Dealing with the Unpredictability of Physical Resources in Real-world Multi-Agent Systems

Nilson Mori Lazarin[1,2][0000−0002−4240−3997],
Carlos Eduardo Pantoja[1,2][0000−0002−7099−4974], and
José Viterbo[1][0000−0002−0339−6624]

[1] Institute of Computing – Fluminense Federal University (UFF),
Niterói - RJ, Brazil
[2] Federal Center for Technological Education Celso Suckow da Fonseca (Cefet/RJ),
Rio de Janeiro - RJ, Brazil
{nilson.lazarin,carlos.pantoja}@cefet-rj.br, viterbo@ic.uff.br

Abstract. Although Multi-Agent Systems (MAS) can handle various problems in heterogeneous environments, unknown problems may arise at runtime due to their inherent heterogeneity when dealing with cyber-physical systems. Therefore, designers of intelligent cyber-physical systems must adopt approaches that enable adaptability and fault tolerance at runtime. This work proposes techniques for dynamically adding, removing, and swapping resources (sensors or actuators) in Embedded MAS at runtime. These approaches utilize the BDI model and a customized agent architecture capable of perceiving the availability of microcontrollers integrated into the system through serial communication. A case study was conducted to analyze five scenarios that an embedded MAS can deal with, starting from where there are no existing physical resources and new resources must be added at runtime and a scenario where failures, replacement, and the need for upgrade scenarios occur. Our approach was evaluated and tested in three different BDI frameworks, demonstrating that swapping resources at runtime is a promissory feature to guarantee the adaptability of intelligent cyber-physical systems.

Keywords: multi-agent systems · embedded multi-agent systems · embedded systems.

1 Introduction

Distributed Artificial Intelligence is a field of Science dedicated to studying, constructing, and applying autonomous systems capable of achieving objectives or performing some set of tasks [28]. A Multi-Agent System (MAS) comprises software agents that can perceive or act autonomously in a real or a virtual environment where they are situated; besides, these agents are cognitive, autonomous, proactive, and have social ability since they can interact with other agents from the MAS to compete or collaborate toward their individual or system goals [29]. Agents can assume cognitive abilities by adopting a cognitive

model. One of the most adopted cognitive models is the Belief-Desire-Intention model (BDI) [6]. This model is based on understanding the practical human reasoning that decides, moment by moment, what action to take to achieve goals based on plans that beliefs, desires, and intentions can activate [30].

These MAS have been applied in several domains due to their advantages that allow an increase in speed and efficiency due to their parallel nature and asynchronous operation, in addition to allowing scalability and flexibility [2]. These systems differ from conventional systems by presenting additional characteristics such as independence since its operation or existence does not depend on other agents; proactivity, as it acts on the environment, on its initiative; collaboration, as it communicates with other agents to organize their actions; cognition, as it draws up action plans to achieve a goal; and adaptability because in case of failure, it looks for executing alternative plans.

An Embedded MAS is a system running on top of devices, where cognitive agents are physically connected to resources to perceive and act in the real world and communicability with other devices [5]. By definition, these systems can deal with different problems in heterogeneous environments. However, unknown problems, and consequently, those not foreseen at design time, may be presented to the MAS at runtime, and there may be no agents in the system capable of dealing with the presented problem. Thus, due to the natural heterogeneity of real environments and the adaptability of agents, the designer must adopt an approach that makes the Embedded MAS adaptable and fault-tolerant at runtime, allowing the maintenance and replacement of resources that are damaged, or even the addition of new resources, to deal with new problems. One of them is to employ an Open MAS that allows mobility. Then, an agent with the necessary skills could be invited to join the society [8]. Besides, a MAS could also dispatch a duplicate capable of dealing with the problem [24].

The resources of an Embedded MAS are defined only at design time. The designer must define them before assembling the device, and, once defined, it is impossible to change them at runtime. For this, the Embedded MAS must be stopped, and the MAS reprogrammed. The swapping of resources — addition or removal — is an interesting feature in the development of Embedded MAS because it adds adaptability at runtime for agents. The system does not need to be turned off, and agents could reason about the availability of resources. In this way, an autonomous agent can be adaptable, continuing to perform actions to achieve its goals in case of hardware failure, for example. Considering the extant BDI agent-oriented languages and frameworks [3][4][9][22], they do not initially provide access to physical resources. Argo is a customized architecture that allows agents to interface with hardware resources but is also not prepared to deal with the swapping of resources [21].

This work introduces a novel functionality that enables the dynamic swapping of physical resources within an Embedded MAS. Consequently, an embedded system with existing physical resources can now have new resources attached or removed, with the agents automatically detecting these additions or absences. As the Embedded MAS uses the serial port to govern the microcontroller, the

agent becomes aware of its availability each time it attempts to access it during perception or action. The primary objective is to enhance the MAS's adaptive capacity and facilitate the development process of embedded systems. To achieve this, we extend Argo agents by incorporating a modified version of Javino [16], a serial interface responsible for message exchange between the microcontroller and agents. Javino can now identify added or removed resources, effectively notifying the agent about these changes.

Adding resources at runtime could be achieved by adopting an Open MAS and agent mobility allowing agents to enter and leave its system anytime [1]. In Jason, this is possible by adopting bio-inspired protocols for moving agents from one Embedded MAS to another [26]. Then, one resource could be added, and one agent with proper plans could be sent to control this resource. However, it is important to note that even with agent mobility, the agents cannot currently identify when a resource has been removed.

In this extended version of the paper [14], we also provide packages of Argo to work along with JaCaMo and Jason using command line interface (CLI). JaCaMo [3] integrates three MAS dimensions — organizational, environmental, and agency — combining Moise[12], CArtAgO[23], and Jason framework[4]. JaCaMo-CLI and Jason-CLI allows the creation of MAS using terminal commands without graphical interfaces. Besides, we assembled a new study case using a single-board computer hosting the Embedded MAS and some microcontrollers managing sensors and actuators considering Jason Embedded [19] using the ChonIDE [13], JaCaMo-CLI, and Jason-CLI. The Embedded MAS is developed using Jason interpreter, the extended Argo agents and packages, and Javino.

The contributions of this work are a novel feature to swap resources in Embedded MAS using BDI agents at runtime, an extended version of Argo agents and Javino for Jason framework, and packages for working with JaCaMo. This paper is structured as follows: Section 2 discusses some related work; In Section 3, we present the swap approach; The swap feature is tested in Section 4, and finally, we present the Conclusions and the References.

2 RELATED WORK

From a practical point of view, the swap of physical resources at runtime could facilitate the process of maintaining and expanding an Embedded MAS since it does need to be stopped to add a new resource or to remove an existing one. If the domain is critical, undesirable stops must be avoided at the most, and turning it off is not an option.

The Argo [21] architecture is a BDI agent capable of capturing and filtering the perceptions [27] coming from the sensors that sense the environment. It is also capable of sending commands to activate and deactivate actuators. Argo processes the perceptions directly as beliefs, and it can reduce the amount of perceptions by activating runtime filters, so the agent can focus only on those necessary to achieve its goals. Argo uses Javino [16] as the serial interface for

accessing the device's resources. Considering the various layers and steps of the development process of an embedded system, Argo facilitates MAS programming because it abstracts the technological issues of interfacing hardware. The agent just needs to know what serial port it is handling. Argo and Javino do help in the development of MAS, but they do not offer a mechanism to identify if the port the agent is handling is available or not. In fact, several solutions allow to define and employ the devices' resources at design-time [18][25][11]. In none of these solutions, the designer adds or removes the resources without stopping the system.

The Resource Management Architecture (RMA) [20] enables the addition of new devices at the edge of an IoT system at runtime. A device using the RMA can use an Embedded MAS to control microcontrollers, and all information gathered could be forwarded to be published using the Sensor as a Service model. In addition, Physical Artifacts using CArtAgO [23] can be used as a resource with or without a dedicated MAS [17]. These devices can be added or removed from the RMA at any time. However, although the dynamism of this IoT architecture, devices can only be added to the network if it is online. Furthermore, swapping the devices' resources is only possible during design time, and it is still impossible to add or remove any resource without stopping the MAS. Besides, it depends on an available IoT network for communicating.

The bio-inspired protocols [26] for moving agents allow an Embedded MAS of a device to take control of another device by moving all its agents and their respective mental states. However, the target device must be identical to the source device for effective hardware control. So, it is still possible to add additional resources to the target device at runtime and move agents with proper plans to handle these new resources. As an Embedded MAS uses a physical architecture with boards running an OS with serial interfacing between agents and microcontrollers, it is possible to add resources at runtime. Then, once agents can communicate and move from one MAS to another using bio-inspired protocols, it is possible to program the agent at design time and move it at runtime, adopting a protocol that does not eliminate the target MAS. In this way, knowing the serial port where the new device is connected and sending the agent prepared to handle it, it is possible to add a resource accessible by BDI agents at runtime in an Embedded MAS. However, removing agents is not yet possible, and the solution depends on the available communication infrastructure.

In our approach, the serial interface informs the agent about the port availability it is trying to access. Then, whenever the agent has a new resource connected to the Embedded MAS, it perceives which port it is connected to. If the resource is removed, the next time the agent tries to gather the perceptions or act, it updates its mental state with the unavailability of the resource. In this new version of Argo, the agent receives this information each time (in the beginning) its reasoning cycle is performed. It is also updated at the end of the cycle if it tries to perform an action using any resource. With this perception, the agent can deliberate whether or not to pursue an intention that might be unreachable.

3 METHODOLOGY

When acting in a dynamic physical environment, agents must be prepared to reason regarding the availability of information and resources. Agents can use their own physical resources to gather information and act upon this environment. Still, as with any physical component, these resources could be damaged, unavailable, or changed by improved technologies. Then, agents must follow the adaptive ability to be aware of which resources are available when it needs to use them. Besides, embedded agents must also be fault tolerant and decide what to do when a resource is not available or damaged. So, swapping devices at runtime is a desired feature for any Embedded MAS. In this section, we review the architecture for constructing a cognitive device using Embedded MAS and the new feature for swapping physical resources using the Jason framework and Argo agents.

It is necessary to observe a four-fold architecture to construct a device managed by an Embedded MAS:

1. **Hardware**. It comprises all available resources of a device. They are physically connected to a microcontroller. These sensors and actuators are responsible for gathering the environment's perceptions and acting upon them. All microcontrollers employed in the device must also be connected in serial ports of a single-board computer (or any micro-processed platform).
2. **Firmware**. It represents the microcontroller programming where the perceptions are mounted and sent to the Embedded MAS based on the agent-programming language or framework adopted. The commands that activate the actuators are also programmed in response to serial messages.
3. **Serial Communication**. All messages exchanged between agents and resources use serial communication. This layer uses a serial interface to manage the message flow between agents and different microcontrollers. Agents need to know which serial port the resources are connected to.
4. **Reasoning**. It includes the Embedded MAS programming running on the single-board computer. Agents are programmed to automatically understand the perceptions of sensors as beliefs; afterward, they can deliberate and send commands back to activate actuators.

This architecture makes it possible to exchange resources at runtime on an already-designed device since all layers are low coupled. New sensors or actuators can be added to the system anytime since they are connected to a microcontroller. After this, they can be connected to a serial port. So, for any agent to interface these new resources, it would only need to know which port to access at runtime. However, it could not know how to manipulate it and would need to learn these skills some other way.

In this paper, we present an approach that allows Argo agents to test the availability of serial ports. Then it can deliberate whether or not to continue pursuing the goals associated with an unavailable resource. Besides, when it becomes available again or a new resource is inserted at runtime, the agent is

aware of the availability of the serial port. We define the swapping of resources as the ability to add, remove, or exchange physical components to the device at runtime. This novel ability of BDI agents guarantees that agents could be adaptive and fault-tolerant regarding hardware resources. The Embedded MAS — and, consequently, the device — does not need to be turned off for predictive, preventive, or corrective maintenance. This characteristic could reduce risks and increase profits in some domains, such as industrial applications.

Any Argo agent interfaces the hardware resources using Javino by accessing which port the microcontroller is connected to the single-board computer. So, when connecting a new microcontroller with new resources in a device managed by an Embedded MAS (or when removing), Javino verifies if the port is accessible or not and informs to the Argo agent who is trying to access it by sending a belief with the port information and if it is *on*, *off*, or *timeout*. Otherwise, when the resource is removed or fails, it can deliberate to drop its intentions related to the disconnected resources, for example. Figure 1 shows the four-fold architecture and the belief representing the port availability (i.e., port(name, status)).

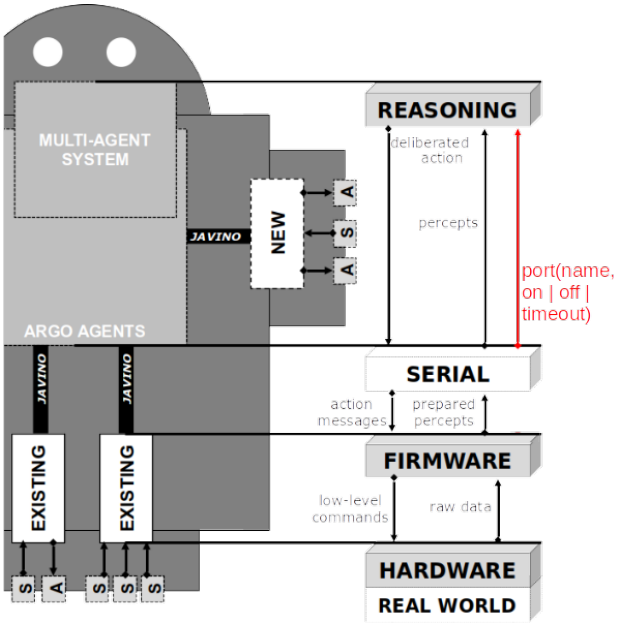

Fig. 1. The four-fold architecture for programming Embedded MAS on top of hardware devices [14].

When connecting a new resource in the system, three possible approaches can occur: an agent needs to learn how to deal with this new resource, a new agent can be employed to handle it, or any agent can already know how to use it. For the first case, the designer must program an external MAS and use an

IoT network [10] infrastructure to transfer the knowledge (plans) or the agent. At first, the designer can send the plans directly to a Communicator agent that redirects the plans to the Argo agent that controls the serial port. For this, Jason Embedded is employed. At last, a new Argo agent with the new desired abilities is transferred to the Embedded MAS using the bio-inspired Protocols [26]. Once the agent arrives at the destination, it can control the new resources and interact with the existing agents in that system. If no communicability or mobility is available, the designer can program the agent's behavior at design time to deal with possible new resources connected at runtime. However, adding totally unknown resources is still an open issue in the domain.

The practical intention is to create cognitive devices where agents are not dependent on resource availability. Agents can be stuck in pursuing goals that could be momentarily or permanently unreachable since the resources are not available anymore. In the worst case, the agent could deliberate based on wrong information, or the whole Embedded MAS could crash with malformed beliefs.

In order to ensure runtime adaptability in Embedded MAS, particularly for adding new resources or updating existing ones, it is essential to design the system with a communicator agent connected to an IoT server when using Jason Embedded. In this way, the system can receive new plans for an Argo agent that already manipulates a resource or can receive a new Argo agent capable of manipulating the resource to be added. When using the Argo packages for JaCaMo or Jason, some agent that knows how to deal with the new resource must exist at design time. Then, in Argo packages, the agent should exist when starting the system since it is impossible to move or create it at runtime. Alternatively, adding new agents at runtime direct from the terminal is possible using Jason-CLI. Figure 2 presents the proposed approach for building an Embedded MAS capable of adding, removing, or changing resources at runtime.

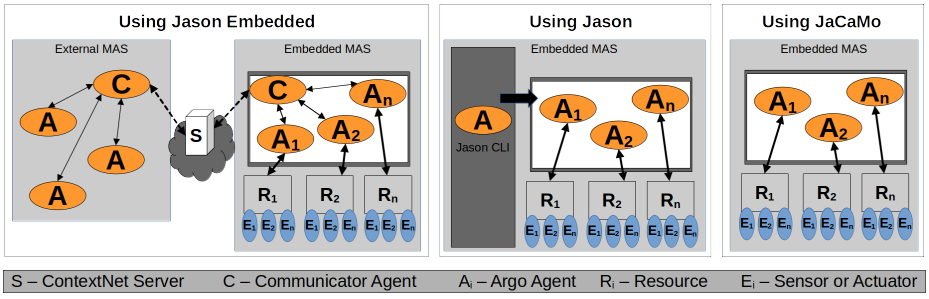

Fig. 2. The swapping methodology for Embedded MAS.

3.1 The Swap Feature in Argo Agents

Argo agents is a customized architecture from Jason's framework for interfacing hardware resources. All the information gathered from sensors is interpreted

as perceptions by Argo. Then, when programming Argo agents at design time, the designer needs to inform the serial port that the agent interfaces to the perceptions flow directly to the agent's belief base. It is important to remark that this process still occurs when resources fail or become unavailable. As said before, the agent is unaware of the port availability, which could lead to undesired behaviors.

Argo has the ability to change the serial port it is accessing and block the flow of perceptions at any time. If Argo is aware that a serial port is not answering anymore, it could try to reach another port or simply block the perceptions from that port. Then, when swapping resources, Argo agents need to access the status of the port which is trying to reach. For this, we defined a belief *port(Name, Status)*, where the name identifies the serial port name, and the status indicates if it is *on*, *off* or *timeout*. For example, when removing a resource located at serial port name *ttyACM0*, the agent receives directly in its belief base *port(ttyACM0, off)*.

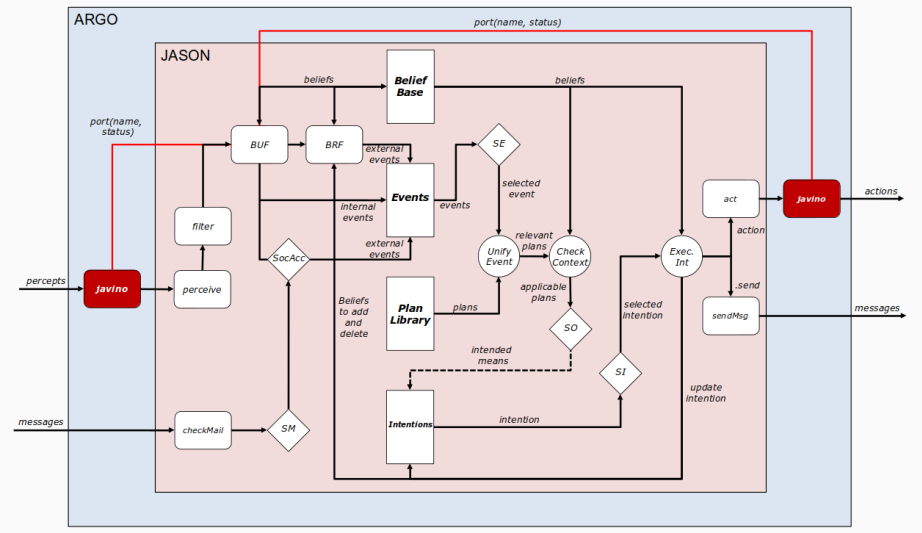

Fig. 3. The Argo's extended reasoning cycle [14].

Every BDI agent from Jason performs a well-defined reasoning cycle where the agent executes an expected behavior in each step. Argo has an extended reasoning cycle that modifies two distinct steps at the beginning of the cycle, when the agents perceive the real environment to gather perceptions, and at the end, when it acts, sending commands to actuators. The remaining steps are inspired by the Practical Reasoning System (PRS) [7]. It defines which events will trigger plans and intentions to define the sequence of actions to be performed.

In the perceive step of an Argo's reasoning cycle, the Javino is the serial interface responsible for gathering the perceptions from sensors and forwarding them to the Belief Update Function (BUF). Javino requests the perceptions by accessing the microcontroller whenever the agent performs a cycle. In this step, we modified Javino to inform whether or not the serial port the agent is trying to connect to is available.

In the same way, at the end of the cycle, the agent performs actions that can reflect in commands to be sent to actuators. In this step, Javino is also responsible for sending serial commands to the microcontroller. In this case, we modified the internal action named *act* to update the agent's belief base by adding the *port(Port, off)* belief in case the serial port is unavailable anymore. Javino tries to access the port, and in case of failure, it returns the aforementioned belief. The modified reasoning cycle of Argo agents is presented in Figure 3.

4 CASE STUDY

This case study involves the analysis of five possible scenarios at runtime considering an Embedded MAS that avoids obstacles. We considered the following scenarios (Figure 4):

- **Scenario 1:** The first scenario presents an Embedded MAS controlling a prototype with no sensor or actuator. In this scenario, there are no goals to achieve by agents, but the MAS must be ready if new resources are added.
- **Scenario 2:** In the second scenario, two new resources (a sensor and an actuator) will be added to the prototype. In this case, the Embedded MAS must identify the type of resource received and find an agent to control it.
- **Scenario 3:** In the third scenario, a case of fault tolerance is tested. The previously added sensor will stop reporting environmental perceptions to the Embedded MAS. In this case, the agent must stop acting to avoid collisions until the resource is functional again.
- **Scenario 4:** The faulty resource from the previous scenario will be replaced in the fourth scenario. However, the new resource will not maintain compatibility with the removed one. In this case, the Embedded MAS must be able to adapt, and agents still deliberate without any change.
- **Scenario 5:** Finally, the fifth scenario, the actuator will be replaced by another type of actuator compatible with the previous one. In this scenario, the Embedded MAS must recognize the resource exchange and fulfill its mission.

To fulfill the proposed scenarios, we implemented an Embedded MAS, which runs and controls the following physical devices: a single-board computer to host the reasoning layer and some microcontrollers to host the firmware layer. USB ports provide serial communication from the reasoning layer to the firmware layer. Finally, in the hardware layer, the actuators accountable for moving the robot were a biped platform and a wheeled robotic chassis; moreover, the sensors accountable for environmental perceptions used were an infrared sensor and an ultrasonic distance sensor.

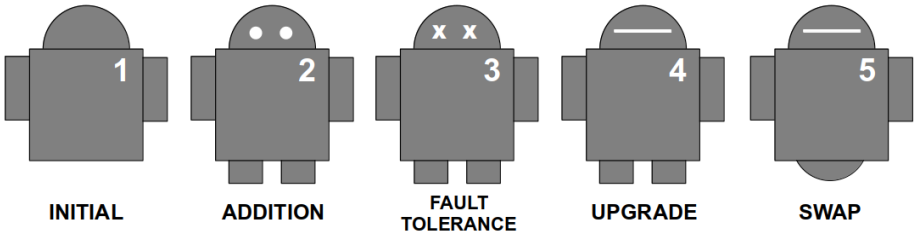

Fig. 4. The scenarios of proposed case study.

4.1 Reasoning Implementation

A MAS was idealized to deal with the unpredictability of physical resources and meet the proposed scenarios, basically containing a coordinator agent for the embedded system and operator agents for dealing with physical resources when they exist. The behavior of these agents is standard, regardless of the framework used, be it Jason Embedded, Jason-CLI, or JaCaMo. Thus, this subsection presents the plans of these agents before presenting each scenario.

Coordinator Agent The coordinator agent initially aims to test the system's serial ports. For this, three achievement goals were implemented, one for each serial port that triggers a plan to validate serial communication connectivity every five seconds and a contingency plan for when the ports are found, and he can achieve the objective of walking around.

The plan to validate connectivity tries to use the serial port and get the exogenous environment's perceptions. Two plans were implemented to meet these perceptions about the serial port's status. One to abort if the port is in an unknown state. The second is to trigger a plan for employing an operator agent when the serial port is online.

If the employing plan fails, a contingency plan is created to wait until an agent capable of handling the resource arrives at the MAS. In addition, a plan to deal with possible problems informed by the operator agents was implemented so that when a resource fails, the coordinator agent will order all agents in the system using a serial port to stop acting or perceiving. So it will again start the testing serial ports achievement plan.

Finally, when the serial ports are connected and controlled by an operator agent, the coordinator will try to achieve the objective of walking around. For this, three achievement plans were created. First, if the coordinator agent does not have perceptions about obstacles, it will order all agents to inform the exogenous perceptions. In the second, if the coordinator has received information that there is no obstacle, it will order the agents to act. Finally, if it has received information about an obstacle, it will order stopping acting. The coordinator agent code is presented in Code 1.1.

Code 1.1. coordinator.asl

```
/* Initial goals */
!testPorts.

/* Plans */
+!testPorts: not using(ttyUSB0) <- !search(ttyUSB0); .wait(5000); !testPorts.
+!testPorts: not using(ttyUSB1) <- !search(ttyUSB1); .wait(5000); !testPorts.
-!testPorts <- +ready; !walkAround.

+!search(Port) <- argo.port(Port); argo.limit(2500); argo.percepts(open).
+port(P,S): P = unknown <- argo.percepts(close); .abolish(_[source(percept)]).
+port(P,S): S=on & resource(R) <- argo.port(none); !newResource(P,R).

+!newResource(Port,RN)[source(self)]: operator(Ag,R) & R = RN <-
    .abolish(_[source(percept)]);
    +using(Port);
    .send(Ag,achieve,work(Port)).
-!newResource(Port,RN) <- .wait(operator(Ag,R)); !newResource(Port,RN).
+!offResource(N)[source(LayerAg)]: operator(A,R) & R = N & Ag = A <-
    .drop_desire(walkAround);
    .abolish(using(_));
    .broadcast(achieve,leavePort);
    !!testPorts.

+!walkAround: not obstacle(O) & using(ttyUSB0) & using(ttyUSB1)<-
    .broadcast(askOne,obstacle(R));
    .wait(obstacle(X));
    !walkAround.
+!walkAround: obstacle(O) & O = no & using(ttyUSB0) & using(ttyUSB1) <-
    .broadcast(achieve,task(front));
    .abolish(obstacle(_)[_]);
    !walkAround.
+!walkAround: obstacle(O) & O = yes & using(ttyUSB0) & using(ttyUSB1)<-
    .broadcast(achieve,task(stop));
    .abolish(obstacle(_)[_]);
    !walkAround.
```

Resource Operator Agent All resource operator agents initially aim to inform the coordinator agent of their abilities. An achievement goal was implemented to meet this objective that sends a message to the coordinator containing the agent's name and the resource's name.

In addition, these agents have a plan that the coordinator agent can trigger to start the serial port control activities. When connecting to the port, agents receive perceptions about serial port status. Two plans were programmed. The first one, if the serial port is on, the agent adds a mental notation that it is ready. Conversely, if the serial port is off, the agent removes the mental notation and sends a message informing the coordinator that the resource is unavailable.

Finally, four plans were implemented: the first was an achievement goal to disconnect from the serial port at the coordinator's request, and the last three were contingency plans. The operator agents' standard code is presented in Code 1.2.

Code 1.2. common/argoAgents.asl

```
/* Initial goals */
!infoCoordinator.

/* Plans */
+!infoCoordinator: myResource(R) <- .my_name(N); .send(coordinator,tell,operator(N,R)).

+!work(Port)[source(coordinator)] <- argo.port(Port); argo.percepts(open); !conf.
+port(Port,Status)[source(percept)]: Status = on & myResource(R) <- +ready.
+port(Port,Status)[source(percept)]: Status = off & myResource(R) <-
    -ready;
    argo.port(none);
    .abolish(_[source(percept)]);
    .send(coordinator,achieve,offResource(R)).

+!leavePort[source(coordinator)] <- !disconf; argo.percepts(close); argo.port(none).
-!task(T).
-!conf.
-!disconf.
```

Below are presented the specific behaviors of each resource operator agent used in the study case.

– **Operator agent 1**: Initially, the agent has a belief about the resource that it can operate. In addition, it has seven achievement goals: the first configures the resource perception cycle to occur every two seconds. The following five are responsible for managing the sending of actuation commands to the resource's microcontroller. The last stops the actuator before releasing the serial port communication. The first operator agent code is presented in Code 1.3.

Code 1.3. resourceOperator1.asl

```
/* Initial goals */
myResource(biped).

/* Plans */
+!conf <- argo.limit(2000).

+!task(T)[source(coordinator)]: not acting & ready & T=front & not status(walk) <-
      !platformOtto(walk).
+!task(T)[source(coordinator)]: not acting & ready & T=left & not status(turnL)<-
      !platformOtto(turnL).
+!task(T)[source(coordinator)]: not acting & ready & T=right & not status(turnR)<-
      !platformOtto(turnR).
+!task(T)[source(coordinator)]: not acting & ready & T=stop & not status(stop) <-
      !platformOtto(stop).
+!platformOtto(Op) <- +acting; argo.act(Op); .wait(3000); -acting.

+!disconf: status(S) & S \== stop <- !platformOtto(stop); !disconf.

{ include("common/argoAgents.asl") }
```

– **Operator agent 2**: Like the previous one, this agent has an initial belief about the resource it can operate. Moreover, an achievement goal is to carry out the configuration of perception. Finally, it has four plans that manipulate a mental notation regarding the existence or not of an obstacle ahead. The second operator agent code is presented in Code 1.4.

Code 1.4. resourceOperator2.asl

```
/* Initial goals */
myResource(obstacleIR).

/* Plans */
+!conf <- argo.limit(1000).

+left(L): right(R) & R=1 & L=1 <- -+obstacle(no).
+right(R): left(L) & R=1 & L=1 <- -+obstacle(no).

+right(R): left(L) & (R=0 | L=0) <- -+obstacle(yes).
+left(L): right(R) & (R=0 | L=0) <- -+obstacle(yes).

{ include("common/argoAgents.asl") }
```

– **Operator agent 3**: This agent has two initial beliefs, the first referring to the resource it is capable of operating and the second referring to the minimum limit in centimeters to consider that there is an obstacle ahead. Finally, it has two plans that manage a mental notation about the existence of an obstacle ahead. The third operator agent code is presented in Code 1.5.

Code 1.5. resourceOperator3.asl

```
/* Initial goals */
myResource(ultrasonicSensor).
minimalDistance(20).

/* Plans */
+!conf <- argo.limit(1000).

+distance(N): minimalDistance(D) & N <= D <- -+obstacle(yes).
+distance(N): minimalDistance(D) & N > D <- -+obstacle(no).

{ include("common/argoAgents.asl") }
```

– **Operator agent 4**: Like the others, the last operator agent has an initial belief about the resource it operates and a plan for configuring the perception cycle. The following four achievement plans manage the actuator. The final plan is responsible for stopping the action when the coordinator agent requests the release of the serial port. The code of the fourth resource operator agent is presented in Code 1.6.

Code 1.6. resourceOperator4.asl

```
/* Initial goals */
myResource(wd).

/* Plans */
+!conf <- argo.limit(1000).

+!task(T)[source(coordinator)]: ready & T=front & not status(running)<-
     argo.act(goAhead).
+!task(T)[source(coordinator)]: ready & T=left & not status(turnL)<- argo.act(goLeft).
+!task(T)[source(coordinator)]: ready & T=right & not status(turnR)<- argo.act(goRiht).
+!task(T)[source(coordinator)]: ready & T=stop & not status(stop)<- argo.act(stop).

+!disconf: status(S) & S \== stop <- argo.act(stop); .wait(2000); !disconf.

{ include("common/argoAgents.asl") }
```

4.2 Scenario 1: Initial Embedded MAS

In this scenario, the Embedded MAS has no sensor or actuator. Resources are not initially used for acting or perceiving the physical environment. The reasoning, in turn, runs on a Raspberry Pi Zero W, powered by a 5200 mAh (5V, 1.0A) portable battery. However, the system must be prepared to receive a new resource at anytime. Thus, an OTG cable and a USB hub are used in the serial communication layer to allow receiving new resources to the system at runtime. Figure 5 shows the necessary hardware to host the Embedded MAS. Additionally, we deploy an image of ChonOS [15] (*chonos-beta-RPI-Zero-W*) in a memory card, insert the card into the device, boot the system, connect it to the network, run the update of all system packages, and restart the device.

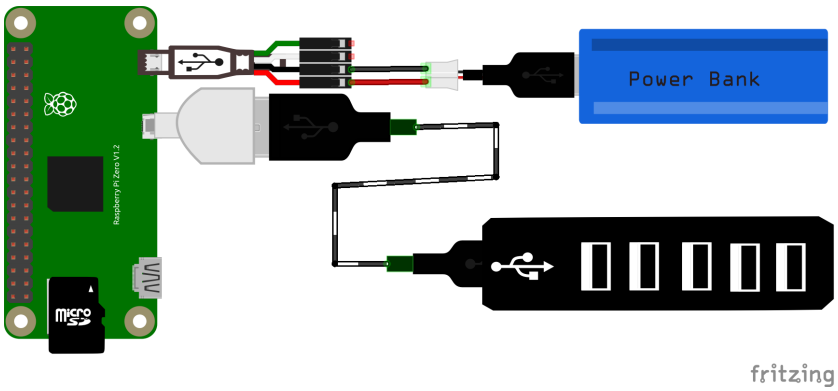

Fig. 5. Schematic of the necessary hardware to host the Embedded MAS.

- Reasoning Layer with *Jason Embedded* It was necessary to program an Embedded MAS with two initial agents, to fulfill the first scenario using the Jason Embedded. For this, we use the IDE that comes with the ChonOS installation (Figure 6). The first agent is the coordinator with an Argo architecture, presented in Code 1.1 from Section 4.1. The second agent is the telephonist with a Communicator architecture, responsible for managing MAS communications through an IoT gateway.

The telephonist agent (Code 1.7) has initial beliefs that represent the address of a gateway and its identification in the IoT network. Its initial goal is to stay connected to the network, and for that, an achievement goal was implemented to use *.connectCN* internal action. Two plans are implemented to allow communication management with other MAS: the first is to answer to the sender's attempt of connection, informing that the communication is ready; the second is to forward the received message to one internal agent in the MAS.

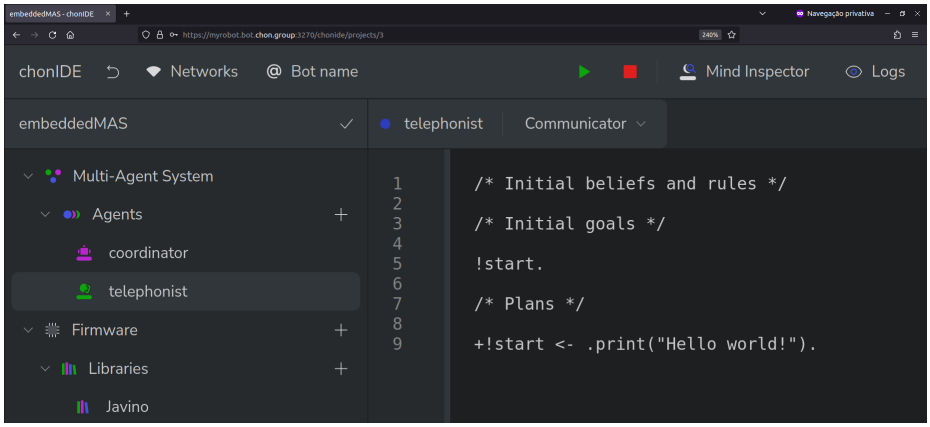

Fig. 6. Developing the Embedded MAS using ChonIDE[13].

Code 1.7. telephonist.asl

```
/* Initial beliefs and rules */
myID("10348514-519f-439b-afd7-7330027f4b70").
srv("skynet.chon.group",5500). //public IoT gateway address

/* Initial goals */
!connect.

/* Plans */
+!connect: myID(ID) & srv(S,P) <- .connectCN(S,P,ID).

+communication(trying)[source(X)] <- .sendOut(X,tell,communication(ok)).

+!retransmit(Dest,Force,Content)[source(X)] <- .send(Dest,Force,Content).
```

A MAS developed with Jason Embedded is an Open MAS. This way, new agents can be added at runtime. Adding operator agents in the initial scenario is unnecessary, and the MAS is easily adaptable to other scenarios.

- Reasoning Layer with *Jason-CLI* It was necessary to program an Embedded MAS with one initial agent, to fulfill the first scenario using the Jason. For it, we installed the ChonOS Jason-CLI package[3]; after that, an initial project was created, and finally, it was necessary to import the Argo package[4] into the project libraries directory. Figure 7 shows the Jason-CLI environment preparation process.

The initial project was changed to serve the scenario, according to Code 1.8. The initial MAS has the coordinator agent, presented in Code 1.1 from Section 4.1, which now uses the Argo architecture.

[3] https://github.com/chon-group/dpkg-jason
[4] https://github.com/chon-group/argo

Fig. 7. Developing the MAS using Jason[4] through CLI.

Code 1.8. embeddedMAS2.mas2j

```
MAS embeddedMAS2 {
    agents: coordinator agentArchClass jason.Argo;
    aslSourcePath: "src/agt";
}
```

- Reasoning Layer with *JaCaMo* It was necessary to program an Embedded MAS with all agents, to fulfill the first scenario using the JaCaMo. For it, we installed the ChonOS JaCaMo-CLI package[5]; after that, an initial project was created, and finally, it was necessary to import the Argo-jcm package[6] into project file. Figure 8 shows the jacamo-cli environment preparation process.

The initial project was changed to serve the scenario, according to Code 1.8. The initial MAS has the coordinator agent, presented in Code 1.1 and operator agents, presented in Codes 1.3-1.6 from Section 4.1, which now uses the Argo architecture.

Code 1.9. embeddedMAS3.jcm

```
mas embeddedMAS3 {
  agent resourceOperator1 { ag-arch: jason.Argo
  }
  agent resourceOperator2 { ag-arch: jason.Argo
  }
  agent resourceOperator3 { ag-arch: jason.Argo
  }
  agent resourceOperator4 { ag-arch: jason.Argo
  }
  agent coordinator { ag-arch: jason.Argo
  }
  uses package: argo "com.github.chon-group:argo-jcm:1.0.1"
}
```

[5] https://github.com/chon-group/dpkg-jacamo
[6] https://github.com/chon-group/argo-jcm

```
nilson@local2:~$ ssh root@myrobot.bot.chon.group
root@myrobot.bot.chon.group's password:
root@myrobot:~# apt install jacamo-cli
Reading package lists... Done
```
Installing jacamo-cli package

```
root@myrobot:~# jacamo app create embeddedMAS3 --console
Creating directory embeddedMAS3
You can run your application with:
   $ jacamo embeddedMAS3/embeddedMAS3.jcm
```
Creating project

```
root@myrobot:~# cd embeddedMAS3
root@myrobot:~/embeddedMAS3# sed '/^\}$/i uses package: argo "com.github.chon-group:argo-jcm:1.0"' embeddedMAS3.jcm > new.jcm
root@myrobot:~/embeddedMAS3# mv new.jcm embeddedMAS3.jcm
```
Importing Argo package

```
root@myrobot:~/embeddedMAS3# jacamo embeddedMAS3.jcm
  > Task :run
  Runtime Services (RTS) is running at 192.168.168.1:35295
  Agent mind inspector is running at http://192.168.168.1:3272
  CArtAgO Http Server running on http://192.168.168.1:3273
  Moise Http Server running on http://192.168.168.1:3274
  [Moise] OrgBoard o created.
  [Moise] ERROR creating group g1: group1 using artifact ora4mas.nopl.GroupBoard
  [Cartago] Workspace w created.
  [Cartago] artifact c1: example.Counter(3) at w created.
  [bob] join workspace /main/o: done
  [bob] join workspace /main/w: done
  [bob] focusing on artifact c1 (at workspace /main/w) using namespace default
  [bob] focus on c1: done
  [bob] focusing on artifact g1 (at workspace /main/o) using namespace default
  [bob] focus on g1: done
  [bob] focusing on artifact o (at workspace /main/o) using namespace default
  [bob] focus on o: done
  [bob] hello world.
^Croot@myrobot:~/embeddedMAS3#
logout
Connection to myrobot.bot.chon.group closed.
nilson@local2:~$
```
Executing hello world

Fig. 8. Developing the MAS using JaCaMo[3] through CLI.

4.3 Scenario 2 - Addition

All interventions performed in the four layers to meet the second scenario will be described in this subsection.

Hardware Layer For this scenario, two resources were assembled and connected to the USB hub of the prototype at runtime. The first, shown in Figure 9, is an actuator (biped platform) formed by four Micro Servo 9g SG90 TowerPro. The second, shown in Figure 10, is an obstacle sensor formed by two IR Sensor Modules.

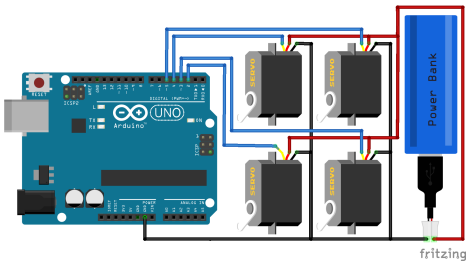

Fig. 9. Schematic of the resource 1.

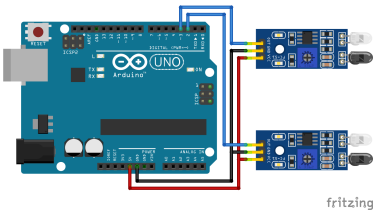

Fig. 10. Schematic of the resource 2.

Firmware and Serial Layer We used the Otto Library to program the microcontroller of the first resource. The microcontroller receives commands (walk, turnL, or turnR) to act in the environment. Besides, it sends two beliefs to the agent, one is the platform's status, and the other is the resource name. Code 1.10 presents the firmware programming of the resource.

Code 1.10. biped.ino

```
#include <Javino.h> //https://github.com/chon-group/javino2arduino
#include <Otto.h> //https://github.com/OttoDIY/OttoDIYLib
Javino javino;
Otto Otto;
String strStatus = "stop";

void serialEvent(){javino.readSerial();}

void setup(){javino.start(9600); Otto.init(4, 5, 2, 3, true, 0);}

void loop(){if(javino.availableMsg()){
  if(javino.getMsg()=="getPercepts"){javino.sendMsg("resource(biped);status("+
    strStatus+");");}
  else{strStatus = javino.getMsg();}}
  delay(100);
  action(strStatus);
}

void action(String strCMD){
  if(strCMD == "walk"){Otto.walk(1,1000,1);}
  else if (strCMD == "turnL"){Otto.turn(1,1000,1);}
  else if (strCMD == "turnR"){Otto.turn(1,1000,-1);}
}
```

The microcontroller of the second resource sends three beliefs to the agent, one is the resource name, and the next two are the status of the obstacle ahead. Code 1.11 presents the firmware programming of the second resource.

Code 1.11. obstacle.ino

```
#include <Javino.h> //https://github.com/chon-group/javino2arduino
Javino javino;

void serialEvent(){javino.readSerial();}

void setup(){javino.start(9600); pinMode(3, INPUT); pinMode(2, INPUT);}

void loop() {
 if(javino.availableMsg()){if(javino.getMsg() == "getPercepts"){
     javino.sendMsg("resource(obstacleIR);left(" +
     String(digitalRead(3))+");right(" + String(digitalRead(2))+");");}
 }
}
```

In both cases, the Javino library for Arduino provides the integration of the firmware layer with the reasoning layer.

Reasoning Layer This layer had different implementations for each of the frameworks used. The operator agent for resource 1 was described in Code 1.3 and the operator agent for resource 2 was described in Code 1.4, both in Section 4.1.

– ***Using Jason Embedded***: In this case, it was necessary to program a Communicator agent in an external MAS to send the operator. This agent has plans to test communication with the operator agent running on the Embedded MAS. After confirming the connection, the agent executes a plan responsible for sending the operator using the mutualism protocol. Finally, an execution request plan is triggered so that the operator agent, newly arrived at the MAS, informs the coordinator agent. Code 1.12 presents the sender agent plans.

Code 1.12. sender.asl

```
embMAS("10348514-519f-439b-afd7-7330027f4b70"). /* Initial belief*/
!start.           /* Initial goal*/
/* Plans */
+!start <-
    .connectCN("skynet.chon.group",5500,"19566fee-4bc6-45eb-8f72-455552d50116");
    +connected; !testComm.

+!testComm: communication(ok) <- !transmit.
+!testComm : connected & not communication(ok) & embMAS(E)<-
    .sendOut(E, tell, communication(trying)); .wait(3000); !testComm.

+!transmit <-
    !sendAgent(operator1); !requestExecution(operator1, infoCoordinator);
    !sendAgent(operator2); !requestExecution(operator2, infoCoordinator);
    .disconnectCN; .stopMAS.
+!sendAgent(Agent): embMAS(E) <- .moveOut(E,mutualism,Agent).
+!requestExecution(Receiver, Operation): embMAS(E) <-
    .sendOut(E, achieve, retransmit(Receiver,achieve,Operation)).
```

– ***Using Jason***: In this case, shell access to the operating system that hosts the Embedded MAS is required. So, the developer must input the agents into the running MAS using Jason-CLI. Code 1.13 presents the necessary terminal commands.

Code 1.13. jason-cli terminal commands

```
root@myrobot:~# cd embeddedMAS2/
root@myrobot:~/embeddedMAS2# jason agent start operator1 --ag-arch=jason.Argo
root@myrobot:~/embeddedMAS2# jason agent load-into --source=newAgents/op1.asl operator1
root@myrobot:~/embeddedMAS2# jason agent start operator2 --ag-arch=jason.Argo
root@myrobot:~/embeddedMAS2# jason agent load-into --source=newAgents/op2.asl operator2
```

4.4 Scenario 3 - Fault tolerance

This scenario is accomplished by the port plan (*+port(Port,Status)*) common to all Argo agents, described in Code 1.2 (in Section 4.1). If some communication problem arises with the reasoning layer (*Status=off* context), the agent sends a message to the coordinating agent(*.send(coordinator,achieve,offResource(R)*).

The coordinator agent, in turn, gives up on reaching the goal of walking around (*.drop_desire(walkAround)*), asks all agents to release the serial ports (*.broadcast(achieve,leavePort)*), clears its beliefs about the use of ports (*.abolish(using(_)*) and start testing the serial ports again, and then employ the operator agents. The Code1.1, described in Section 4.1, presents the coordinating agent's plans to deal with this scenario of failure in communication with the resources.

4.5 Scenario 4 - Upgrade

Interventions performed in the four layers to meet scenario 4 are described in this subsection.

Hardware Layer A distance sensor was made to meet this scenario. It consists of an ultrasonic sensor HC-SR04. Besides, resource 2 was removed from the USB port, and resource 3 has been plugged into its place. The resource schematic is shown in Figure 11.

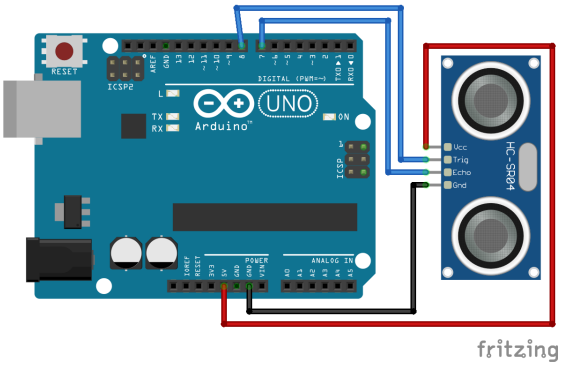

Fig. 11. Schematic of the resource 3.

Firmware and Serial Layer To program the firmware for this resource, we used the HCSR04 library. The microcontroller informs the operator agent of the distance in centimeters and the resource's name. To allow communication between the layers, the Javino library for Arduino was used. The resource schedule is presented in the Code 1.14.

Code 1.14. ultrasonic.ino

```
#include <Javino.h> //Available at: https://github.com/chon-group/javino2arduino
#include <HCSR04.h> //Available at: https://www.arduinolibraries.info/libraries/hcsr04
Javino javino;
HCSR04 hc(8, 7);

void serialEvent(){javino.readSerial();}

void setup() {javino.start(9600); pinMode(7, INPUT); pinMode(8, OUTPUT);}

void loop(){if(javino.availableMsg()){
    if(javino.getMsg()=="getPercepts"){javino.sendMsg("distance("
    +String(hc.dist())+");resource(ultrasonicSensor);");}}
}
```

Reasoning Layer In the reasoning layer, using Jason Embedded, it was necessary to implement a MAS to send the coordinator agent in the same way as performed in scenario number 2 (Code1.12). The only difference in this scenario is that only one agent is sent. Operator agent 3, described in Code1.5 (section 4.1). Similarly, when using Jason, the developer must have access to the operating system hosting the MAS to run the terminal commands, add an empty agent to the SMA, and later load the plans described above.

4.6 Scenario 5 - SWAP

Interventions performed in the four layers to meet scenario 5 are described in this subsection.

Hardware Layer In this scenario, a 2WD robotic platform was used. Resource 1 has been removed from the USB port, and this resource has been plugged into its place. The schematic of resource 4 is shown in Figure 12.

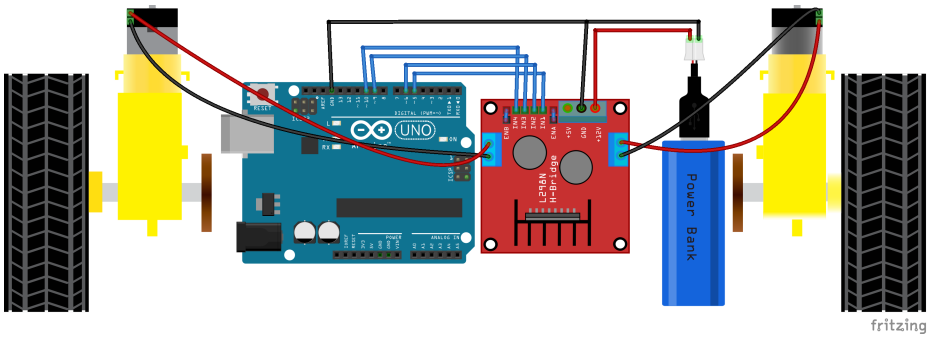

Fig. 12. Schematic of the resource 4.

Firmware and Serial Layer The microcontroller of resource 4 receives the commands (goAhead, goLeft, goRight, or stop) sent by the operator agent to act in the environment. In addition, the microcontroller reports the actuators' status to the agent. The Javino library for Arduino provides communication between the agent and the resource. The programming is presented in Code 1.15.

Reasoning Layer In the reasoning layer, using Jason Embedded, it was necessary to implement a MAS to send the coordinator agent in the same way as performed in scenarios 2 and 3 (Code1.12). This operator agent is described in Code1.6, in Section 4.1. Similarly, when using Jason, the developer must have access to the operating system hosting the MAS, to run the terminal commands to add an empty agent to the SMA and later load the plans described above.

Code 1.15. 2WD.ino

```
#include <Javino.h> //Available at: https://github.com/chon-group/javino2arduino
Javino javino;
String strMotorStatus;

void serialEvent(){ javino.readSerial(); }

void setup(){javino.start(9600); pinMode(5, OUTPUT); pinMode(6, OUTPUT);
             pinMode(9, OUTPUT); pinMode(10, OUTPUT);}

void loop(){
if(javino.availableMsg()){
  String strMsg = javino.getMsg();
  if(strMsg=="getPercepts")javino.sendMsg("resource(wd);motor("+strMotorStatus+");");
    else if(strMsg=="stop")stopRightNow();
    else if(strMsg=="goLeft")turnLeft();
    else if(strMsg=="goRight")turnRight();
    else if(strMsg=="goAhead")goAhead();
  }
}

void stopRightNow(){digitalWrite(5, LOW); digitalWrite(6, LOW); digitalWrite(9, LOW);
                    digitalWrite(10, LOW); strMotorStatus="stopped";}

void goAhead(){stopRightNow(); digitalWrite(6, HIGH); digitalWrite(10, HIGH);
    strMotorStatus="running";}

void turnRight(){stopRightNow(); digitalWrite(5, HIGH); digitalWrite(10, HIGH);
    strMotorStatus="turningRight";}

void turnLeft(){stopRightNow(); digitalWrite(6, HIGH); digitalWrite(9, HIGH);
  strMotorStatus="turningLeft";}
```

5 CONCLUSIONS

This work presented three approaches to adding, removing, and swapping resources (sensors or actuators) in Embedded MAS at runtime using a customized agent architecture capable of perceiving the availability of microcontrollers integrated into the system via serial communication. The first approach uses an IoT infrastructure to receive or send agents able to operate resources unknown by a society of agents. In this approach, we used Jason Embedded and ChonIDE.

Considering the access to the CLI of the operating system that hosts the Embedded MAS, the second approach uses the developer's ability to create and include new agents capable of operating a new resource in the runtime. In this approach, we used Jason-CLI.

In the third approach, we developed an Embedded MAS using JaCaMo-CLI. However, in this approach, the creation or reception of new agents at runtime is still open; therefore, all possible resources to be added at runtime must exist at design time.

A case study with five possible runtime scenarios with an embedded MAS was conducted to analyze the proposed approaches, demonstrating that swapping resources at runtime is a promissory feature to guarantee the adaptability of intelligent cyber-physical systems.

Adding resources allows an Embedded MAS to be updated and improved at runtime without stopping it. Stopping a MAS can lead to some undesired situ-

ations, for example, in a mission-critical domain, which could generate failures because of the absence of information. Besides, when adding a new resource, it would be necessary to modify the physical structure of the device, offering some continuity and availability risks of the service that the device is running. Until recently, any resource addition forces the device to be turned off, limiting the adaptability inherent to a Cognitive MAS.

This discussion can also be expanded toward replacing and removing resources at runtime. In embedded systems, it is not uncommon for components to be damaged when interacting with the real world, given their unpredictability. In the presented approaches, the replacement could be performed without risks to the Embedded MAS if the damaged resource is replaced by another one of the same logical structure connected to the same serial port. Removing a resource (whether damaged or intentionally removed) leads to readapting the Embedded MAS to avoid pursuing intentions and objectives that can no longer be achieved due to the absence of interfacing. In this case, mechanisms for removing intentions, objectives, or plans are necessary.

Swapping resources at runtime still requires a multidisciplinary effort from the designer team since it has to know several areas (electronics, operating systems, object-oriented and agent-oriented programming). Besides, adding an totally unknown resource is still an open issue in the domain. In future works, a mechanism is needed for the dynamic management of resources in Embedded MAS so that, when adding a new resource, the MAS would automatically recognize the device and its functionalities without transferring agents from other systems or searching the necessary plans to operate the resource.

References

1. Artikis, A., Pitt, J.: Specifying open agent systems: A survey. In: International Workshop on Engineering Societies in the Agents World. pp. 29–45 (2008)
2. Balaji, P.G., Srinivasan, D.: An Introduction to Multi-Agent Systems, pp. 1–27. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14435-6_1
3. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with jacamo. Science of Computer Programming **78**(6), 747–761 (2013). https://doi.org/10.1016/j.scico.2011.10.004, special section: The Programming Languages track at the 26th ACM Symposium on Applied Computing (SAC 2011) Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments
4. Bordini, R., Hübner, J., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. Wiley Series in Agent Technology, Wiley (2007)
5. Brandão, F.C., Lima, M.A.T., Pantoja, C.E., Zahn, J., Viterbo, J.: Engineering approaches for programming agent-based iot objects using the resource management architecture. Sensors **21**(23) (2021). https://doi.org/10.3390/s21238110
6. Bratman, M.E.: Intention, Plans and Practical Reasoning. Cambridge Press (1987)
7. Bratman, M.E., Israel, D.J., Pollack, M.E.: Plans and resource-bounded practical reasoning. Computational intelligence **4**(3), 349–355 (1988)

8. Demazeau, Y., Costa, A.R.: Populations and organizations in open multi-agent systems. In: Proceedings of the 1st National Symposium on Parallel and Distributed AI. pp. 1–13 (1996)
9. Dennis, L.A., Farwer, B.: Gwendolen: A BDI language for verifiable agents. In: Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning, Society for the Study of Artificial Intelligence and Simulation of Behaviour. pp. 16–23 (2008)
10. Endler, M., Baptista, G., Silva, L.D., Vasconcelos, R., Malcher, M., Pantoja, V., Pinheiro, V., Viterbo, J.: Contextnet: Context reasoning and sharing middleware for large-scale pervasive collaboration and social networking. In: Proceedings of the Workshop on Posters and Demos Track. PDT '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2088960.2088962
11. Hamdani, M., Sahli, N., Jabeur, N., Khezami, N.: Agent-Based Approach for Connected Vehicles and Smart Road Signs Collaboration. Computing and Informatics **41**(1), 376–396 (4 2022). https://doi.org/10.31577/cai_2022_1_376
12. Hubner, J.F., Sichman, J.S., Boissier, O.: Developing organised multiagent systems using the moise+ model: programming issues at the system and agent levels. International Journal of Agent-Oriented Software Engineering **1**(3-4), 370–395 (2007). https://doi.org/10.1504/IJAOSE.2007.016266
13. Souza de Jesus, V., Mori Lazarin, N., Pantoja, C.E., Vaz Alves, G., Ramos Alves de Lima, G., Viterbo, J.: An ide to support the development of embedded multi-agent systems. In: Mathieu, P., Dignum, F., Novais, P., De la Prieta, F. (eds.) Advances in Practical Applications of Agents, Multi-Agent Systems, and Cognitive Mimetics. The PAAMS Collection. pp. 346–358. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-37616-0_29
14. Lazarin., N., Pantoja., C., Viterbo., J.: Swapping physical resources at runtime in embedded multiagent systems. In: Proceedings of the 15th International Conference on Agents and Artificial Intelligence - Volume 1: ICAART. pp. 93–104. INSTICC, SciTePress (2023). https://doi.org/10.5220/0011750700003393
15. Lazarin, N., Pantoja, C., Viterbo, J.: Towards a Toolkit for Teaching AI Supported by Robotic-agents: Proposal and First Impressions. In: Anais do XXXI Workshop sobre Educação em Computação. pp. 20–29. SBC, Porto Alegre, RS, Brasil (2023). https://doi.org/10.5753/wei.2023.229753
16. Lazarin, N.M., Pantoja, C.E.: A robotic-agent platform for embedding software agents using raspberry pi and arduino boards. In: 9th Software Agents, Environments and Applications School (2015)
17. Manoel, F., Pantoja, C.E., Samyn, L., de Jesus, V.S.: Physical Artifacts for Agents in a Cyber-Physical System: A Case Study in Oil & Gas Scenario (EEAS). In: SEKE. pp. 55–60 (2020)
18. Michaloski, J., Schlenoff, C., Cardoso, R., Fisher, M., others: Agile Robotic Planning with Gwendolen (2022)
19. Pantoja, C.E., de Jesus, V.S., Lazarin, N.M., Viterbo, J.: A Spin-off Version of Jason for IoT and Embedded Multiagent Systems. In: Intelligent Systems. Springer International Publishing, Cham ((in preparation) 2023)
20. Pantoja, C.E., Soares, H.D., Viterbo, J., Alexandre, T., Seghrouchni, A.E.F., Casals, A.: Exposing iot objects in the internet using the resource management architecture. International Journal of Software Engineering and Knowledge Engineering **29**(11n12), 1703–1725 (2019). https://doi.org/10.1142/S0218194019400175
21. Pantoja, C.E., Stabile, M.F., Lazarin, N.M., Sichman, J.S.: Argo: An extended jason architecture that facilitates embedded robotic agents programming. In: Bal-

doni, M., Müller, J.P., Nunes, I., Zalila-Wenkstern, R. (eds.) Engineering Multi-Agent Systems. pp. 136–155. Springer International Publishing, Cham (2016)
22. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In: Multi-agent programming, pp. 149–174. Springer (2005)
23. Ricci, A., Piunti, M., Viroli, M., Omicini, A.: Environment Programming in CArtAgO, pp. 259–288. Springer US, Boston, MA (2009). https://doi.org/10.1007/978-0-387-89299-3_8
24. Shehory, O., Sycara, K., Chalasani, P., Jha, S.: Agent cloning: an approach to agent mobility and resource allocation. IEEE Communications Magazine **36**(7), 58–67 (1998). https://doi.org/10.1109/35.689632
25. Silva, G.R., Becker, L.B., Hübner, J.F.: Embedded architecture composed of cognitive agents and ros for programming intelligent robots. IFAC-PapersOnLine **53**(2), 10000–10005 (2020). https://doi.org/10.1016/j.ifacol.2020.12.2718, 21st IFAC World Congress
26. Souza de Jesus., V., Pantoja., C., Manoel., F., Alves., G., Viterbo., J., Bezerra., E.: Bio-inspired protocols for embodied multi-agent systems. In: Proceedings of the 13th International Conference on Agents and Artificial Intelligence - Volume 1: ICAART,. pp. 312–320. INSTICC, SciTePress (2021). https://doi.org/10.5220/0010257803120320
27. Stabile Jr., M.F., Pantoja, C.E., Sichman, J.S.: Experimental Analysis of the Effect of Filtering Perceptions in BDI Agents. International Journal of Agent-Oriented Software Engineering **6**(3-4), 329–368 (2018)
28. Weiss, G.: Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. MIT Press, Cambridge, MA, USA, 1st edn. (2000)
29. Wooldridge, M.: An Introduction to MultiAgent Systems. Wiley (2009)
30. Wooldridge, M.: Intelligent Agents. In: Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. MIT Press, Cambridge, MA, USA, 1st edn. (2000)