, and a quite large intention stack is generated. In these cases, if the robotic agent has to achieve a goal within a time limit, it may not succeed. Our idea is to apply perception filters in these cases, so as to enable the agent to deliberate in time, in order to act in such critical applications. ARGO aims to be a practical architecture for programming automated embedded agents using BDI agents in the robotics domain.

In a MAS using ARGO, there are two types of agents which can be employed: ARGO agents and common agents provided by the Jason framework. An ARGO agent is able to directly control the actuators at runtime and it receives perceptions from the sensors automatically within a pre-defined time interval. Once the agent has received perceptions, it can filter them based on its actual configuration. It is also able to change its filters at runtime based on its needs (the same can occur when accessing its devices).

An ARGO agent is able to communicate with others common Jason agents, but only ARGO agents can control devices and receive perceptions from the real world. Because of this characteristic of the architecture, ARGO agents can send their received perceptions to other agents: they can either delegate for Jason agents the reasoning about these perceptions if idesirable or process all incoming percepts by themselves. In the first case, the ARGO agents are dedicated only to activate/deactivate devices, to get perceptions and to distribute perceptions to other agents instead of overcharging their reasoning by processing all received and filtered perceptions. In the latter case, a delay in some action response can occur if the processing cost of reasoning with the received perceptions is higher than the expected response time for example. An overview of ARGO can be seen in Figure 4.

An agent can assume to be an ARGO agent by defining the Argo architecture in the MAS design; otherwise, the standard agent architecture of Jason is automatically defined. An ARGO agent is supposed to connect to one or more devices at runtime by choosing which serial port it wants to access (until the limit of

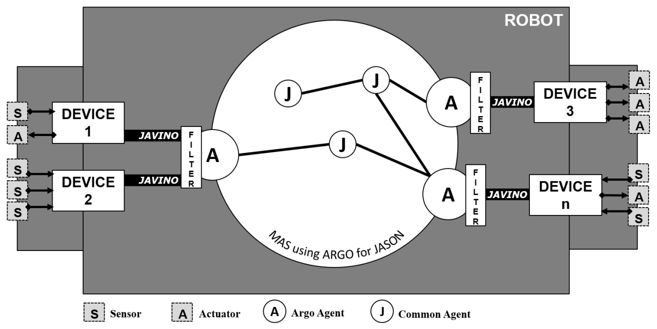

Fig. 4. ARGO overview.

127 serial ports); however it can only use one port at a time, for both sensing and acting. Besides that, different ARGO agents must not use the same serial port at the same time, because when exists a competition for communicating at the same port, there may be a data loss [8].

4.3 Internal actions

As mentioned before, an ARGO agent has the ability to control devices at runtime. It means that it can evolve in the real world using the robot's actuators and sensors. In practice, the agent controls devices using serial communication by choosing a serial port where the desired component is connected. Once defined the serial port, the agent can start receiving perceptions or can send a command to an actuator.

However, if the serial port is fixed for an agent, it will not be able to change it or to connect to other devices. Another issue is the perceiving ability: a Jason agent receives perceptions from sensors in every BDI cycle, even when it does not need them. Since an agent is an autonomous entity, we believe that an ARGO agent has to be able to decide when to perceive the real world at runtime. This means that the agent can start and stop perceiving from sensors when needed or it can define an interval for receiving these perceptions. Similarly, it can also directly control the actuators by defining at runtime which serial port to use. Moreover, bu using the ARGO architecture perception filtering technique, an agent can change at runtime its filters based on its needs, hence customizing its perception policy.

Therefore, we propose five internal actions for programming agents in Jason along with ARGO architecture. A Jason's internal action is a kind of action that is used to extend the agent capabilities. The proposed internal actions are:

1. **limit(x):** defines the sensing interval, where x is a value in milliseconds;
2. **port(y):** defines which serial port should be used by the agent, where y is a literal representing the port identification, e.g. COM8;
3. **percepts(open—block):** decides whether or not to perceive the real world;

4. **act(w):** sends to the hardware an action, represented by literal w, to be executed by a microcontroller;
5. **change_filter(filterName):** defines the filter to constrain perceptions in runtime, where filterName is the name of the XML file containing the filter constraints.

4.4 Customizing Jason for ARGO

In Jason's reasoning cycle, as mentioned in section 2.1, the agent gets its percepts from the simulated environment provided by Jason. We extended the reasoning cycle of Jason, shown in Figure 5, to providing a customized architecture for ARGO agents. First, Javino middleware is now responsible for getting percepts coming from low-level layers and sends them to the perceive step. Before being incorporated in the belief base, percepts can be filtered based on the agent's active filter. Then, filtered perceptions are processed and the reasoning cycle flows up to the act step, where the agent can perform basic Jason's actions or an action to control the actuators of the robot, which once more involves Javino middleware.

In order to create ARGO architecture, it was necessary to customize Jason framework, in particular by extending the *AgArch* class. This class is responsible for the Jason's native architecture and provides a list of perceptions sent by the Jason's environment in Java and the communication with other agents [2]. In the extended architecture, Javino middleware was inserted as a communication bridge to the hardware sensors and actuators. Besides that, the serial port identification had to be added to the native *AgArch* class in order to define to which serial port the Javino has to communicate.

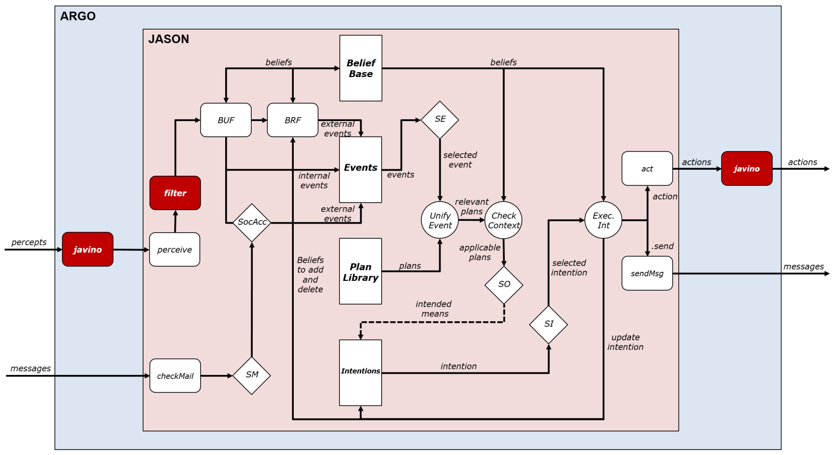

Fig. 5. ARGO reasoning cycle.

In the *TransitionSystem* class, two new attributes *blocked* and *limit* were created, as well as a new function *realWorldPerceptions*. The blocked attribute is responsible for blocking or unblocking the perceptions and the limit attribute specifies a time interval for perceiving the real world (data from sensors). The *realWorldPerceptions* verifies in each cycle *(i)* if the percepts are blocked; or *(ii)* if the time limit for the next perception has been reached. If the percepts are not blocked and the time limit was reached, Javino requests the percepts from sensors and sends them to the perceive method in Agent class.

Before the agent processes the percepts coming from Javino, they can be filtered using the method *filter* also implemented in the Agent class. In this case, all agents have the ability to filter percepts, because this method was implemented in the native Agent class. The modifications executed do not change Jason's original functionality, except for the simulated environment which is not used since Javino gets the percepts from the real world. We opted for creating a customized architecture instead of an infrastructure because the later one obliges all agents to be ARGO agents.

5 Experiments

5.1 Case study

In order to evaluate the overall architecture and to assure the impact of the perception filter, we assembled a robot composed of four distance sensors, four light sensors, four temperature sensors, an Arduino board and an Arduino 4wd chassis. A sensor of each type was placed in each of the four sides of the robot (front, back, right and left). The robot was placed on a flat surface two meters away from a wall. When started, the robot would perceive the environment and move forward at a constant speed[2] until the distance to the wall was less than a specified value. As soon as it perceived that the distance was smaller, the robot should stop. The robot can be seen in Figure 6.

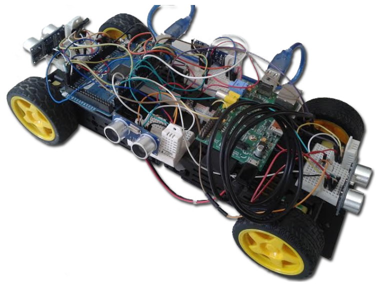

Fig. 6. The robot used in the experiments.

[2] The speed is about 10 cm/s and it is not used in the experiments since it is constant.

5.2 Experiment design

The experiment presented was designed based on the experimental design guidelines presented in [12]. According to the author, the goal of a proper experimental design is to obtain the maximum information with the minimum number of experiments. The procedure separates the effects of various factors that might affect the performance and allows to determine if a factor has a significant effect or if the observed difference is simply due to random variations caused by measurement errors and/or parameters that were not controlled. It is important to define the meaning of four terms:

1. *Response Variable* is the outcome of an experiment. In the experiments executed, the response variables are the processing time taken by the agent to stop after perceiving the wall and the distance it stopped from the wall;
2. *Factors* are the variables that affect the action response variable. Factors can be Primary or Secondary. Primary factors are those whose effects need to be quantified while secondary factors are those that impact the performance but whose impact we are not interested in quantifying. The primary factors chosen for this experiments were the distance the agent should stop from the wall, the time interval for receiving the perceptions and the filter used;
3. *Levels* are the values that a factor can assume. The factors and levels used are presented in Table 1;
4. *Replication* is the repetition of all or some experiments. If all experiments in a study are repeated three times, the study is said to have three replications.

Factor	**Levels**		
Distance	40 cm	80 cm	120 cm
Perception interval	20 ms	35 ms	50 ms
Filter	No filter	Front Side	Front Distance

Table 1. Factors and levels used for the experiment

The three filter levels represent the filter configurations that were used. "No filter" represents that the ARGO architecture did not make use of the perception filters, "Front Side" represents that the filter removed all the perceptions, except the ones from the sensors present on the front side of the robot. "Front Distance" represents that the filter removed all the perceptions, except the ones from the distance sensor present on the front side of the robot. Three executions were conducted for every combination of levels in Table 1.

5.3 Implementation

The agent has an initial belief that represents the distance limit from the wall that the robot should stop. It has also an initial intention that leads to a configuration plan, where it is defined both the serial port to which the Arduino board is connected and the perception interval limit. We ran experiments varying this value using 20 ms, 35ms and 50ms. Perceptions are then unblocked, since initially perception is blocked by default. The next action activates the filter that is responsible for filtering every perception except those from the front sensor of the robot. We ran experiments using a filter for all sensors except for the distance sensor in the front of the robot, and another round of experiments using no filters at all. The last action of the plan is an achievement action for a plan responsible for starting moving the robot.

The first action of the start plan is a message for the microcontroller to activate the motors and to move ahead. A belief with a status indicating that the robot is moving ahead is then added to the belief base of the agent. An achievement action for the moving plan is performed by the agent. The moving plan is responsible for verifying if the received filtered perception of the front distance sensor of the robot is greater than the initial belief of the distance limit. If so, the agent sends a message to the microcontroller to keep moving ahead. Otherwise, the robot crossed the distance limit and should stop. For this end, the agent sends a message to the microcontroller to stop the motors of the robot.

Some plans for using the temperature sensors and the light sensors were also provided. In this cases, when the perceptions of these both sensors are received, the agent sends a message to the microcontroller to turn on/off a specific led light positioned on each side of the robot, which informs when the received values crossed the limit specified in the agent code (in this case 100 for the light and 25C for temperature). The agent code is shown in Figure 7.

In our case, we used a single agent for controlling the robot, because we employed only one microcontroller where all the sensors and the robot's motors were connected to. If more than one agent tries to connect to the same serial port, conflicts arise. However, the architecture is sufficiently flexible to alow to develop a MAS for controlling the robot; in such a case, each employed ARGO agent could be responsible for controlling a kind of sensor (light, distance, and temperature), and the robot would be equipped with three microcontrollers.

5.4 Results

The first response variable analyzed was the distance the agent stopped from the wall. Figure 8 shows the results of all possible value combinations of the different factors presented in Table 1. Bars that do not appear in the Figure mean that the agent collided with the wall.

One should notice initially that in all cases, the agent that didn't filter its perceptions collided with the wall (there is no any blue bar in the Figure). In some cases, for instance the distance limit 120 cm, the agent with front side filter arrived eventually to stop before the wall; however, in these cases it stopped always closer to the wall when compared to the agent that used front distance filtering.

The agent using front distance filtering outperformed the others in quite all the experiments, and it was able to successfully stop before hitting the wall in all the experiments when the distance limit was 80 cm or 120 cm. Since this

```
value(40).
!config.

+!config: true <-
        .port(COM5);
        .limit(20);
        .filter(byValue);
        .percepts(open);
        !start.

+!start : true <-
        .act(front);
        +status(front);
        !moving.

+!moving: dist(f, X) &
                value(J) & X>J &
                status(front) <-
        .act(front);
        !moving.

+!moving: dist(f, X) &
                value(J) &
                X<=J & status(front) <-
        .act(stop).

-!moving <-
        !!moving.

+light(X,Y) : Y>100 <-
        .act(ledLightOn).

+light(X,Y) : Y<=100 <-
        .act(ledLightOn).

+temp(X,Y) : Y>25 <-
        .act(ledTempOff).

+temp(X,Y) : Y<=25 <-
        .act(ledTempOn).
```

Fig. 7. Agent code.

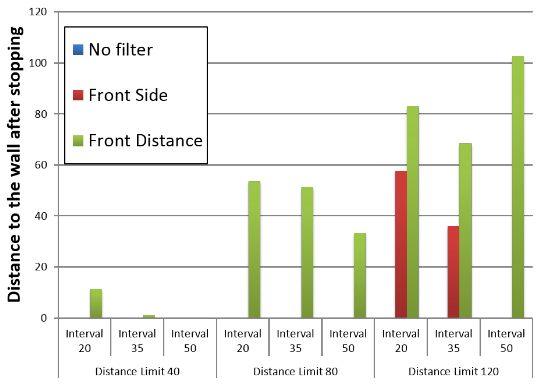

Fig. 8. Distance to the wall after stopping.

agent focuses only in perceptions coming from the front sensor, Jason's internal mechanism generates less events, and the agent can thus reason faster than an agent without any filter. However, in some experiments (for example, distance limit 40 cm and perception interval 50), neither agent could avoid the collision.

The second response variable analyzed was the elapsed time taken by the agent to stop after perceiving the wall. For this experiment, we calculated the variation assigned to each factor, as detailed in [12]. This statistical analysis is useful to check which factors are responsible for the differences in the response variable. The calculated values are presented in Table 2.

Factor	**Variation attributed**
Distance Limit (L)	1,415%
Perception Interval (I)	0,165%
Filter (F)	88,965%
Interaction between L and I	0,525%
Interaction between L and F	3,715%
Interaction between I and F	0,265%
Interaction between L and I and F	1,725%
Error	3,28 5%

Table 2. Variation assigned to each factor in the analysis of the response time.

The results confirm the importance of the perception filter in reducing the processing time, since almost all variation was attributed to this factor. This result suggests that ARGO architecture, by integrating Javino and the perception filters, can be used for developing embedded robotic agents in a way that the agent can benefit from the BDI architecture with a smaller influence of one of its major drawbacks that would be the high processing time.

6 Related Work

Robot architectures usually deal with platforms, sensors, actuators, programming language and reasoning mechanisms. One challenge is how to integrate these components in a way that a robot can deliberate to perform a task without failing to accomplish its goal. In [21] the authors propose a cognitive control architecture integrating knowledge representation of sensory and cognitive reasoning of a robotic agent using GOAL. The architecture consists of four decoupled layers: robot platform, robot behavioral control, environment interface and cognitive control. The robot platform employed was the humanoid NAO and it used URBI as middleware for interfacing with the robot's hardware via TCP/IP protocol. The robot behavioral control layer is responsible for processing sensory data and monitoring and executing behaviors. Besides, this layer communicates (using TCP/IP) with the reasoning and the robot platform layer, transmitting sensory data and actions execution respectively. The interface layer uses a translation mechanism between the sensory information acquired from the behavioral layer and the percepts sent to the cognitive layer. This layer is necessary because symbolic and sub-symbolic information can use different languages. The mechanism is based on a standard template using XML files mapping, which indicates how to map data but also when to do it. The cognitive control layer uses GOAL [10], which is a logic-based programming language for cognitive agents.

Similarly, ARGO's architecture also divides the robot programming into layers, separating sensory data from the agent's reasoning. We exploit the advantages of Jason extending it for programming robotic agents. ARGO provides three layers to be programmed: hardware, firmware and agent reasoning. Our proposed architecture provides a support for exchanging the hardware and the firmware without concerning with the reasoning layer; furthermore, it is possible to change the agent programming language without changing either the hardware or the firmware. This is possible because Javino is responsible for exchanging serial messages between these layers, and it does not link them to each other. We do not provide a translation mechanism in high-level layers because of the processing cost, which can affect the robot efficiency. However, the translation from raw data into percepts is done in the firmware layer. Since ARGO aims to be used in open platforms, the programmer must code the firmware layer. For commercial platforms such as NAO and Lego Mindstorms, a percepts mapping process must be provided.

Some other works also use Jason for this end, such as [16] [15]. In [16], CArtAgO [18] is used as the functional layer for providing artifacts that represent sensors and actuators of a robot, and Jason is used as the reasoning layer. Despite using artifacts, which is an interesting abstraction for the devices employed, the authors use a simulator named Webots and do not embed the MAS. In [15], the authors provide a Jason extension for ROS named Rason.

Javino's protocol provides a mechanism for avoiding noisy data in communication between the firmware and the reasoning layer. However, we do not treat noisy data coming directly from sensors, when they provide well-formed but wrong values. In [7], the authors present a programming language for cognitive robots and software agents using the 3APL [11] language, which implements a deliberation cycle for selecting and executing practical reasoning rule statements and goal statements. They also provide an architecture consisting of beliefs, goals, actions and practical reasoning rules as a mental state. The beliefs represent the robots percepts of an environment. The authors focused only on the programming constructs, they do not provide information about how a robot platform should interact with the high-level language.

In [6], a Teleo-Reactive (TR) extension for programming robots is presented, supported by a double tower architecture which provides a percepts handler that atomically updates the BeliefStore (a repository of beliefs). After that, the architecture reconsiders all rules affected by this change. The authors assert that actions and percepts can be dispatched through ROS interface to the robot platform. TR extension uses low-level procedures written in procedural programming languages for sensorial data and actuators actions. Concerning implementation, they used Qu-Prolog, simulators and Lego Mindstorms robots.

ARGO has the same intention of facilitating the programming of robotic agents providing a mechanism for automatically updating the agents belief base. The TR extension provides an inhibition process of some behaviors in response to percepts while ARGO provides a runtime process for filtering percepts that are not needed at a specific moment. Filtering perceptions in ARGO prevents unnecessary event triggering in the deliberative cycle of Jason, therefore, the agent deliberation should be more efficient.

7 Conclusions and further work

According to our studies, we concluded that using perception filters in applications where the response time is critical is an essential feature for agents developed in Jason. For this end, we have proposed the ARGO architecture. Perception filters enhance ARGO performance and make it practical feasible since it reduces significantly the perception processing and the events generated for each perception. Hence, it is a major feature in the ARGO architecture. However, in some applications, we believe that a delay in responses for perceptions processing can be tolerated and do not interfere with the goal of the MAS (i.e. applications where the response is not time-bounded).

In the ARGO architecture, an agent in a MAS can control different microcontrollers since the programming layer is independent of the microcontroller choice. This is an important issue because it is not bonded to a specific microcontroller technology allowing mixing other microcontrollers to a single prototype. Moreover, it is not bonded even to the MAS implementation, since it is possible to change the MAS code without changing the microcontroller code. This is possible because the microcontrollers run separately and they communicate with the MAS using serial communication. Basically, every ARGO agent requests perceptions or send actions acquiring a serial port connected to a microcontroller using Javino. This creates an uncoupled development environment for prototypes and robotic platforms using Jason framework, thus offering different ways of controlling low-cost boards with agents for several purposes.

In a MAS using ARGO it is possible to merge common agents (default agents from Jason framework) and ARGO agents (customized architecture) into a single project, but separating some responsibilities. Since just ARGO agents can get perceptions from the real world, a design issue may be raised: is it a better solution that uses only ARGO agents, possibly overcharging some ARGO agents or to delegate to common agents some processing information and deliberation responsibilities, thus isolating ARGO agents only to sensing and acting functions. We leave these questions as future work.

In order to achieve the customized architecture, some modifications were performed in the Jason framework. However, they do not change the original Jason functionalities, because they are just used for ARGO agents. Hence, there is no difference between the Jason framework and the ARGO architecture when a MAS uses using only common agents. The ARGO customized architecture benefits from the Jason framework extensibility ability to provide custom made architectures.

In our experiments, we show that applying the perception filter together with Javino reduces significantly the time of processing perceptions in Jason. In a real-time collision scenario, where the agent had to reason and stop before colliding with an obstacle placed at 120cm, 80cm, and 40cm, the experiments showed the agent was able to stop before colliding only by using perception filters . The ARGO architecture aims to provide programming structures that allow coding robotic agents using Jason. It means that an agent can decide when to act and to perceive at runtime. Furthermore, it is able to change perceptions filters based on its needs, and to decide what device it will be connected to at a certain time during its execution.

For future work, we intend to extend the ARGO architecture for programming multi-robot systems through a communication protocol between robotic agents. Moreover, it is necessary to test ARGO in different domains and apply robotics technics such as SLAM. We will also intend to provide other hardware-side libraries, for instance for PIC and Intel families.

Acknowledgments

Márcio F. Stabile Jr. is financed by CNPq. Carlos Pantoja is financed by CAPES. Jaime Simão Sichman is partially financed by CNPq, proc.303950/2013-7.

References

1. Barros, R.S., Heringer, V.H., Lazarin, N.M., Pantoja, C.E., Moraes, L.M.: An agent-oriented ground vehicles automation using Jason framework. In: 6^{th} International Conference on Agents and Artificial Intelligence. pp. 261–266 (2014)
2. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons Ltd (2007)

3. Bratman, M.E.: Intention, Plans and Practical Reasoning. Cambridge Press (1987)
4. Calce, A., Forooshani, P.M., Speers, A., Watters, K., Young, T., Jenkin, M.R.: Autonomous aquatic agents. In: ICAART (1). pp. 372–375 (2013)
5. Chen, S.Y.: Kalman filter for robot vision: A survey. IEEE Transactions on Industrial Electronics 59(11), 4409–4420 (2012)
6. Clark, K., Robinson, P.: Robotic agent programming in TeleoR. In: Robotics and Automation, 2015 IEEE International Conference on. pp. 5040–5047 (2015)
7. Dastani, M., de Boer, F., Dignum, F., Van Der Hoek, W., Kroese, M., Meyer, J.J., et al.: Programming the deliberation cycle of cognitive robots. In: Proc. of the 3rd International Cognitive Robotics Workshop (2002)
8. Guinelli, J.V., Junger, D., Pantoja, C.E.: An Analysis of Javino Middleware for Robotic Platforms Using Jason and JADE Frameworks. In: 10^{th} Software Agents, Environments and Applications School (2016)
9. Hama, M.T.: Uma plataforma orientada a agentes para o desenvolvimento de software em veículos aéreos não-tripulados. Master's thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil (2012)
10. Hindriks, K.V.: Programming rational agents in GOAL. In: Seghrouchni, A., Dix, J., Dastani, M., Bordini, H.R. (eds.) Multi-Agent Programming: Languages, Tools and Applications, pp. 119–157. Springer US, Boston, MA (2009)
11. Hindriks, K.V., De Boer, F.S., Van der Hoek, W., Meyer, J.J.C.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems 2(4), 357–401 (1999)
12. Jain, R.: Art of Computer Systems Performance Analysis: Techniques For Experimental Design Measurements Simulation and Modeling. Wiley (2015)
13. Kuiper, D.M., Wenkstern, R.Z.: Agent vision in multi-agent based simulation systems. Autonomous Agents and Multi-Agent Systems 29(2), 161–191 (2015)
14. Lazarin, N.M., Pantoja, C.E.: A robotic-agent platform for embedding software agents using raspberry pi and arduino boards. In: 9^{th} Software Agents, Environments and Applications School (2015)
15. Morais, M., Meneguzzi, F., Bordini, R., Amory, A.: Distributed fault diagnosis for multiple mobile robots using an agent programming language. In: Advanced Robotics (ICAR), 2015 International Conference on. pp. 395–400 (2015)
16. Mordenti, A., Ricci, A., Santi, D.I.A.: Programming robots with an agent-oriented bdi-based control architecture: Explorations using the jaca and webots platforms. Bologna, Italy, Tech. Rep (2012)
17. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: de Velde, W.V., Perram, J.W. (eds.) Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world (MAAMAW'96). Lecture Notes in Artificial Intelligence, vol. 1038, pp. 42–55. Springer-Verlag, USA (1996)
18. Ricci, A., Piunti, M., Viroli, M., Omicini, A.: Environment programming in CArtAgO. In: Seghrouchni, A., Dix, J., Dastani, M., Bordini, H.R. (eds.) Multi-Agent Programming: Languages, Tools and Applications, pp. 259–288. Springer US, Boston, MA (2009)
19. Santos, F.R., Hübner, J.F., Becker, L.B.: Concepção e análise de um modelo de agente BDI voltado para o planejamento de rota em um VANT. In: 9^{th} Software Agents, Environments and Applications School (2015)
20. Stabile Jr., M.F., Sichman, J.S.: Evaluating perception filters in BDI Jason agents. In: 4^{th} Brazilian Conference on Intelligent Systems (BRACIS) (2015)
21. Wei, C., Hindriks, K.V.: An agent-based cognitive robot architecture. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) Programming Multi-Agent Systems: 10th International Workshop, ProMAS, Valencia, Spain, pp. 54–71. Springer, Berlin (2013)
22. Wooldridge, M.J.: Reasoning about rational agents. MIT press (2000)