

Swapping Physical Resources at Runtime in Embedded MultiAgent Systems

Nilson Mori Lazzarin^{1,2}^a, Carlos Eduardo Pantoja^{1,2}^b and José Viterbo¹^c

¹*Institute of Computing (IC), Fluminense Federal University (UFF), Niterói-RJ, Brazil*

²*Federal Center for Technological Education Celso Suckow da Fonseca (Cefet/RJ), Rio de Janeiro, Brazil*

Keywords: Multi-Agent Systems, Embedded Multi-Agent Systems, Embedded Systems.

Abstract: An Embedded MultiAgent System (MAS) is a cognitive system embedded into a physical device responsible for controlling the existing resources and communicability with other devices. An Embedded MAS provides autonomy and proactivity to physical devices using the BDI model. Designing a device implies choosing sensors and actuators as resources and programming firmware and reasoning at design time. However, at runtime, resources could sometimes be damaged, presenting malfunctioning, or need to be changed. Then, performing predictive, preventive, or corrective maintenance at runtime is impossible since the designer must stop the Embedded MAS to swap resources and reprogram the system. This paper presents a novel ability for swapping resources at runtime in Embedded MAS using an extended version of Argo agents and the Jason framework. A case study analyses the new swap ability in different situations: removing and changing existing resources, adding new known and unknown resources, and causing a failure in a resource. The study case shows how the new swap ability can make devices with Embedded MAS adaptable and fault-tolerant.

1 INTRODUCTION

A MultiAgent System (MAS) is composed of software agents that can perceive or act in a real or a virtual environment where they are situated. These agents are cognitive, autonomous, proactive, and have the social ability since they can interact with other agents from the MAS to compete or collaborate toward their individual or system goals (Wooldridge, 2009). Agents can assume cognitive abilities by adopting a cognitive model. One of the most adopted cognitive models is the Belief-Desire-Intention model (BDI) (Bratman, 1987). 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.

Agent-based systems provide autonomy, pro-activity, and social ability to physical devices (Matarić, 2007). An Embedded MAS is a system running on top of devices, where cognitive agents are physically connected to resources to per-

ceive and act in the real world and communicability with other devices (Brandão et al., 2021). Commonly, these devices use a four-fold architecture: the hardware layer is composed of the set of resources (sensors and actuators) that represents the agent's capabilities in the real world; the firmware layer is responsible for the functions that the devices perform according to the agent's deliberations. In this layer, the designer programs the resources connected to one or more microcontrollers; the interface layer allows the agent to communicate with the microcontroller using serial communication. The Embedded MAS must run in a single-board computer — or any platform that hosts an Operating System (OS) — where microcontrollers are connected. Then, in the reasoning layer, the agent can deliberate based on perceptions gathered from sensors and act by sending serial commands (Pantoja et al., 2016).

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 re-programmed. The swapping of resources — addition or removal — is an interesting feature in the development of Embedded MAS because it adds adaptability

^a <https://orcid.org/0000-0002-4240-3997>

^b <https://orcid.org/0000-0002-7099-4974>

^c <https://orcid.org/0000-0002-0339-6624>

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 (Bordini et al., 2007)(Pokahr et al., 2005)(Dennis and Farwer, 2008), they do not initially provide access to physical resources. Argo is a customized architecture that allows agents to interface with hardware resources, but it is not prepared to deal with swapping resources (Pantoja et al., 2016).

Adding resources at runtime could be achieved by adopting an Open MAS and agent mobility. An open MAS allows agents to enter and leave its system anytime (Artikis and Pitt, 2008). Another Jason extension uses bio-inspired protocols for moving agents from one Embedded MAS to another (Souza de Jesus. et al., 2021). Then, one resource could be added, and one agent with proper plans could be sent to control this resource. But, even with mobility, the agents cannot identify the resource removal.

This work presents a new feature for swapping physical resources at runtime in Embedded MAS. Therefore, one embedded system that already has physical resources available may have new resources attached to it or removed, and agents will automatically be aware of these new resources or their absence. As the Embedded MAS uses the serial port to connect to microcontrollers, the agent is aware if the port is available or not every time it tries to reach it by perceiving the real world or acting. It aims to improve the MAS's adaptive capacity and facilitate the embedded system's development process. For this, we extend Argo agents to identify which resource has been added and removed using a modified version of Javino (Lazarin and Pantoja, 2015), the serial interface responsible for the message exchange between the microcontroller and agents. Javino identifies if the required resource is connected to the device and informs the agent.

We assembled one device to test these new features. A Single Board Computer hosts the Embedded MAS and some microcontrollers managing sensors and actuators. The Embedded MAS is developed using Jason, the extended Argo agents, and Javino. The contributions of this work are a novel feature to swap resources in Embedded MAS using BDI agents at runtime and an extended version of Argo agents and Javino for Jason framework. 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 not 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 (Pantoja et al., 2016) architecture is a BDI agent capable of capturing and filtering the perceptions (Stabile Jr. et al., 2018) 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 (Lazarin and Pantoja, 2015) 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 (Michaloski et al., 2022)(Silva et al., 2020)(Hamdani et al., 2022). In none of these solutions, the designer adds or removes the resources without stopping the system.

The Resource Management Architecture (RMA) (Pantoja et al., 2019) 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 (Ricci et al., 2009) can be used as a resource with or without a dedicated MAS (Manoel et al., 2020). 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 (Souza de Jesus. et al., 2021) 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 re-

sources 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 a serial interface named 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*. Then, when the device has a new resource connected the agent automatically receives this belief and can access the resources. 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., $port(name, status)$).

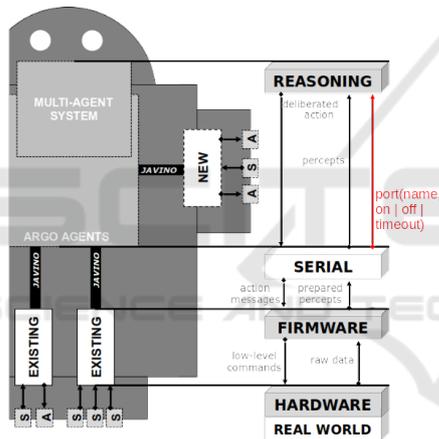


Figure 1: The four-fold architecture for programming Embedded MAS on top of hardware devices.

When connecting a new resource in the system, two possible approaches can occur: an agent needs to learn how to deal with this new resource, or can be employed a new agent to handle it. In these cases, the designer must program an external MAS and use an IoT network (Endler et al., 2011) 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. At last, a new Argo agent with the new desired abilities is transferred to the Embedded MAS using the bio-inspired protocols (Souza de Jesus. et al., 2021). Once the agent arrives at the destination, it can control the new resources and interact with the other existing agents in that system.

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.

To provide adaptability at Embedded MAS at runtime, mainly for the addition of new resources or for updating existing resources is mandatory that the system be built with a communicator agent connected to an IoT server. 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 that is capable of manipulating the resource to be added. Figure 2 presents the proposed approach for building an Embedded MAS capable of swapping resources at runtime.

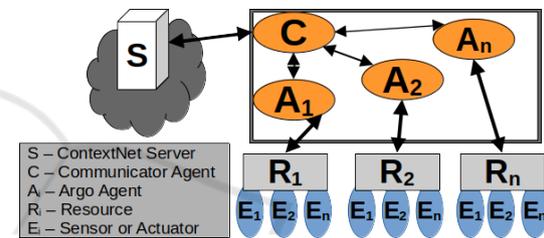


Figure 2: The swapping methodology for Embedded MAS.

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 $port(ttyACM0, off)$.

the minimum level and another to indicate the maximum level. Finally, resource 4 installed on the roof of the house consists of a sensor to indicate when it is raining.

This case study consists of analyzing six possible scenarios presented in Table 1.

Table 1: The summary of the case studies.

Scenario	Resource			
	1	2	3	4
0	operating	operating	operating	-
1	operating	operating	<i>failure</i>	-
2	<i>maintenance</i>	operating	operating	-
3	operating	<i>swapped</i>	operating	-
4	operating	operating	<i>updated</i>	-
5	operating	operating	operating	<i>added</i>

- (*Scenario 0:*) This scenario represents the normal functioning of the house. When the reservoir (resource 3) indicates a low water level, the power supplies (resources 1 and 2) must transfer water to the reservoir. The supply is interrupted if any source indicates that the water level is low. The supply must be stopped when the reservoir indicates that the water level is full.
- (*Scenario 1:*) This scenario represents a failure in resource 3, which is responsible for managing reservoir levels. If this failure occurs, the water supply must be immediately interrupted until the resource is available again.
- (*Scenario 2:*) This scenario represents maintenance on resource 1, responsible for managing the cistern (water captured by rain). When this resource is under maintenance, the water supply is managed only by resource 2.
- (*Scenario 3:*) This scenario represents a replacement of resource 2, which is responsible for managing the well (groundwater source). In this case, the resource will be replaced by another resource with different sensors. This new resource informs new types of perceptions from sensors to the Embedded MAS. In this case, it also maintains compatibility with the removed resource.
- (*Scenario 4:*) This scenario represents a replacement of resource 3, which is responsible for managing the water level in the reservoir (water tank). In this case, the resource will be replaced by a resource with different sensors incompatible with the removed resource.
- (*Scenario 5:*) This scenario represents the addition of a fourth resource to the Embedded MAS, which is responsible for sensing rain. This resource adds the ability to use rainwater in the house. Even if the reservoir is not indicating that

it is at the minimum water level when it is raining, it will request the activation of resource 1 until it is complete, looking to optimize the capacity of rainwater tank storage.

4.1 Embedded MAS Implementation

To fulfill the proposed scenarios, we implemented an Embedded MAS¹, which runs and controls the following physical devices: a single-board computer (Raspberry Pi 3) to host the reasoning layer and some microcontrollers (Arduino) to host the firmware layer. The Cognitive Hardware on Network - Operational System (ChonOS²) was used to develop the Embedded MAS, a specific-purpose GNU/Linux distribution for facilitating the development and debugging of agent-based embedded systems.

The microcontroller and the single-board computer communication are performed using a serial port, mediated by the Javino³ library. Figure 5 presents the schematic of the implementation.

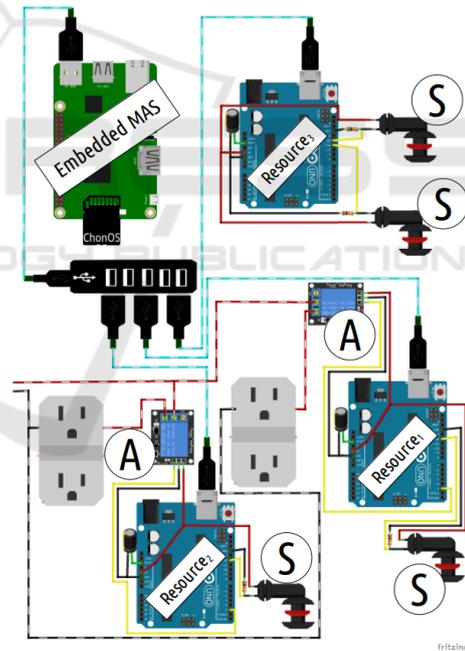


Figure 5: The device's physical components available for the Embedded MAS.

4.1.1 Firmware Layer

The microcontrollers respond to the agents, sending the perceptions gathered from sensors or executing

¹<http://icaart2023.chon.group/>

²<http://chonos.sf.net>

³<http://javino.sf.net>

the actuation commands. Resource 1 is connected to serial port `/dev/ttyACM0` and accepts the `cisternPumpOn` and `cisternPumpOff` commands to turn the cistern pump on and off. Additionally, it receives the `getPercepts` command at each cycle execution, which returns the following perceptions to the agent:

- `resource(cistern)`, the resource ID;
- `pump(cistern, on | off)`, indicating whether the uptake pump is on or off; and
- `level(cistern, empty | ~empty)`, indicating whether or not the cistern is at its minimum water level.

Resource 2 is connected to the serial port `/dev/ttyACM1` and accepts the `wellPumpOn` and `wellPumpOff` commands to turn the well pump on and off. Additionally, it accepts the `getPercepts` command, which returns the following perceptions:

- `resource(well)`, the resource ID;
- `pump(well, on | off)`, indicating whether the uptake pump is on or off;
- `level(well, empty | ~empty)`, indicating whether or not the well is at its minimum water level.

Resource 3 is connected to the serial port `/dev/ttyACM2` and accepts the `getPercepts` command, which returns the following perceptions:

- `resource(tank)`, the resource's identification;
- `level(well, empty | ~empty | full)`, indicating whether the water tank is empty, not empty or full.

4.1.2 Resources Swapped at Runtime

To exemplify scenario 3, a new resource 2 shown in Figure 6 was built. This resource differs from the old one by using an ultrasound sensor. This new resource model must be connected to the same serial port as the old one and be compatible with the commands accepted by the previous one to guarantee the success of scenario 3. In addition, it must maintain compatibility with the `getPercepts` command, returning information in the same format as the previous one. In addition, the resource may provide new beliefs. In this case, the resource informs the `precisionLevel(well, CM)`, indicating the supply source level in centimeters.

To exemplify scenario 4, was built a new resource 3 to manage the water reservoir level shown in Figure 7. Unlike the old one that used a level sensor, this one uses an ultrasonic sensor and provides the reservoir level only in centimeters. The `getPercepts` command returns the following information:

- `resource(tankNewModel)`, resource identification;
- `level(tankNewModel, CM)`, indicating the water level in the tank.

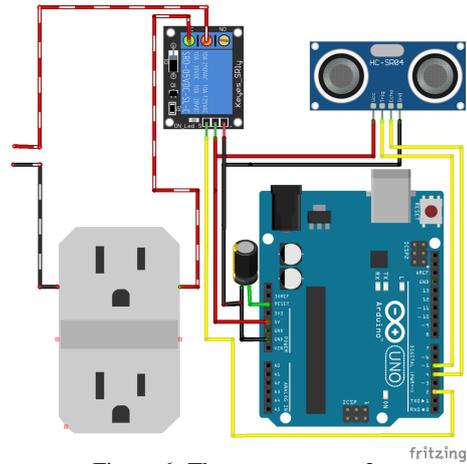


Figure 6: The new resource 2.

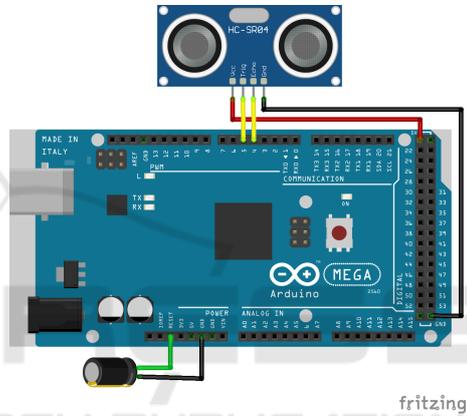


Figure 7: The new resource 3.

To exemplify scenario 5, the resource 4 shown in Figure 8 was built. It was connected to the `/dev/ttyACM3` serial port. It returns the following information:

- `resource(rain)`, the resource identification;
- `rainStatus(raining | ~raining)`, indicating whether or not it is raining.

4.1.3 Reasoning Layer

We implemented a MAS using the Jason framework in the reasoning layer, composed of five agents with well-defined functions. Three extended Argo agents, one Jason agent, and one communicator agent.

All Argo agents control the available resources of the house. Initially, the *CisternPhantom* controls resource 1, the *WellPhantom* controls resource 2, and the *TankPhantom* controls resource 3. They must connect to the respective serial port to control their resources. Then, they all have two initial beliefs, representing which serial port to access and the name of the resource: `myResourcePort(Port)` and `myRe-`

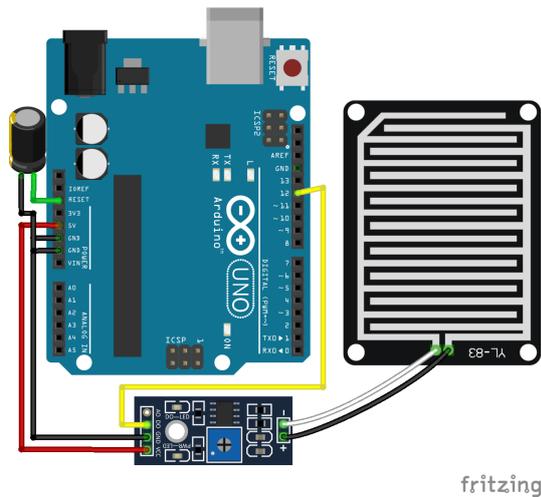


Figure 8: The resource 4.

source(Resource). Besides, Argo agents can control the interval time of gathering information from sensors and decide when open or close the flow of perceptions coming. *CisternPhantom* and *WellPhantom* agents has two achievement plans (*!conf* and *!pump(Op)*). The former sets up the serial port (*.port()*), the minimum interval of perceiving the environment (5 seconds), and opens the perception flow directly to the agent's mind (*.percepts(open)*). The latter has two possible contexts: send the microcontroller the commands to turn on or off the respective pump. Finally, it has a belief plan (*+level(R, S)*) that triggers a plan to turn off the pump (*!pump(off)*) if, during the water filling, the water level reaches the minimum. Codes 1, 2, and 3 shows the *CisternPhantom*, *WellPhantom*, and *TankPhantom* implementation.

The fourth agent (*Lurch*) is a Jason agent who manages the house's functioning. This agent has two achievement plans: *!getInformation* and *!pump(Op)*. The former updates its beliefs about the resources' state and agents every 5 seconds by consulting all agents (*.broadcast(askOne, Literal)*). The latter sends messages to the *WellPhantom* and *CisternPhantom* agents to turn on or off the water supply. In addition, it has three belief plans: the first (*+ready(no)*) is specific for if the reservoir agent is not available, it requests to interrupt the water supply; the last two (*+level(tank, S)*) request the activation or deactivation of the water supply, according to the reservoir level. Code 4 shows *Lurch* implementation.

The fifth agent (*Morticia*) is a Communicator agent that uses an IoT middleware to communicate with other MAS. Its initial beliefs have the information to connect with the ContextNet server. It is neces-

Code 1: *CisternPhantom* in *addamsMansion.mas2j*.

```
myResourcePort(ttyACM0).
myResource(cistern).
!conf.

+!conf: myResourcePort(R)<-
    .port(R); .limit(5000); .percepts(open).

+!pump(Op)[source(X)]: Op=on & ready(R) &
R=yes & level(LR,S) & S=~empty & pump(PR,PS)
& PS=off<-
    .act(cisternPumpOn).

+!pump(Op)[source(X)]: Op=off & ready(R) &
R=yes & pump(PR,S) & S=on <-
    .act(cisternPumpOff).

+level(R,S): myResource(MyR) & R=MyR &
S=empty & pump(P,F) & P=MyR & F=on <-
    !pump(off).
```

Code 2: *WellPhantom* in *addamsMansion.mas2j*.

```
myResourcePort(ttyACM1).
myResource(well).
!conf.

+!conf: myResourcePort(R)<-
    .port(R); .limit(5000); .percepts(open).

+!pump(Op)[source(X)]: Op=on & ready(R) &
R=yes & level(LR,S) & S=~empty & pump(PR,PS)
& PS=off<-
    .act(wellPumpOn).

+!pump(Op)[source(X)]: Op=off & ready(R) &
R=yes & pump(PR,S) & S=on <-
    .act(wellPumpOff).

+level(R,S): myResource(MyR) & R=MyR &
S=empty & pump(P,F) & P=MyR & F=on <-
    !pump(off).
```

Code 3: *TankPhantom* in *addamsMansion.mas2j*.

```
myResourcePort(ttyACM2).
myResource(tank).
!conf.

+!conf: myResourcePort(R)<-
    .port(R); .limit(5000); .percepts(open).
```

Code 4: *Lurch* in *addamsMansion.mas2j*.

```
!getInformation.

+!getInformation <-
  .abolish(ready(_)[-]);
  .abolish(pump(_,-)[-]);
  .abolish(level(_,-)[-]);
  .broadcast(askOne, ready(S));
  .broadcast(askOne, pump(P,S));
  .broadcast(askOne, level(P,S));
  .wait(5000);
  !getInformation.

+!pump(Op) <-
  .send(cisternPhantom,achieve,pump(Op));
  .send(wellPhantom,achieve,pump(Op)).

+ready(no)[source(tankPhantom)] <-
  .abolish(level(_,-)[source(tankPhantom)]);
  !pump(off).

+level(tank,S)[source(tankPhantom)]: S=full <-
  !pump(off).

+level(tank,S)[source(tankPhantom)]:S=empty<-
  !pump(on).
```

sary to have a Universally Unique Identifier (UUID), the address of a public IoT server, and the connection port. *Morticia* has two achievement plans: one to connect to the server (+!connect) and a plan for retransmitting messages received from other MAS to some internal agent (+!retransmit(*Dest*, *Force*, *Content*)). Code 5 show the *Morticia* implementation. Finally, it has a belief plan (+communication(*trying*)) to respond to any possible external communication attempt received.

4.2 Scenarios 0-3

The Embedded MAS operates properly for scenario 0. To comply with scenarios 1, 2, and 3, all Argo agents in the MAS have plans to deal with possible communication failures (scenario 1), unavailability (scenario 2), and even resource switching (scenario 3), as long as it maintains compatibility with beliefs previously known by the agent.

The Code 6 shows two belief plans: the first (+resource(*R*)) checks if the resource connected to the serial port the agent is managing is the same resource it expects. The second (+port(*Port*, *Status*)) has two distinct contexts: if the status is off, timeout, or on, the agent updates its belief (-+ready(*Literal*)) about

Code 5: *Morticia* in *addamsMansion.mas2j*.

```
myID("feee647d-c798-44c0-a6d2-099d88e8a59d").
cNAddress("skynet.chon.group").
cNPort(3273).
!connect.

+!connect: myID(ID) & cNAddress(S) &
cNPort(P)<-
  .connectCN(S,P,ID).

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

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

Code 6: Generic Beliefs Plans for all Argo Agents.

```
+resource(R): myResource(MyR) & MyR \== R <-
  -+ready(no).
+port(Port,Status): (Status=off |Status=timeout) <-
  -+ready(no).
+port(Port,Status): Status=on & resource(R) &
myResource(MyR) & MyR=R <-
  -+ready(yes).
```

being able to interface with the environment.

4.3 Scenario 4

Scenario 4 consists of exchanging a resource for another different from the existing one. In this scenario, the new resource must be connected to the same serial port as the old one. In addition, it will be necessary to transfer new plans so that the agent can continue to perform its function. A new MAS with only one communicator agent (*Cousin Itt*) was programmed on the developer's computer to transfer the necessary plans to the *TankPhantom* agent. Itt is responsible for connecting to the ContextNet server and transferring the plans to the communicator agent of the embedded MAS (*Morticia*). Code 7 presents the new plans.

The agent's initial beliefs are the UUID of the target communicator agent — *Morticia*, in Embedded MAS —, the UUID used to connect to the contextNet network, the server address, and the port. In addition, Itt has five achievement plans: the first (+!conf()) is responsible for connecting the Itt's MAS in the contextNet server; the second (!test) tries to communicate with the addressed MAS, sending a belief (communication(*trying*)) and expecting a belief in response (communication(*ok*)), thereby establishing the communication channel between the both MAS; the third

Code 7: *Cousin Itt* in *Itt's Office.mas2j*.

```
house("feee647d-c798-44c0-a6d2-099d88e8a59d").
myID("d1f8a5d5-b720-4a4b-88e2-2542a44a2964").
cNAddress("skynet.chon.group").
cNPort(3273).
!conf.
```

```
#!conf: myID(ID) & cNAddress(S) & cNPort(P) <-
  .connectCN(S,P,ID); +connected; !test.
```

```
#!test : connected & house(Morticia) & not
communication(ok) <-
  .sendOut(Morticia, tell, communication(trying));
  .wait(3000); !test.
```

```
#!test: communication(ok) <- !transmit.
```

```
#!transferKnowledge(R, K): house(M) <-
  .sendOut(M, achieve, retransmit(R,tellHow,K)).
```

```
#!requestExecution(R, O): house(M) <-
  .sendOut(M, achieve, retransmit(R,achieve,O)).
```

```
#!transmit <-
  !transferKnowledge(tankPhantom,
    "+!newResource(X)[source(Z)] <-
    .percepts(close); .abolish(resource(_)[_]);
    .abolish(myResource(_)[_]);
    .abolish(ready(_)[_]);.abolish(port(,-)[_]);
    .abolish(level(,-)[_]);
    +myResource(X); .percepts(open.)");
  !requestExecution(tankPhantom,
    newResource(tankNewModel));
  !transferKnowledge(lurch, "+!newPerceptions:
    level(tankNewModel,V)[source(tankPhantom)]
    & V<10 <- !pump(inactive); .wait(5000);
    !newPerceptions.");
  !transferKnowledge(lurch, "+!newPerceptions:
    level(tankNewModel,V)[source(tankPhantom)]
    & V>50 <- !pump(active); .wait(5000);
    !newPerceptions.");
  !transferKnowledge(lurch, "+!newPerceptions <-
    .wait(5000); !newPerceptions.");
  !requestExecution(lurch, newPerceptions);
  .disconnectCN;
  .stopMAS.
```

(*!transferKnowledge(R, K)*), sends plans to a specific agent in the Embedded MAS; the fourth (*!requestExecution(R, O)*) sends an achieve message to be pursued by a specific agent in the Embedded MAS; the last one (*!transmit*) is the plan responsible for transferring the new plans to the agents in the Embedded MAS.

4.4 Scenario 5

Scenario 5 adds an entirely new resource to the Embedded MAS. For this, it is necessary to verify which serial port the resource will acquire at the destination MAS and send a new agent to control the new resource. To add a new resource at runtime in the Embedded MAS (*addamsMansion.mas2j*), another MAS was created on the developer's computer (*uncleFesterLab.mas2j*). This MAS has a communicator agent (*uncleFester*) responsible for transferring a second agent *Argo* (*roofPhantom*), to the Embedded MAS.

Code 8 presents *UncleFester* plans. *UncleFester* has the following initial beliefs: Morticia's UUID, its own UUID, the ContextNet server address, and the port. The agent has five achievement plans: the first (*!conf*) connects to the ContextNet server; the second (*!test*) tests the connection with the communicator agent of the destination MAS (Morticia) by sending a belief referring to the connection, and a belief is expected in response. The third (*!sendAgent(Agent)*) contains the actions to activate the transferring protocol; the fourth (*!requestExecution(R, O)*) sends an achievement message to the Embedded MAS to execute a specific plan. The fifth (*!transmit*) orchestrates how to send the agent and the achievement message to the Embedded MAS.

Finally, Code 9 presents the *RoofPhantom's* plans that will be transferred to the Embedded MAS. This agent has the following initial beliefs: the serial port address of the device it will manage (*myResourcePort(ttyACM3)*); and the name of the resource it will manage (*myResource(rain)*). This agent has two achievement plans: one to set up the serial interfacing (*!conf*) and a plan with the actions to request the activation of the water pump from the cistern during the rain. It also has a plan in case the water reservoir is full during the rain. Finally, it has a belief addition plan, which requests information from the reservoir when it identifies the onset of rain.

5 CONCLUSIONS

This work presented a methodology for swapping resources at runtime in Embedded MAS using Jason and customizable agent architectures capable of interfacing hardware and moving plans and agents through an IoT network. Adding resources allows an Embedded MAS to be updated and improved at runtime without having to stop it. Stopping a MAS can lead to some undesired situations, for example, in a mission-critical domain, which could generate fail-

Code 8: *Uncle Fester* in *FesterLab.mas2j*.

```
house("f6ee647d-c798-44c0-a6d2-099d88e8a59d").
myID("19566fee-4bc6-45eb-8f72-455552d50116").
cNAddress("skynet.chon.group").
cNPort(3273).
!conf.

+!conf: myID(ID) & cNAddress(S) & cNPort(P) <-
    .connectCN(S,P,ID);
    +connected; !test.

+!test : connected & house(Morticia) & not
communication(ok)<-
    .sendOut(Morticia, tell, communication(trying));
    .wait(3000);
    !test.
+!test: communication(ok) <-
    !transmit.
+!sendAgent(Agent): house(H) <-
    .moveOut(H,mutualism,Agent).
+!requestExecution(R, O): house(H) <-
    .sendOut(H, achieve, .retransmit(R,achieve,O)).
+!transmit <-
    !sendAgent(roofPhantom);
    !requestExecution(roofPhantom, conf);
    .disconnectCN;
    .stopMAS.
```

Code 9: *roofPhantom* in *festerLab.mas2j*.

```
myResourcePort(ttyACM3).
myResource(rain).

+!conf: myResourcePort(R)<-
    .port(R);
    .limit(5000);
    .percepts(open).

+!economizeWater: level(tankNewModel,V)
    & V>10
    & V<50 <-
    .send(cisternPhantom,achieve,pump(on)).

-!economizeWater <- .
+rainStatus(raining) <-
    .send(tankPhantom, askOne,
        level(tankNewModel,V), Reply);
    -+Reply;
    !economizeWater.
```

ures 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. Currently, 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 our approach, 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 so as not to pursue 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.

Regarding the composition of agents of an Embedded MAS for the swapping of resources at runtime, it is mandatory to have a Communicator agent to send agents and plans from one MAS to another since the communication between different MAS happens using an IoT network which only these agents have access. Besides, these agents are responsible for invoking bio-inspired protocols. In this work, we use Mutualism for its non-destructive behavior for the origin and destiny MAS. All agents in our methodology are specialized by option. Every agent has specific skills (e.g., communicators, communicate and Argo agents interface hardware). The existence of hybrid agent architectures would be possible. However, such an option could overload the agent's reasoning since both the flow of messages and perceptions can generate undesirable bottlenecks in processing.

The swapping of 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). In future work, 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 the need to transfer agents from other systems. For example, if the house presented in the case study is using a dynamic mechanism addition of resources in its Embedded MAS, it would be enough to connect it to the house, and all the necessary skills would be automatically loaded into the Embedded MAS. For instance, one limitation is that the agent needs to know the hardware being added and depend on an available IoT infrastructure.

REFERENCES

- Artikis, A. and Pitt, J. (2008). Specifying open agent systems: A survey. In *International Workshop on Engineering Societies in the Agents World*, pages 29–45.
- Bordini, R., Hübner, J., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. Wiley.
- Brandão, F. C., Lima, M. A. T., Pantoja, C. E., Zahn, J., and Viterbo, J. (2021). Engineering approaches for programming agent-based iot objects using the resource management architecture. *Sensors*, 21(23).
- Bratman, M. E. (1987). *Intention, Plans and Practical Reasoning*. Cambridge Press.
- Bratman, M. E., Israel, D. J., and Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(3):349–355.
- Dennis, L. A. and Farwer, B. (2008). 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*, pages 16–23.
- Endler, M., Baptista, G., Silva, L. D., Vasconcelos, R., Malcher, M., Pantoja, V., Pinheiro, V., and Viterbo, J. (2011). 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*, New York, NY, USA. Association for Computing Machinery.
- Hamdani, M., Sahli, N., Jabeur, N., and Khezami, N. (2022). Agent-Based Approach for Connected Vehicles and Smart Road Signs Collaboration. *Computing and Informatics*, 41(1):376–396.
- Lazarin, N. M. and Pantoja, C. E. (2015). A robotic-agent platform for embedding software agents using raspberry pi and arduino boards. In *9th Software Agents, Environments and Applications School*.
- Manoel, F., Pantoja, C. E., Samyn, L., and de Jesus, V. S. (2020). Physical Artifacts for Agents in a Cyber-Physical System: A Case Study in Oil & Gas Scenario (EEAS). In *SEKE*, pages 55–60.
- Matarić, M. J. (2007). *The Robotics Primer*. Mit Press.
- Michaloski, J., Schlenoff, C., Cardoso, R., Fisher, M., and others (2022). Agile Robotic Planning with Gwendolen.
- Pantoja, C. E., Soares, H. D., Viterbo, J., Alexandre, T., Seghrouchni, A. E.-F., and Casals, A. (2019). Exposing iot objects in the internet using the resource management architecture. *International Journal of Software Engineering and Knowledge Engineering*, 29(11n12):1703–1725.
- Pantoja, C. E., Stabile, M. F., Lazarin, N. M., and Sichman, J. S. (2016). Argo: An extended jason architecture that facilitates embedded robotic agents programming. In Baldoni, M., Müller, J. P., Nunes, I., and Zalila-Wenkstern, R., editors, *Engineering Multi-Agent Systems*, pages 136–155, Cham. Springer International Publishing.
- Pokahr, A., Braubach, L., and Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In *Multi-agent programming*, pages 149–174. Springer.
- Ricci, A., Pianti, M., Viroli, M., and Omicini, A. (2009). *Environment Programming in CArtaGO*, pages 259–288. Springer US, Boston, MA.
- Silva, G. R., Becker, L. B., and Hübner, J. F. (2020). Embedded architecture composed of cognitive agents and ros for programming intelligent robots. *IFAC-PapersOnLine*, 53(2):10000–10005. 21st IFAC World Congress.
- Souza de Jesus., V., Pantoja., C., Manoel., F., Alves., G., Viterbo., J., and Bezerra., E. (2021). Bio-inspired protocols for embodied multi-agent systems. In *Proceedings of the 13th International Conference on Agents and Artificial Intelligence - Volume 1: ICAART.*, pages 312–320. INSTICC, SciTePress.
- Stabile Jr., M. F., Pantoja, C. E., and Sichman, J. S. (2018). Experimental Analysis of the Effect of Filtering Perceptions in BDI Agents. *International Journal of Agent-Oriented Software Engineering*, 6(3-4):329–368.
- Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. Wiley.