

Towards a Docker-based architecture for open multi-agent systems

Gustavo Lameirão de Lima, Marilton Sanchotene de Aguiar

Graduate Program in Computing (PPGC), Federal University of Pelotas (UFPEL), Pelotas, Brazil

Article Info

Article history:

Received Nov 8, 2022

Revised Jul 10, 2023

Accepted Aug 2, 2023

Keywords:

Agent-based simulation

Docker

Multi-agent systems

Open multi-agent systems

ABSTRACT

In open multi-agent systems (OMAS), heterogeneous agents in different environments or models can migrate from one system to another, taking their attributes and knowledge and increasing developing complexity compared to conventional multi-agent systems (MAS). Furthermore, the complexity of opening may be due to the uncertainties and dynamic behavior that the change of agents entails, needing to formulate techniques to analyze this complexity and understand the system's global behavior. We used Docker to approach these problems and make the architecture flexible to handle distinct types of programming languages and frameworks of agents. This paper presents a Docker-based architecture to aid OMAS development, acting on agent migration between different models running in heterogeneous hardware and software scenarios. We present a simulation scenario with NetLogo's Open Sugarscape 2 Constant Growback and JaCaMo's Gold Miners to verify the proposal's feasibility.

This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.



Corresponding Author:

Gustavo Lameirão de Lima

Graduate Program in Computing (PPGC), Federal University of Pelotas (UFPEL)

Pelotas, Rio Grande do Sul, Brazil

Email: gustavolameirao@inf.ufpel.edu.br

1. INTRODUCTION

Currently, several areas use concepts of artificial intelligence to solve problems, and often the solutions are based on shallow learning (such as gradient boosting, random forest), deep learning, or heuristics (genetic algorithms). These solutions perform as the problem is well defined and the dataset is large and centralized [1]. However, there are also dynamic problems with several interacting entities, and it is necessary to use solutions with decentralized control capable of adapting at runtime [1]. In this context, multi-agent system (MAS) is another solution in the artificial intelligence field.

These systems are composed of multiple agents, physical or virtual entities with autonomous behavior, and can act on their own [2], [3]. These agents can perceive the environment in which they are, and through actuators, they can act in that environment. In MAS, agents can exchange information for different purposes, aiming to obtain collective knowledge, update beliefs, and optimize strategies.

A specific type of MAS still allows interaction between agents participating in different models, the open multi-agent systems (OMAS). In OMAS, we analyze heterogeneous agents (with other characteristics) from different perspectives (inserted in different environments/models) that can migrate from one system to another, taking their attributes and knowledge [4]. The heterogeneity of agents may come from some differences between the models, such as architecture, objectives, or policies [5].

However, different problems arise when developing applications in OMAS when compared to MAS [6]. First, there may be implementation problems where agents and models can be created by differ-

ent teams, in other programming languages, or even in various platforms/agent architectures. Often conflicts of objectives may not guarantee that agents will act cooperatively and coordinately [7], [8]. Moreover, there is still the difficulty generated by the uncertainties and the dynamic behavior that the change of agents entails.

OMAS need to deal with problems not present in closed systems. For example, the migration of agents between models can occur at runtime, and the motivation for this migration of an agent from one system to another can be different, usually of the developer's choice, such as execution failures, self-will, or some trigger. In addition, conflicts of interest may occur between the new agents when not designed to work in that set [5]. Although the subject has been known for a long time [9], there is still research in the area, as in [10]-[15].

The concept of openness reduction is related to the model's overload from the perspective of the possibility of agents entering and exiting the system without changing the design of many components. Thus, when the greater the degree of openness, the smaller the number of changes to a model to receive or send agents [4]. In this way, a perfect open system would not need transformations (0 steps) to accommodate new components, while on the other side, there are systems that would require a complete redesign when new agents arrive [4]. Therefore, given the context presented, it is necessary to make a research effort to reduce the complexity of OMAS regarding the cited problems.

In this context, Docker could solve these problems because it is an open-source tool that automates container application development, adding an application deployment engine to a virtualized container execution environment. This design creates a lightweight and fast environment that runs code and tests in a single flow, from development to production. This paper presents a Docker-based model to aid in developing OMAS and to facilitate the migration of agents between different models that can run in heterogeneous hardware and software scenarios, different development environments, and tools for MAS or other operating systems.

There are differences between virtual machines and containers because virtual machines are an abstraction of physical hardware that transforms one server into multiple servers. The hypervisor, software that creates and runs the virtual machine, allows multiple machines to run on a single host. Each machine includes a complete copy of an operating system, the application, required binaries, and libraries. Unfortunately, machines can also be slow to boot [16]. Unlike virtual machines, containers are an abstraction in the application layer that binds code and its dependencies together. Multiple containers can run on the same machine sharing the operating system kernel, each running as isolated userspace processes. Container images are more space-optimized than virtual machines [16].

By implementing specific agent registration and routing modules to define system steps as the criteria for determining where agents will go, the original models only need to describe how they will send and receive agents, decreasing transaction complexity within the model. In addition, agents that move between models will transfer knowledge, carrying their attributes with them. In addition, the use of Docker allows for greater modularization of the system, where the implementation of each module within a container makes it possible to run different tools on different operating systems and allows easy code replacement by changing the execution of the container [17].

This modularization allows, for example, to extend of an initial structure of agent routings between models by a model that can retrain agents before returning to the simulation models. Finally, modularization will allow it to add new behaviors to the system that models did not foresee in its conception phase. In general, the approach aims to advance the generalization of OMAS, simplifying the opening process.

We organized this paper is first, we present related work in section 2. Then, section 3 presents details of the proposed approach, flow, implementation details, and the schedule of activities. Next, section 4 shows the results obtained, emphasizing the developed simulation scenario, which is of great importance to verify the feasibility of the implementation. Finally, section 5 presents the final considerations of the study.

2. RELATED WORK

The first perspective from most of the related work considers OMAS from an architecture/organization point of view, as in [4], [6], [12], [18]-[23]. These studies differ from the approach of this work, mainly because, among all OMAS problems, we focus on the openness part, allowing the agents to move between models. The effects of the conflicts of interest that could come with the agents moving between models are something model-related, requiring to be resolved by the model using any technique, such as the ones described in this study and others. Our goal is to provide mechanisms to simplify the agents' movement between models reducing changes in the model's original code. Furthermore, [1], [5], [24]-[26], present similar aspects with some parts of our

approach. We will describe in the following paragraphs the main similarities and differences between those studies.

Ramirez and Fasli [24] propose a model that uses both NetLogo and Jason agents in an especially complex model for cognitive agents, a Disaster-Rescue simulation. The approach to bring those two different architectures of agents, connecting Jason to NetLogo, is to either include part of NetLogo's internal classes inside the Jason code or do the opposite, including Jason's internal classes inside the NetLogo code since both applications use Java code. Even though this study considers a closed multi-agent model (not an open one), it has some similarities to our study in that they have communication between agents with different architectures. The main difference is that, in this case, both applications use Java code. We point out a limitation of this work if some agent programmer wants to use this approach to communicate, for instance, NetLogo, to another agent platform that uses another language. Our approach creates a platform that runs isolated code (inside containers) that allows completely different programming environments, allowing the developer to run any code (Java, Python) through our application programming interface (API) service.

Uez [5] propose a methodology for specifying OMAS. The structure uses two main ideas: the independent modeling of each of the dimensions of the system (agent, environment, and organization); and the specification of edge concepts, providing information at design time that helps include at runtime the elements of the dimension open. Furthermore, the authors designed elements targeting code for the JaCaMo framework in the implementation. In addition, the study has two case studies, which allow viewing the results throughout the development phases. The main difference between this study and our approach is that the programmer must adapt the model and architecture to the solution. In contrast, our architecture can be adapted to the current running implementation that the programmer has, just using the I/O mechanisms that the agents must have to communicate with our solution. Compared to the parameters analyzed in the study, our platform works on the agent dimension and the implementation phase (the study deals with agent/environment/organization dimensions and analysis/project/implementation phases). We deal with the agent dimension because our primary focus is to transport agents between models, even though it is possible to extend our platform to transport parts of the environment, like artifacts. On the other hand, we focus the approach on the implementation phase because we want the programmer only to rebuild part of the model to the platform. In this way, the programmer needs to include the mechanisms to communicate with our platform. That way, the primary usage of our platform is in a model already built (implementation phase).

Perles *et al.* [1] present the development of a Java-based framework for the development of adaptive open multi-agent systems. The framework developed by the authors, named AMAK, uses three fundamental classes based on object-oriented principles: adaptive multi-agent system (AMAS), Agent, and Environment. Whoever uses the framework must implement these abstract classes, adapting to the model. Furthermore, the authors use developing a socio-technical environmental system as a case study to bring environmental well-being. The architecture proposed in this study is mainly diverse from ours because it is almost a new programming language since the authors compare their results with other agent programming languages (such as Jade, NetLogo, and GAMA). However, our approach does not require the programmer to remake its model into something new. Instead, the programmer must insert the code to allow the agents to communicate with the architecture, which leads to less effort to allow the agents to move between the models.

Pfeifer *et al.* [26] propose a MAS to monitor and manage container-based distributed systems. Their system allows users to observe and verify the quality and progress of the application over time, improving parameters such as quality of service (QoS). This study is unrelated to OMAS but has concepts similar to those we used in our approach. Furthermore, they encourage the connection between MAS and DevOps, using DevOps tools such as Docker. The main difference is that we do not use Docker to monitor MAS systems. Instead, we run our system in a container-based approach on Docker.

Dähling *et al.* [25] use MAS in the internet of things (IoT) field since both share similarities (distributed devices, cooperation). They mainly use MAS in IoT to build large-scale and fault-tolerant systems. They propose a cloud-native multi-agent platform (MAP), named cloneMAP, to use cloud-computing techniques to enable fault-tolerance and scalability. This approach is related to MAS rather than to OMAS. It is similar to our study using DevOps tools associated with MAS, such as Docker. Still, the study's primary goal is not related to openness and allowing agents to move between models. Also, similarly to [1], this study compares results directly to agent programming platforms, such as Java Agent Development Framework (JADE), which differs from our approach. In our approach, we keep the model similar to before the openness making changes to insert the mechanisms to allow the model to communicate with our platform.

Table 1 summarizes the similar studies found, indicating if it is focused on the organization aspect of OMAS, if it uses some DevOps tools, and deal with more than one MAS platform. The aspect of DevOps is important because it is a way to implement multiple parts of the systems that could run different languages and OS, making it possible to use all kinds of different MAS tools. The last aspect is how many MAS platforms/tools are being used. This aspect is essential to making the tool max embracing as possible.

Table 1. Summarization of related work

Work	Year	Organization-related	DevOps Tools	MAS Platforms
[4]	2013	Yes	–	Single
[6]	2010	Yes	–	Single
[18]	1996	Yes	–	Single
[19]	2006	Yes	–	Single
[20]	2012	Yes	–	Single
[21]	2003	Yes	–	Single
[22]	2009	Yes	–	Single
[23]	2021	Yes	–	Single
[12]	2020	Yes	–	Single
[24]	2017	Other	–	Multiple
[5]	2018	Other	–	Single
[25]	2021	Other	Docker and Kubernetes	Single
[26]	2021	Other	Docker	Single
[1]	2018	Other	–	Single
Our Approach	2022	Other	Docker	Multiple

From the 15 studies analyzed, the main focus of 9 of them is to deal with organizational problems that come with the openness from OMAS. Of the other 6, 3 are language dependent, not allowing the usage of MAS/OMAS tools that runs in a different development/usage environment, as another tool that uses another language. Lastly, besides ours, only one study deals with more than one MAS platform. This study is only MAS, not OMAS-related, so it does not deal with the problem of transporting agents between models. Also, this study connects two MAS platforms that run in the same language (Java), not considering other tools that might use other languages.

In conclusion, our approach contributes to the state-of-art in how the programmer will use the platforms. For example, some studies propose new approaches that require the programmer to rebuild their models according to the proposes. In our architecture, the programmers would not have to change their entire model. Instead, they will add the structures needed to their models to communicate with our architecture, allowing an easier transition from a closed MAS to an open MAS. Another related work already used container/cloud-based approaches related to the MAS system, showing that this approach is promising. Also, some studies have tested the communication between NetLogo and Jason agents.

3. THE PROPOSED APPROACH

Our approach aims to develop an environment that facilitates the development of OMAS using Docker [17] as a basis, allowing the migration of agents between models that can run in heterogeneous scenarios. The structure allows code to run inside containers that contain the same operation structure where they were developed, avoiding problems like programs that run in the development stage but in the production stage have problems. In our approach, each image is generated based on a description file, called *DockerFile*, containing information on the structures needed in the container, such as the operating system, programming languages, and code to be executed. In addition, Docker has an image bank, the *Docker Hub Container Image Library*, which contains the images frequently used by developers, extending the approach's applicability.

The containers are responsible for executing each essential block of the architecture structure. Therefore, they can be easily replaced or added, making the architecture more modularized and adaptable to new scenarios not foreseen in the conception, making it more robust. We also deal with security concerns, such as creating separate virtual networks so that each container has a partial view of the system, limiting access to sensitive code. Finally, our approach can use online platforms such as Amazon's (AWS Docker) to run code in the cloud due entire structure being developed based on Docker, which enables high application portability, expanding the universe of applications in the architecture.

The Figure 1 presents a general description of the proposed architecture in a block diagram format. In the Docker-compose block, we have the generation of images and services based on the *docker-compose.yaml* file and the *Dockerfile* of each service. Each *Dockerfile* contains the sequence of instructions needed to assemble the image with all the requirements that each service needs. Then, based on the images, Docker-compose uses the parameters of each service (container name, exposed ports, network, volumes, command/file to be executed) and builds them. Finally, when we define all services on the Docker-compose, they can be executed using just one command.

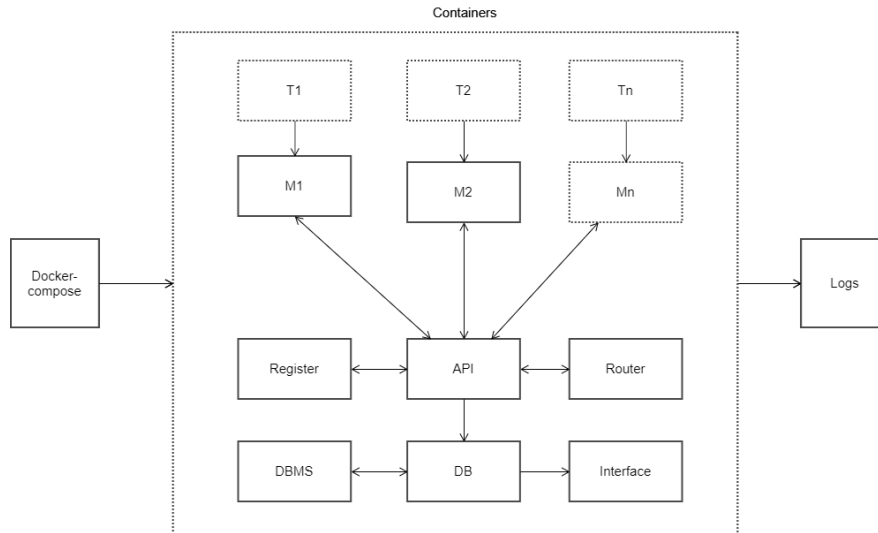


Figure 1. Organization of the proposed architecture

The blocks M_1, M_2, \dots, M_n represent the containers responsible for running the models. In the main configuration file, the designer can define a parameter *auto run*. When we set this parameter to *True*, the models are executed automatically when the container is mounted. If it is *False*, we must start execution of the templates by a trigger container (T_1, T_2, \dots, T_n). It is necessary to implement communications within the models' architecture, such as triggers and functions of how agents enter/exit the model. So far, there are containers ready to support NetLogo (Java container) and JaCaMo (Java via Gradle) models (see Section 4. for more information).

T_1, T_2, \dots, T_n are execution trigger containers, responsible for sending a message to the models to start their execution. We use these containers when necessary to implement a more robust logic to start executing the models. For example, they can run codes that read a sensor, wait for measurement, or run a certain model only when agents are assigned.

The application programming interface (API) block is the container responsible for managing access to the database. This container acts as an intermediary when any system part must write, read, update, or delete information from the database. Currently, the API is implemented in Python, using the Flask framework [27]. All the methods are accessible via HTTP, using JavaScript Object Notation (JSON) format on every message, as a standard for APIs.

The database (DB) block represents the container responsible for the database where we store, for each model, all agent arrival/departure information to be accessed by other containers that may need this information. It is important to note that we do not implement business rules in the DB. Also, this container is responsible for carrying out database operations (read, write, update, and delete). How containers must manipulate the information before or after involving the database is their responsibility, which will only pass the final command to this one.

The database management system (DBMS) block represents the container that facilitates access to the DB. It provides a Web interface (by default exposed to the host machine) that can import/export structured query language (SQL) content/files and view the information in real-time. This container can help with debugging and making logs of the application. Also, the DBMS must be chosen according to the DB because they must be compatible.

The register block is a container responsible for managing all agents on the platform. Every agent must have an identification to be used by the architecture, so the first insertion requests for new agents coming from any model containers need a unique identification. So, the Register is responsible for generating a unique identification for every agent, then forwarding them back to the environments.

The router block is the model responsible for receiving all agents that left a given model, analyzing and judging which model to send the agent. This step specifies the agent's entry and exit protocols from different containers/models. This block is an essential part of the proposal because models delegate to Router the distribution task of the agents among the models. When the simulation environment does not share (or only partially) information about the world, the Router can handle several problems related to the absence of this information. Judgment can occur differently, such as analyzing the most promising agents, machine learning codes, and executing a new MAS model that retrains agents. In the simulation scenario, there are two judgment options that the Router can make of the list of agents to be processed: i) randomly chooses the agent and the target model (random mode); and ii) process agents in a single queue, considering all models, and randomly defines the destination model (general sequential mode).

The interface block is a container that receives agent movement information through the system and generates reports exposed and formatted to view metrics of the architecture's execution. In the current implementation, this container has an Apache Webserver running PHP (Hypertext Preprocessor) code that reads all information from agents that have already gone through the Router and generates a report with all agents, attributes, and the path they took. The general form of the report is through a front-end (hypertext markup language (HTML)), cascading style sheet (CSS), and JS (JavaScript) code), which the host can access by exposing the port that apache runs on port 80 (by default).

Each tuple contains all essential attributes for every agent interaction so far. In the end, it presented the longest path taken by agents. Notably, a search option exists to filter the results by any of the columns. Also, it is possible to filter results by agent, model, and whether the tuple has been processed by the model/router. Finally, each agent already processed by the Gold Miners model (JaCaMo container) has a *.asl* file (AgentSpeak Language) containing relevant information to the model. So, on this interface, the user can check the *.asl* to see its content.

Finally, the logs block is not a container but a module responsible for generating logs of the outputs of each container in the system. We can obtain several types of logs from the simulation, such as regular docker logs (via docker log command), DB tuples via SQL export, and prints on txt files (using terminal operators when running the container on docker-compose), for example. In addition, some structures have particular log types, such as JaCaMo's default log generator (based on Java's logging API). These logs can assist in debugging tasks or generating data for execution analysis.

4. RESULTS

In the proposed simulation scenario, we built containers to support two different development tools for MAS: NetLogo [28] and JaCaMo [29]. We choose two models from each tool's documentation: NetLogo's Open Sugarscape 2 Constant Growback [30] and JaCaMo's Gold Miners [31], with adaptations. We used three model containers, running two isolated copies of Open Sugarscape and one of Gold Miners. The main objective of this simulation scenario is to validate the approach feasibility, allowing agents to move between models freely, even though the two tools use different agent architectures.

4.1. The open sugarscape model's adaptations in NetLogo

The Open Sugarscape model is a simulation used in artificial societies, simulating a population with limited resources where we represent each agent with an ant that, to survive, must move around the environment in search of sugar. Each ant is born with an amount of sugar (from 5 to 25), metabolism (from 1 to 4), and vision (from 1 to 6). Metabolism defines how much energy the ant loses when moving. If the ant's energy reaches zero, it dies and leaves the model. Vision determines how many distance positions the ant can see sugar from its initial position. Figure 2 illustrates the NetLogo interface simulating the original model.

In Figure 2, we can view the environment, input parameters, and some metrics. For example, it is possible to configure the number of agents at startup and the buttons to start, execute, and pause the code on the left. In the center, we can see the environment. The red dots represent the agents, while the others represent the sugar inserted into the environment, ranging from yellow to white, respectively, from more to less intensity. Finally, graphs represent simulation output metrics, such as population, metabolism, and vision averages.

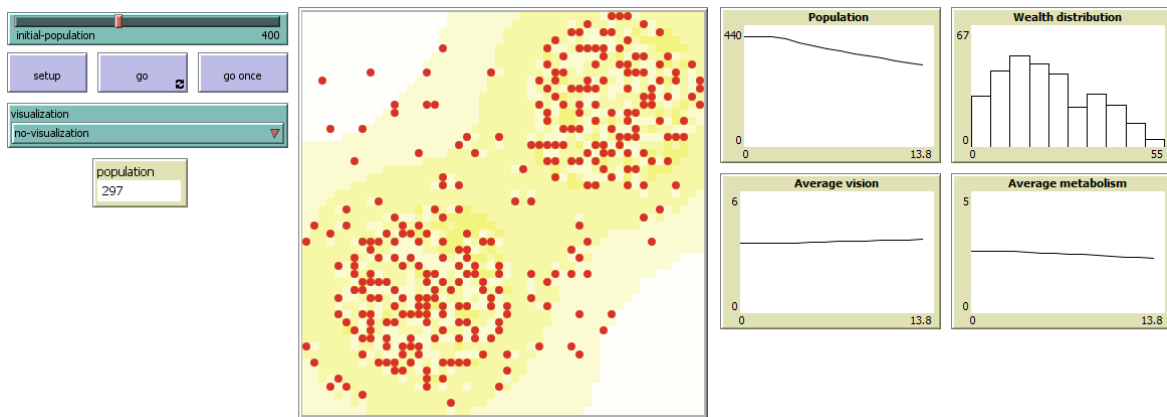


Figure 2. Sample sugar scape model in NetLogo IDE

Figures 3(a) and 3(b) show, respectively, the source code needed for the NetLogo can receive and send data from the architecture. Our approach provides functions *check_new_agent_on_api* and *send_agent_to_api* to the user for setting the triggers to send (and what information should be sent) and receive an agent. For this specific model, when an agent dies (according to the original model, an agent dies when its food becomes zero), the model sends this agent to the architecture and removes this agent from the simulation.

```

to check_new_agent_on_api
  py:setup py:python
  py:run "from db_python_netlogo import receive_agent"
  let result py:runresult (word "receive_agent('" ("m1") "'")")
  ifelse(length result > 0)
  [
    foreach result
    [
      x ->
      let tuple read-from-string item 1 x

      create-turtles 1
      [
        set agent_id item 0 x
        set sugar random-in-range 5 25
        set metabolism item 1 tuple
        set vision item 2 tuple
        set historic item 2 x

        set shape "circle"
        move-to one-of patches with [not any? other turtles-here]
        set vision-points []
        foreach (range 1 (vision + 1)) [ n ->
          set vision-points sentence vision-points (
            list (list 0 n) (list n 0) (list 0 (- n)) (list (- n) 0))
          ]
        run visualization
        print "Agent created"
      ]
    ]
  ]
  [
    print ("There is no new agent on DB")
  ]
end

```

(a)

```

to send_agent_to_api
  py:setup py:python
  py:run "from db_python_netlogo import send_function"
  ;;Example: "send_function('12', '[1 2 3]', '1-1')")

  let updated_historic ""
  ifelse (historic = "")
  [
    set updated_historic 1
  ]
  [
    set updated_historic (word "" (historic) "-1")
  ]
  let tuple []
  set tuple lput sugar tuple
  set tuple lput metabolism tuple
  set tuple lput vision tuple

  let result py:runresult (word
    "send_function('" (agent_id) "' , '" tuple "' , '" (updated_historic) "'")
  )

  print("Agent sent?")
  print(result)
end

```

(b)

Figure 3. NetLogo functions responsible for (a) receiving and (b) sending agents between the architecture and the NetLogo model

We made another adjustment when the agent returned to the simulation; instead of using his last information about food, we generated this parameter randomly, while the other parameters were the same as the last simulation. We made that adjustment because of the way that the original simulation works. If the agent leaves the simulation when its food is zero, and we use that same information again when the agent returns, it would cause a loop because the agent would get to the simulation with zero food and die again.

The only two attributes added on the NetLogo agents are *agent_id* and *historic*. We used *agent_id* to check the agent's unique id for the architecture, while the regular id is used just for the NetLogo's simulation. The attribute *historic* shows the path of what models the agents went through.

The only dependency required is Py NetLogo's extension to adapt any NetLogo model to our platform. We need this extension to send/receive information from the API through Python code. We have small functions to do so, and all the programmer has to do is include those functions and use them whenever the model needs to send/receive information about the agents. We have examples of how to adapt and use our functions on GitHub.

4.2. The Gold Miners model's adaptations in JaCaMo

Figure 4 illustrates the JaCaMo interface simulating the Gold Miners model. This model simulates a set of agents representing miners whose role is to navigate the environment. For example, when an agent finds a gold node, it stops his current objective, gets the gold, and brings it back to the central deposit. In the Figure, it is possible to see the representation of the agents (blue dots), the deposit (box with X in the center), the environmental barriers (positions not accessible to the agents, black boxes), the gold nodes (yellow boxes), the positions already visited by agents (white boxes) as well as those not accessed (gray boxes).



Figure 4. Gold Miners running on JaCaMo graphical user interface (GUI)

An excerpt of the source code (Java and ASL) needed for the JaCaMo can send and receive data from the architecture is shown in Figure 5. Figures 5(a) and 5(b) show the Java code for receiving and deleting agents. Figures 5(c) and 5(d) show the ASL code for checking for new agents and removing excluded agents from the architecture. The architecture presents two agents for sending/receiving information: *killer_agent* and *check_new_agents*. The *killer_agent* waits for an agent's message to be removed from the simulation. When it happens, this agent saves all the agent's information in a *.asl* file (that will be used when this agent reenters the simulation) and then removes the agent from the simulation.

The *check_new_agents* job is to check if an agent is waiting to enter the current model. If so, the agent is inserted into the model. This agent uses Java and AgentSpeak Language (ASL) code to communicate with the API and insert the agent into the simulation. We can also adapt this code to insert extra initial information to the agent, like beliefs, goals, or focus on Cartago artifacts. When the model includes an agent in the simulation, *check_new_agents* verifies if there is a *.asl* file with the agent id, that is, that the new agent has already been on the model. If the file exists, the model creates the agent with its previous information (using the previous *.asl* file). If not, the model creates the agent with a fresh new *.asl* file used as a template.


```

System.out.println("Sugar: " + sugar);
System.out.println("Metabolism: " + metabolism);
System.out.println("Vision: " + vision);
char ch="";
String bels = "agent_id("+agent_id+")";
bels = bels + ",path("+ ch + agent_path + ch + ")";
bels = bels + ", sugar("+sugar+")", metabolism("+"metabolism+"), vision("+"vision+"");
System.out.println("Agent bels: "+bels);
Settings s = new Settings();
s.addOption(Settings.INIT_BELS, bels);
s.addOption(Settings.INIT_GOALS,
"jcm::focus_env_art([art_env(mining,m2view,default)],5)");
try {
String asl_file_name = "list/"+agent_id+".asl";

if (!Files.exists(Paths.get("src/agt/"+asl_file_name))){
asl_file_name = "miner3.asl";
}

System.out.println("asl_file_name: "+asl_file_name);

rs.createAgent(agent_id, asl_file_name, null, null, null, s, ts.getAg());
rs.startAgent(agent_id);
System.out.println("Agent created by custom file");
} catch (Exception e) {
e.printStackTrace();
}
}

```

(a)

```

String tuple_agent_id = String.valueOf(args[0]);
String tuple_data = "[" + String.valueOf(args[3]) +
" " + String.valueOf(args[4]) + " " + String.valueOf(args[5]) + "]";
String tuple_path = String.valueOf(args[1]) == "" ? "3" :
String.valueOf(args[1]).replace(" ","")+"-3";

System.out.println("Tuple - agent_id: "+tuple_agent_id);
System.out.println("Tuple - data: "+tuple_data);
System.out.println("Tuple - path: "+tuple_path);

if (rs.killAgent(tuple_agent_id, null, 0)){
System.out.println("Agent "+tuple_agent_id+" removed from the simulation...");

String postUrl = "http://"+host+":5000/api/v1/resources/model_to_router";
JSONObject json_obj = new JSONObject();
json_obj.put("agent_id",tuple_agent_id);
json_obj.put("data",tuple_data);
json_obj.put("path",tuple_path);

HttpClient httpClient = HttpClientBuilder.create().build();
HttpPost post = new HttpPost(postUrl);
StringEntity postingString = new StringEntity(json_obj.toString());
post.setEntity(postingString);
post.setHeader("Content-type", "application/json");
HttpResponse response = httpClient.execute(post);
} else {
System.out.println("Error while removing agent from simulation...");
}
}

```

(b)

```

+!check_new_agent : true
<-
.print("Checking and creating agent if it exists on API");
mylib.check_new_agent;
.wait(1000);
!check_new_agent.

```

(c)

```

+kill(V0, V1, X, B0, B1, B2) : true
<- .print("I've received a message to kill agent ",X);
.print("Killing agent ",X);
mylib.my_delete_ag(V0, V1, X, B0, B1, B2);
.wait(2000);
.print("This agent has being removed from the simulation: ",X).

```

(d)

Figure 5. Java and ASL functions inserted into JaCaMo code to provide the functions responsible for (a) receiving, (b) deleting, (c) checking, and (d) removing agents from/to the architecture

In the current adaptation of the original model, we use only one miner agent, along with the architecture's agents (*check_new_agents* and *killer_agent*) and the leader agent. So, when the agent joins the simulation, it takes control of the miner agent. This agent navigates through the map, brings back his old information (about previous simulations), navigates through the map, gets gold, brings it back to the depot, and then leaves the simulation (sends a message to the *killer_agent* to be removed from the simulation). Finally, when the agent leaves the simulation, the model checks the next agent that will join the simulation and takes control of the miner agent so the simulation can continue its execution.

Some adaptations are needed to adapt any JaCaMo model to our platform. First, if the project is not already running via Gradle (which is the default option), we need to adapt the model to do so. The steps required to achieve this goal are presented in JaCaMo's GitHub documentation. After, we need to include two libraries on the build Gradle file: JSON-simple (to deal with JSON-format information) and HTTP client (to deal with HTTP requests). The last step is to include the killer/checker agents (with their ASL files) who deal with the API. With these steps done, the model is ready to communicate with the API. Now, the programmer can use the system's functions to send and receive agents between the API and the model. We have examples of how to adapt and use our functions on GitHub.

4.3. The simulation scenario

In the simulation scenario, M_1 and M_2 containers are running simultaneously two replicas of the Sugar Scape NetLogo model, and the M_3 container is running the Gold Miners model, on JaCaMo. All three models run indefinitely and use the *auto-run* option when no trigger container starts the simulation. When an agent dies, the current model is responsible for sending it to the Router (and then to DB via API); it will go through a process that will decide which models the agent will return.

At the beginning of the execution, instead of creating the agents directly by the model, the models M_1 and M_2 ask the Register container to create the agents so that they receive an identification used in the OMAS. In the current implementation of the simulation scenario, only the NetLogo models ask for new agents. The JaCaMo model is receiving and sending agents but not creating them on the initialization, even though it is possible.

After the initial step of creating and registering all agents, the M_1 , M_2 and M_3 models continue their execution, along with the Router processing. Whenever an agent leaves any of the models, it goes to the DB container (via API). Once new information is on the DB, the Router knows that there are new agents to be processed, so it reads the agents and sends them to a model according to the routing type (by default, randomly). Both Log or Interface containers can access the flow of this information.

It is essential to point out that most adaptations are not substantial, mischaracterizing the original models to be adapted to the architecture; instead, they are due because these models need to be prepared to receive agents from outside, to be able to do so. We have provided a template of the trigger functions to send and receive information to the architecture. The users must add it to their model, adapt to what information they want to send or receive, and use the triggers when they want the model to send or receive agents. We emphasized that this feature is important because we aim to reduce the complexity of the code adaptation. On the documentation, the user can find all the information needed. Also, those adaptations show that the architecture allows completely different agent architectures to communicate. For example, NetLogo programming uses Logo language, while JaCaMo uses ASL on belief, desire, intention (BDI) concepts. Though heterogeneous models, the agent carries all its information while moving in the architecture. Thus, it is possible to adapt information when we create an agent on each model because we included/excluded from the models in real-time.

We provide videos and tutorials on the GitHub page [32] about the scenarios tested that were essential to check the architecture's strengths. With these tests, the platform allowed two MAS models, coming from the original documentation of each platform (NetLogo and JaCaMo), becoming from a closed to an open MAS execution with few extra steps. Furthermore, we could familiarly implement the original models. Instead, we had to insert the codes and the agents necessary to communicate with our platform. Finally, it is essential to point out that both platforms have access to all information from the agent. For example, JaCaMo's model could access any attribute from the NetLogo's agent (sugar, metabolism, and vision) to make something useful in the simulation. The opposite is true: NetLogo has access to the *.asl* file of every agent, making it possible to use some information, like a belief, for something useful in the simulation. The data exchange and the openness with feel steps are two key contributions of our work.

Both scenarios use similar parameters. The main difference is that one scenario runs both models inside Docker. In contrast, the other scenario runs JaCaMo outside Docker (but still communicates with the platform) to be able to use a GUI. When running JaCaMo outside of Docker, the API container must expose a port to establish communication with the host machine.

In the scenarios, after downloading the platform from documentation (available on GitHub), the programmer starts the simulation with the command *docker-compose up -d*. This command will tell Docker to download and build all images and containers necessary, according to the *docker-compose.yml* file. After everything is set and running, all container logs are accessible via Docker Desktop or Docker command line interface (CLI). While CLI is a Command Line Interface, Docker Desktop is a GUI that allows the user to check what is happening inside containers, run commands on containers, and so on. In addition, each container shows logs and information about itself. For instance, in the API container, it is possible to check all the requests that the API receives and processes, while in the JaCaMo container, the user can check all alive agents, minds, artifacts, and so on. Besides the Docker logs, the DBMS and Interface blocks are accessible via the browser (address on docker-file), allowing users to access all agents' navigation through the models and their parameter values.

5. CONCLUSION

This paper presented a proposal of architecture to assist the development of OMAS. Similar work to this proposal was summarized and discussed. Then, we introduced the proposed approach, methodology, and implementation details. Finally, we showed the results obtained, emphasizing the case of the study developed in the architecture prototype, which is of great importance to verify the feasibility of the implementation. Our approach presents an environment that facilitates the development of OMAS using Docker. This architecture allows the migration of agents between models that can run in heterogeneous scenarios. Furthermore, the structure provides code to run inside containers that contain the same operation structure where they were developed, avoiding problems like programs that run in the development stage but in the production stage have issues. In addition, the architecture acts as the union step between models. Docker allows models to be executed in different scenarios on various platforms, using distinct development languages that run on multiple operating systems. In the presented simulation scenario, we used containers running NetLogo's Open Sugarscape 2 Constant Growback and JaCaMo's Gold Miners with minor adaptations to validate the approach feasibility. The prototype presented allowed agents to move freely between models, sharing all agent information with respective models and reducing the complexity of the code adaptation, demonstrating that it is sufficient but

straightforward to understand the proposed approach better. The migration of agents between models occurs at runtime. The motivation for this migration of an agent from one system to another can be different, usually of the developer's choice, such as execution failures, self-will, or some trigger. In addition, we can deal with conflicts of interest between the new agents when designed to work outside of that model. In future work, we want to explore new triggers that make agents switch models, such as geographic boundaries and parallel models. Geographic boundaries are models where, when the agent reaches the edge of the environment, it passes to the other model. Parallel models are models where the same agent participates in more than one model simultaneously, but each model evolves particular attributes of the agent. Also, even though the platform runs on a local machine, several platforms (i.e., Amazon's AWS Docker) allow the entire structure being developed based on Docker to run in the cloud, enabling application portability and expanding the universe of applications in the architecture. Finally, the architecture implementation supports two relevant agent platforms, NetLogo and JaCaMo. However, we want to go further and support other agent platforms, such as JADE (Java-based) or Mesa (Python-based).

ACKNOWLEDGEMENTS

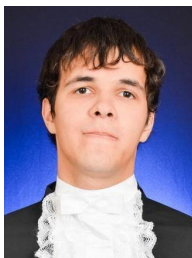
This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.




REFERENCES

- [1] A. Perles, F. Crasnier, and J.-P. Georgé, "Amak - a framework for developing robust and open adaptive multi-agent systems," in *Proceedings of International Conference on Practical Applications of Agents and Multi-Agent Systems – Highlights of Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection*, J. Bajo, J. M. Corchado, E. M. Navarro Martínez, E. Osaba Icedo, P. Mathieu, P. Hoffa-Dabrowska, E. del Val, S. Giroux, A. J. Castro, N. Sánchez-Pi, V. Julián, R. A. Silveira, A. Fernández, R. Unland, and R. Fuentes-Fernández, Eds., Cham: Springer International Publishing, 2018, pp. 468–479, doi: 10.1007/978-3-319-94779-2_40.
- [2] V. R. Lesser, "Cooperative multiagent systems: A personal view of the state of the art," *IEEE Transactions on knowledge and data engineering*, vol. 11, no. 1, pp. 133–142, 1999, doi: 10.1109/69.755622.
- [3] M. Wooldridge and M. J. Wooldridge, *Introduction to multiagent systems*. EUA: John Wiley & Sons, Inc., 2001, doi: 10.5555/1695886.
- [4] W. Jamroga, A. Meski, and M. Sreter, "Modularity and openness in modeling multi-agent systems," *Electronic Proceedings in Theoretical Computer Science*, vol. 119, pp. 224–239, Jul. 2013, doi: 10.4204/eptcs.119.19.
- [5] D. M. Uez, "Open aeolus: um método para especificação de sistemas multiagentes abertos," Ph.D. thesis, Automation and Systems Engineering, Federal University of Santa Catarina, Florianópolis, Brazil, 2018, [Online]. Available: <https://repositorio.ufsc.br/handle/123456789/205584>.
- [6] F. Dalpiaz, A. K. Chopra, P. Giorgini, and J. Mylopoulos, "Adaptation in open systems: Giving interaction its rightful place," in *Proceedings of the Conceptual Modeling – ER 2010*, J. Parsons, M. Saeki, P. Shoval, C. Woo, and Y. Wand, Eds., Berlin, Heidelberg: Springer, 2010, pp. 31–45, doi: 10.1007/978-3-642-16373-9_3.
- [7] T. D. Huynh, N. R. Jennings, and N. Shadbolt, "Developing an integrated trust and reputation model for open multi-agent systems," *Autonomous Agents and Multi-Agent Systems*, vol. 13, pp. 119–154, 2004, doi: 10.1007/s10458-005-6825-4.
- [8] S. Kaffille and G. Wirtz, "Modeling the static aspects of trust for open mas," in *2006 International Conference on Computational Intelligence for Modelling Control and Automation and International Conference on Intelligent Agents Web Technologies and International Commerce (CIMCA'06)*, 2006, pp. 186–186, doi: 10.1109/CIMCA.2006.150.
- [9] R. M. van Eijk, F. S. de Boer, W. Van Der Hoek, and J.-J. C. Meyer, "Open multi-agent systems: Agent communication and integration," in *Proceedings of the International Workshop on Agent Theories, Architectures, and Languages*, 1999, pp. 218–232, doi: 10.1007/10719619_16.
- [10] J. M. Hendrickx and S. Martin, "Open multi-agent systems: Gossiping with deterministic arrivals and departures," in *Proceedings of the 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2016, pp. 1094–1101, doi: 10.1109/ALLERTON.2016.7852357.
- [11] M. Franceschelli and P. Frasca, "Proportional dynamic consensus in open multi-agent systems," in *Proceedings of the IEEE Conference on Decision and Control (CDC)*, 2018, pp. 900–905, doi: 10.1109/CDC.2018.8619639.
- [12] Z. Houhamdi and B. Athamena, "Collaborative team construction in open multi-agents system," in *Proceedings of the 21st International Arab Conference on Information Technology (ACIT)*, 2020, pp. 1–7, doi: 10.1109/ACIT50332.2020.9300116.
- [13] S. Noh and J. Park, "System design for automation in multi-agent-based manufacturing systems," in *Proceedings of 20th International Conference on Control, Automation and Systems (ICCAS)*, 2020, pp. 986–990, doi: 10.23919/ICCAS0221.2020.9268357.
- [14] W. Jiang, Y. Chen, and T. Charalambous, "Consensus of general linear multi-agent systems with heterogeneous input and communication delays," *IEEE Control Systems Letters*, vol. 5, no. 3, pp. 851–856, 2021, doi: 10.1109/LCSYS.2020.3006452.
- [15] M. Franceschelli and P. Frasca, "Stability of open multiagent systems and applications to dynamic consensus," *IEEE Transactions on Automatic Control*, vol. 66, no. 5, pp. 2326–2331, 2021, doi: 10.1109/TAC.2020.3009364.
- [16] C. Anderson, "Docker [software engineering]," *Ieee Software*, vol. 32, no. 3, pp. 102–c3, 2015.
- [17] J. Turnbull, *The Docker Book: Containerization Is the New Virtualization*, Melbourne, Australia: James Turnbull, 2014.
- [18] Y. Demazeau and A. R. Costa, "Populations and organizations in open multi-agent systems," in *Proceedings of the 1st National Symposium on Parallel and Distributed AI (PDAI'96)*, 1996, pp. 1–13.




- [19] J. Gonzalez-Palacios and M. Luck, "Towards compliance of agents in open multi-agent systems," in *Proceedings of the International Workshop on Software Engineering for Large-Scale Multi-agent Systems*, Shanghai, China: Springer, 2006, pp. 132–147, doi: 10.1007/978-3-540-73131-3_8.
- [20] A. Artikis, "Dynamic specification of open agent systems," *Journal of Logic and Computation*, vol. 22, no. 6, pp. 1301–1334, 2011, doi: 10.1093/logcom/exr018.
- [21] S. Paurobally, J. Cunningham, and N. R. Jennings, "Ensuring consistency in the joint beliefs of interacting agents," in *Proceedings of 2nd International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS*, 2003, pp. 662–669, doi: 10.1145/860575.860682.
- [22] M. P. Singh and A. K. Chopra, "Programming multiagent systems without programming agents," in *Proceedings of the International Workshop on Programming Multi-Agent Systems*, 2009, pp. 1–14, doi: 10.1007/978-3-642-14843-9_1.
- [23] S. Hattab and W. Lejouad Chaari, "A generic model for representing openness in multi-agent systems," *The Knowledge Engineering Review*, vol. 36, p. e3, 2021, doi: 10.1017/S0269888920000429.
- [24] W. A. L. Ramirez and M. Fasli, "Integrating netlogo and jason: a disaster-rescue simulation," in *2017 9th Computer Science and Electronic Engineering (CEECE)*, 2017, pp. 213–218, doi: 10.1109/CEECE.2017.8101627.
- [25] S. Dähling, L. Razik, and A. Monti, "Enabling scalable and fault-tolerant multi-agent systems by utilizing cloud-native computing," *Autonomous Agents and Multi-Agent Systems*, vol. 35, no. 1, pp. 1–27, 2021, doi: 10.1007/s10458-020-09489-0.
- [26] V. Pfeifer, W. F. Passini, W. F. Dorante, I. R. Guilherme, and F. J. Affonso, "A multi-agent approach to monitor and manage container-based distributed systems," *IEEE Latin America Transactions*, vol. 20, no. 1, pp. 82–91, 2021, doi: 10.1109/TLA.2022.9662176.
- [27] M. Grinberg, *Flask web development: developing web applications with python*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2018, doi: 10.5555/2621997.
- [28] S. Tisue and U. Wilensky, "Netlogo: A simple environment for modeling complexity," in *International conference on complex systems*, vol. 21, pp. 16–21, 2004.
- [29] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "Multi-agent oriented programming with jacamo," *Science of Computer Programming*, vol. 78, no. 6, pp. 747–761, 2013, doi: 10.1016/j.scico.2011.10.004.
- [30] J. M. Epstein and R. L. Axtell, *Growing artificial societies: Social Science from the Bottom Up*, Cambridge, MA, USA: Bradford Books, Oct. 1996.
- [31] R. H. Bordini, J. F. Hübner, and D. M. Tralamazza, "Using jason to implement a team of gold miners," in *International Workshop on Computational Logic in Multi-Agent Systems*, 2006, pp. 304–313, doi: 10.1007/978-3-540-69619-3_18.
- [32] G. L. de Lima and M. S. de Aguiar, "Architecture's github page," 2022, [Online]. Available: https://github.com/GustavoLLima/open_mas_docker_opt (accessed: October 26, 2022).

BIOGRAPHIES OF AUTHORS



Gustavo Lameirão de Lima    is a Ph.D. student in Computing at the Federal University of Pelotas - UFPEL. In addition, he is a Master in Computer Engineering at the Federal University of Rio Grande - FURG (2018), Electronics Technician at the Instituto Federal Sul Rio Grandense - IFSUL (2013) and Technologist in Internet System Technology at the Instituto Sul Rio Grandense IFSUL (2013). 2016). He can be contacted at email: gustavolameirao@inf.ufpel.edu.br or gustavolameirao@gmail.com.



Marilton Sanchotene de Aguiar    Associate Professor at the Federal University of Pelotas (UFPEL) of the Undergraduate Courses in Computer Science and Engineering and the Graduate Program in Computing. Director of the Center for Technological Development at the UFPEL. His research has focused on applications of Artificial Intelligence to educational, environmental, and health problems. He can be contacted at email: marilton@inf.ufpel.edu.br.