LI(A)RA Team - A Declarative and Distributed Implementation for the MAPC 2022

Marcelo Custódio, Michele Rocha, Ricardo Battaglin, Giovani P. Farias, and Alison R. Panisson(✉)

Department of Computing (DEC), Federal University of Santa Catarina (UFSC), Araranguá, Brazil
alison.panisson@ufsc.br

Abstract. With the increasing computational power and the evolution of the distributed computing paradigm and artificial intelligence techniques and methodologies, computing is becoming increasingly distributed and intelligent. The multi-agent paradigm is one of the most powerful paradigms for implementing distributed artificial intelligence. There are multiple (multi-)agent-oriented programming languages, platforms and methodologies to implement these systems, and the Multi-Agent Programming Contest aims to provide challenging scenarios in which researchers can explore their preferred programming languages, platforms and methodologies to implement multi-agent systems, collecting benchmarks, etc. In this paper, we describe the multi-agent system implemented by the LI(A)RA team for the Multi-Agent Programming Contest 2022, including details about the system's implementation, the technologies and methodologies used, and also discuss the team's results.

Keywords: Artificial Intelligence · Multi-Agent Systems · Agent-Oriented Programming Languages · Multi-Agent Programming Contest

1 Introduction

The multi-agent systems paradigm emphasises the thinking of systems as multiple intelligent entities. This paradigm is becoming very popular, not only because of the rise of the use of artificial intelligence techniques but also because it naturally meets the current demand to design and implement distributed intelligent systems such as smart homes, smart cities, and personal assistants, for instance, also facilitating the integration between those systems. Nowadays, we are able to argue that multi-agent systems are one of the most powerful paradigms for implementing complex distributed systems powered by artificial intelligence techniques [11], for example, incorporating argumentation-based reasoning and communication [22,24], modelling and reasoning about uncertain information and theory of mind [27,31], and many other techniques.

T. Ahlbrecht et al. (Eds.): MAPC 2022, LNAI 13997, pp. 165–194, 2023.
https://doi.org/10.1007/978-3-031-38712-8_7

Multi-agent systems are built upon core concepts such as distribution, reactivity, and individual rationality. To support the development of multi-agent systems, a large number of tools have been developed, such as agent-oriented programming languages and methodologies [4]. Consequently, practical applications of multi-agent technologies have become a reality, many of them solving complex and distributed problems [10,23,32,33]. In addition, it also allows the execution of various tasks and makes it possible the integration with various technologies, for example, chatbot technologies [8,9].

The Multi-Agent Programming Contest (MAPC) was created with the aim of exploring the potential of multi-agent systems, providing challenging scenarios to explore agent-oriented programming languages, platforms and methodologies for developing multi-agent systems. Also, MAPC provides reference problems in which different researchers can compare their results, such as benchmarks. MAPC 2022 brought the Agents Assemble III scenario, including a normative system, the idea of agents roles, and complex tasks agents should exhibit attitudes of collaboration and coordination in order to score points. In this paper, we describe the solution proposed by the LI(A)RA team for the MAPC 2022 scenario. LI(A)RA is a project, created in 2022, for teaching agent technologies for undergraduate students at the Federal University of Santa Catarina. The project also aims to research the software engineering process behind participating in the contest. The proposed solution was implemented using Jason Platform [5], focusing on (i) a purely declarative implementation, which aligns with the purpose of the original language, and (ii) distributed mechanisms for coordination and collaboration.

This paper is organised as follows. First, in Sect. 2 we describe the Agents Assemble III scenario used during the Multi-Agent Programming Contest 2022, pointing out the differences between this year's scenario and the previous ones, also highlighting the main challenges in this new scenario according to our point of view. In Sect. 3, we describe the implementation of LI(A)RA's solution for the MAPC 2022 scenario, including the methodology applied during development and the agents' strategies implemented according to different aspects of the Agents Assemble III scenario, for example, movement strategies, synchronisation strategies, among others. In Sect. 4, we describe our results in the MAPC 2022 contest. Finally, in Sect. 5, we present our conclusions, pointing out some future directions our team intend to adopt.

2 The Agents Assemble III Scenario

In 2022, the Multi-Agent Programming Contest brought a revision from the scenario presented in the previous MAPC 2019[1] and 2020/2021[2], named the Agents Assemble III. The main difference regarding the previous year's scenario was: (i) considering roles for agents and establishing different capabilities for agents according to their roles. Also introducing role zones, which are places

[1] https://multiagentcontest.org/2019/.
[2] https://multiagentcontest.org/2020/.

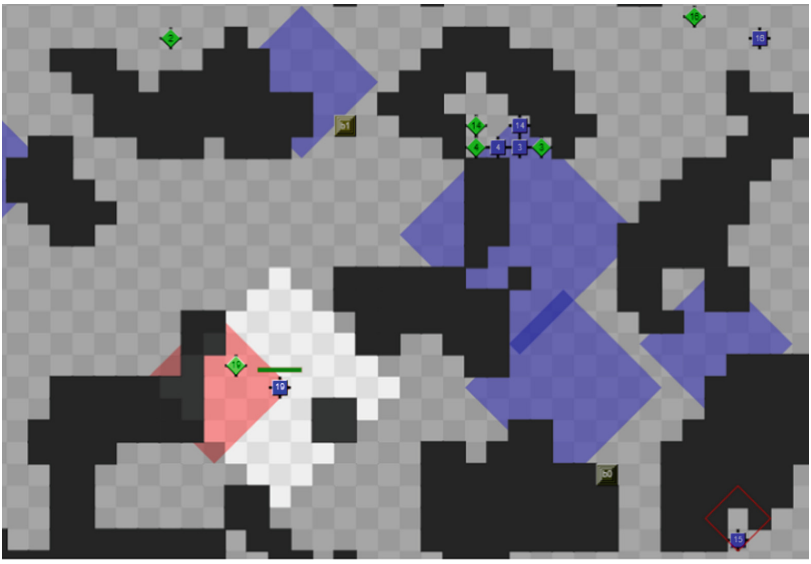

Fig. 1. Example of the Agent Assemble World. (Color figure online)

in the environment agent could adopt and change roles; and (ii) a normative system in which agents would be punished according to they violate the norms established during the simulation.

In the Agents Assemble III scenario, agents are situated in a grid world environment with limited local vision, and they are required to organise and coordinate themselves to assemble and deliver complex structures made of blocks, which are called "tasks". In Fig. 1, we can observe an example of an instance of the Agents Assemble III World, in which we are able to observe the following aspects:

1. blue numbered small squares and green numbered small lozenges represent agents, in which agents with the same colour and format are from the same team. All matches are between two teams, which is why there are two teams[3] in the grid world in Fig. 1.
2. large blue and red translucent lozenges represent role and goal zones, respectively. Role zones (large blue translucent lozenges) are regions in which an agent is able to change its role, choosing among the different roles available during a particular match, which will enable it to execute particular actions during the simulation. Goal zones (large red translucent lozenges) are regions in which agents are able to deliver tasks. Agents are required to be at a goal zone in order to deliver tasks, also respecting the specificity of each task (form of the structure made of blocks, relative position of the agent delivering the tasks, etc.).

[3] Future versions of the contest may allow matches among more than 2 teams.

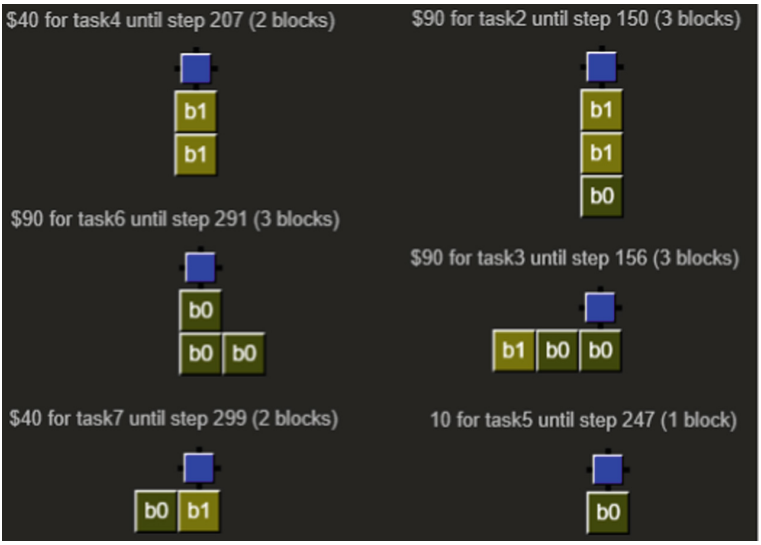

Fig. 2. Examples of Tasks.

3. black blocks are obstacles that agents cannot go through, they can deviate or clear[4] obstacles in order to move through the grid. For example, blue agent number 15 (at the bottom left in Fig. 1) is clearing an obstacle that was north of it (clear actions plot red outline lozenge in the environment).
4. blocks with different shades of yellow represent the dispensers. Dispensers can be used by agents to request blocks. When an agent requests a block from a dispenser, the dispenser generates a block over the dispenser, and the block becomes available to agents to attach it. Dispensers only generate blocks of their type, for example, a dispenser of the type "b1" only generates blocks of the type "b1". Figure 1 shows two different dispensers, one of the type "b1" and one of the type "b0".
5. finally, we also are able to observe, in Fig. 1, agent 19's range of vision, highlighted by the lighter region around it. Agents have a limited local vision, and an agent is able to observe only those things within its range of vision. For example, agent 19 is perceiving the adversary agent 19, the goal zone, and obstacles within its range of vision.

There are many challenges associated with the MAPC 2022 scenario, some of them already known from previous contests, and new challenges consequence of the addition of roles and normative specifications. Below we emphasise the most challenging aspects of the MAPC 2022 scenario according to our point of view.

– **The absence of absolute position:** one of the most challenging aspects of the MAPC 2022 (also 2019 and 2020/2021) scenario is the absence of abso-

[4] Clear actions are only available for some roles agents can adopt during the simulation.

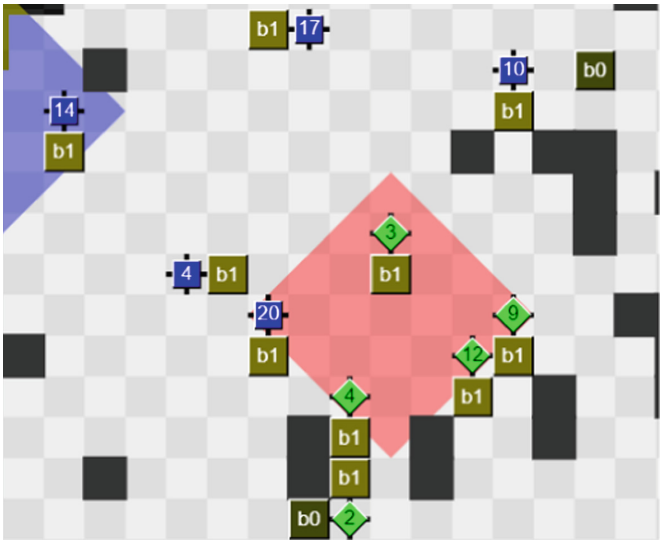

Fig. 3. Example of a Crowded Goal Zone.

lute position. It means, all agents start in a different position on the computational grid which implements the environment, and all of them understand they are at the coordinates $x = 0$ and $y = 0$ at the beginning of the match. Consequently, when agents reach important regions of the world (for example, goal zones and dispensers), the perspective of those things' position is different from each other. For example, the dispenser "b1" in Fig. 1 is at `position(3,-12)` ($x = 3$ and $y = -12$) for blue agent 19, and at `position(-7,-1)` for the blue agent 4 (considering that Fig. 1 shows the configuration of the match at the start point, in which agents understand they are at the coordinates $x = 0$ and $y = 0$).

- **The uncertainty of success when executing actions:** another challenging aspect of the MAPC 2022 scenario is that agents randomly fail to execute their actions. It means, an agent only knows if its actions had the expected result after executing the actions and perceiving its success during the next step of the environment simulation. For example, when an agent executes the action to move north, it is expected the agent to leave its current position, let's say `position(10,10)`, and ends at the north of it, i.e., at `position(10,9)`. However, at the current configuration of the scenario, the agent only knows if it actually moved after, when it perceives the next step of the simulation (it requires the agent to perceive if it had succeeded in executing that action), perceiving a change in its position.
- **The complex tasks:** another challenging aspect of the MAPC 2022 scenario are the complex tasks, ranging from task formed by 1 to 4 blocks in different configurations. Figure 2 shows some examples of tasks that appeared during a match. Not only do agents need to cooperate and coordinate to build and

deliver the tasks formed by more than one block, but also it requires delivering the tasks in specific regions of the environment called goal zones. Goal zones were scarce, they became crowded during most of the matches, and they also became regions of interest by other teams that applied aggressive strategies in which they attacked other agents (by executing clear actions at other agents and their blocks) trying to deliver tasks.
- **The crowded goal zones:** another challenging aspect of the MAPC 2022 scenario is the scarce goal zones (sometimes very small ones). This aspect of the simulation was problematic in finding space to build and deliver the complex tasks, given that agents should find the appropriate space between other agents to build and deliver the task on those zones. Figure 3 shows an example of a crowded goal zone, in which the LI(A)RA (green) agents 4 and 2 are delivering a task of two blocks, and agents 3, 9 and 12 are waiting for other agents to build the complex task and deliver it. Also, Fig. 3 shows 4 agents from another team also approaching the goal zone to deliver tasks.
- **The random clear events:** another challenging aspect of the MAPC 2022 scenario is the random clear events that occur randomly in the world. They are challenging aspects because they basically reset agents when they occur over agents, taking the energy from the agents and (mostly of the time) destroying the blocks they were carrying.
- **Selecting one action by step:** an aspect of the MAPC 2022 scenario that changes how to implement the system is that agents are able to execute only one action by step of the simulation, then, if the simulation has 600 steps, agents will execute at most 600 actions during the simulation. While it makes the matches fair in the sense all agents from all teams will execute at most the same number of actions, it also changes how we implement agents, being necessary to think about a strategy in which agents will choose one action for each step, sending the action at the correct time, i.e., sending their 30th action between the 30th and 31st step of the simulation.

3 Implementation

3.1 Technology

To implement the LI(A)RA team's solution for the Multi-Agent Programming Contest 2022, we have used the Jason platform [5]. Jason extends the AgentSpeak(L), an abstract logic-based agent-oriented programming language introduced by Rao [30], which is one of the best-known languages inspired by the BDI (*Beliefs-Desires-Intentions*) architecture [6], one of the most studied architectures for cognitive agents. Also, Jason is part of the JaCaMo Framework [3], which allows us to implement complex multi-agent systems encompassing all dimensions necessary, named agents, environment, and organisation.

In Jason, besides agents are implemented based on the BDI architecture [6], they also are defined with a plan library that provides the know-how for agents, inspired by the procedural reasoning system (PRS) [14], providing an architecture that combines practical reasoning and planning, in which agents are able

to handle challenging tasks in a dynamic environment. The agents' knowledge is defined through beliefs and, as usual, the knowledge available for agents may not necessarily be complete or accurate, considering the environment may be large and may change the agent has not perceived.

In Jason, beliefs and goals are represented by predicates and a set of n terms of first-order logic, as follows:

```
predicate(term_1,term_2, ..., term_n)
```

For example, the triggering event `role(worker)` is composed of the predicate '`role`' and the term '`worker`', meaning the agent has perceived it is playing a role named worker.

Furthermore, predicates can be annotated with meta-information related to that information, as introduced in [5] also used by others [16,17,21]. The syntax for annotated predicates is as follows:

```
predicate(term_1,term_2, ..., term_n)[ann_1,ann_2,...,ann_n]
```

where each `ann_i` represents the SPSVERBc5th annotation for that particular predicate, with the following syntax:

```
functor(term'_1,term'_2, ..., term'_n)
```

where an atom (called functor) is followed by a number of terms (called arguments). This extension of the language provides more expressiveness, as pointed out by [21] in the context of argumentation. A common meta-information originally used in Jason Platform [5] is the source of information, for example:

```
likes(john,icecream)[source(mary)]
```

described that `mary` has said that `john` likes ice cream, i.e., `mary` is the source of the information `likes(john,icecream)`.

In Jason, agents are able to represent the information they believe to be true, for example, `likes(john,icecream)`, and information they believe to be false, for example, ¬`likes(john,icecream)`, and, using negation as failure, they are able to query information they have no knowledge about, for example, `not(likes(john,icecream))` and `not(`¬`likes(john,icecream))`, representing that the agent does not know if `likes(john,icecream)` is true or false.

Plans are composed of a triggering event, a context, and the body of the plan, which represent a recipe (set of ordered actions and sub-goals) to achieve that particular goal. The body of a plan may include updates to the belief base, actions and (sub)goals. Triggering events are used to react to the addition (or deletion) of beliefs or goals. The context establishes the precondition for the plan, defining what must be true in order for the plan to be executed. The following example shows the abstract syntax of Jason plans:

```
triggering_event : context <- body.
```

For example, agents would be able to react to the perception of a role zone, creating a goal to move to that particular role zone using the following plan:

```
+roleZone(X,Y)[source(percept)] :
        not(movintToRoleZone(_,_)) &
        my_expected_role(MyRole) &
        not(role(MyRole))
    <-
        +movingToRoleZone(X,Y);
        !moveTo(X,Y,rolezone).
```

in this particular plan, the `context` for executing this plan requires that the agent is not moving to any role zone, i.e., `not(movintToRoleZone(_,_))`, and it is playing a role during the simulation that is different from the expected role it should be playing (given the strategy adopted by our team), i.e., `my_expected_role(MyRole) & not(role(MyRole))`.

In the agent's plan library, there may be several plans to react to the same triggering event corresponding to a goal, an external event, etc. and they represent choices an agent can make to achieve their goals. While sophisticated plan selection functions can be implemented, originally, in the Jason Platform, plans are analysed using the order they have been declared in the plan library of an agent, similar to architecture with vertically layered priorities [18].

For example, the piece of code below contains 4 plans that implement 4 different ways an agent may achieve the goal `!moveTo(X,Y)`, implementing a very simple movement strategy that could be used in MAPC 2022 scenario. When an agent creates a goal, for example, `!moveTo(10,20)`, it will verify which plans could be used to achieve that particular goal, i.e., which plans apply at that current moment of its execution. Then, the agent will use the first plan which applies to try to achieve its goal. Considering a particular scenario in which the agent is at `position(5,10)`, then there are two plans the agent could use to achieve the goal `moveTo(10,20)`, named `plan1` and `plan3`. However, considering the original implementation for the plan selection, the agent will prioritise always horizontal movement over vertical movement, i.e., going east first, executing the action `move(e)`, until reaching `position(10,10)`, then the first plan will not be applicable anymore, and the only applicable plan will be the `plan3`, when the agent will go south, executing the action `move(s)`, until reaching the position `position(10,20)`, achieving its goal.

This plan selection mechanism specifies a very important aspect of the technology, which must be considered in order to adequately implement strategies. For example, the example below shows a very simple movement strategy, and it could implement a different movement strategy by just reordering plans 3 and 4 first, prioritising vertical movement over horizontal ones.

```
+!moveTo(X,Y): position(XMy,YMy) & XMy < X <- move(e). //plan1
+!moveTo(X,Y): position(XMy,YMy) & XMy > X <- move(w). //plan2
+!moveTo(X,Y): position(XMy,YMy) & YMy < Y <- move(s). //plan3
+!moveTo(X,Y): position(XMy,YMy) & YMy > Y <- move(n). //plan4
```

Another important aspect of a multi-agent system is communication. In Jason platform, agents are able to communicate using already implemented internal actions with the following format:

```
.send(receiver,performative,content)
```

in which `receiver` is the agent (or set of agents) that will receive that particular message, `performative` indicates the performative used in that particular message, which will provide the intention behind that communication, providing meaning for that communication together with the `content` of that message. For example, an agent named `ag_11` is able to tell other agents about a role zone it found during its execution using the following message:

```
.send([ag_1,ag_2,ag_3],tell,roleZone(10,20))
```

in which the agent sends a message to agents `ag_1`,`ag_2` and `ag_3`, telling that it has found a role zone at coordinate `(10,20)`, i.e., `roleZone(10,20)`. All agents will receive that information and they will believe that `roleZone(10,20)[source(ag_11)]`. Although Jason platform provides a set of predefined performatives with well-defined semantics [35] based on the KQML [12], other performatives can be easily added, extending those already available, for example, to allow sophisticated dialogues based on argumentation [25,26,28,29].

Furthermore, an agent program can be implemented in different modules/files and integrated into a single agent, each module providing part of its knowledge and capabilities [15,20], i.e., part of its beliefs and plans, respectively. Although there are many sophisticated manners to approach modules of agent programs, as pointed out by [20], we use a very simple approach in which agent knowledge and plans can be implemented in different files and after that integrated into an agent program. For example, imagine we implemented strategies for moving, attacking, and exploring the MAPC scenario in different files named: `move.asl`, `attack.asl` and `explore.asl`. Then, in order to have an agent that integrates the strategies for moving[5] and exploring, we only need to include both files as part of the agent program as follows:

```
{ include("move.asl")    }
{ include("explore.asl") }
```

In case we would have an agent that integrates the strategies for exploring and attacking, we would include the correspondents files as part of the agent program as follow:

```
{ include("explore.asl") }
{ include("attack.asl")  }
```

[5] Note that we have multiples implementations for the movement strategies, in which an agent is able to create a goal to move to a specific location, `!moveTo(10,20)`, but also referring what it is expected to find there if necessary, for example, a role zone `!moveTo(10,20,rolezone)`.

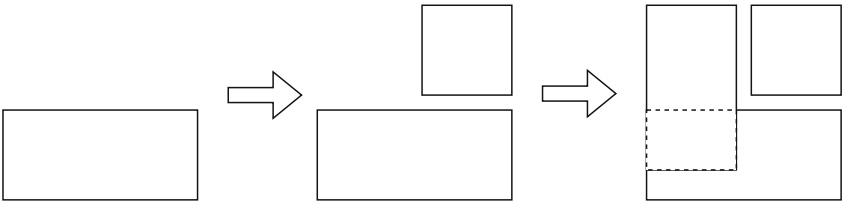

Fig. 4. The Incremental Approach.

3.2 Methodology

The multi-agent system was implemented using an *incremental approach*. Incremental approaches, also called *evolutionary approaches*, are widely acknowledged in the literature and they arise from the need for flexibility in the process of development. The incremental approach consists of incremental developments, where parts of the software are postponed in order to produce some useful set of functions earlier in the development project [7]. The basic idea in the incremental approach is to expand increments of an operational software product [2]. That means, each stage of development, is not intended to produce a complete system, but to produce multiple versions of the system, in which each new version adds new functionalities. Figure 4 shows a generic representation for the incremental approach, in which new "*modules*" are added to the system.

There are many specific models that may be accommodated under the incremental approach. One of the most interesting is the *Extreme Programming* (XP) [1], which normally is used in projects with uncertainty or changing requirements, and it is an example of *agile approaches* [13], which aim at supporting changes and rapid feedback during software development.

We found this approach very adequate for our team, given the technology used, as described in Sect. 3.1, and the methodology used to implement our solution, in which all members implemented parts of the system, aligning the project during a weekly meeting. Also, XP requires comprehensive documentation, which was a goal for our team.

It is important to note that an incremental approach allows us to think about the system in a more modular way, in which modules of behaviour/capabilities can be implemented and tested, individually and integrated. In particular, we have implemented the modules presented in Table 1, implementing different strategies used during the MAPC 2022, which we will discuss in the next section.

A challenge for an incremental and modular approach to development is the documentation and standardisation of code. While the documentation was basically made towards commentating on the code, all developers shared a table of predicates used during implementation, presented in Tables 4, 5, and 6. One of the benefits of using a declarative programming language is that the code provides most of the semantics necessary to understand it, but even though, the developers have shared those tables containing a short description of all predicates used in the agents' code.

Table 1. Modules Implemented.

Module Name	Description
“move.asl”	movement strategies
“memory_updates.asl”	memory update strategies
“strategy.asl”	decision-making strategies
“collect_blocks.asl”	strategies for collecting blocks
“adopt_role.asl”	strategies for adopting roles
“exploration.asl”	exploration strategies
“complete_task.asl”	strategies for completing tasks
“synchronism.asl”	strategies for synchronising agents
“task_delivery_organization.asl”	strategy for task organisation
“connect_and_deliver.asl”	strategies for connecting and delivering tasks
“after_event.asl”	strategies for dealing with effects caused by events
“change_round.asl”	strategies for changing the round

This approach of sharing the used predicates also is important when programming different modules, even from the perspective of a single developer, because predicates declared, added, or defined in a particular module may be necessary for other modules. For example, the module named “`strategy`” uses predicates declared all over the other modules, most of them basically keeping the information of what that particular agent is doing in that particular step of the simulation, corresponding to the general strategy for our agents we will describe in the next section.

3.3 Strategies

An important aspect of our implementation, which also defines most of how we implemented our strategies, is that all agents are instances of the same code. On the one hand, this aspect of the implementation makes it more difficult to think of a solution. On the other hand, it explores the implementation of agents that show characteristics of adaptability, autonomy, and flexibility. Also, it is aligned with the characteristics of the MAPC 2022 environment, in which agents are able to execute one action by step of the simulation, deciding which action to execute based on the information they perceived during the previous steps. That means agents will execute a new action towards reacting to the perception of a new step in the simulation, deciding what to do based on their own execution (considering all information it acquires) during the match.

Considering both aspects of the scenario and implementation, we adopt a strategy in which agents react to the perception of each step of the simulation than reasoning about what they should do based on the current state of the environment around each agent and its previous actions and perceptions, memorising what they are doing at that particular step, and then executing the selected action in the environment. When they perceive the next step of the

simulation, they remember what they were doing in the previous step, reasoning about the next action to execute according to that information.

In our strategy, agents may reach a state in which they remember they are doing concurrent activities, for example, the activities of *exploring* and *helping*, indicating they remember to be *exploring* the environment and also *helping* other agents to complete a task in the previous steps of the simulation. To deal with concurrent goals, we implemented a plan library with priorities based on the memory of agents. That means, for example, agents should always prioritise finishing a task in which they are helping other agents than exploring. The piece of code below shows an example of how this strategy is implemented:

```
+!step(S): helping(A,T,B,X1,Y1,X2,Y2,P) <- ....  //priority 1
+!step(S): collectingBlocks(X,Y,P)      <- ....  //priority 2
                .
                .
+!step(S): exploring  <- ....                    //priority n
```

In the example above, we show part of the agents' plan library with n plans, in which the agent will prefer to select plan 1 than plan 2 (if plan 1's context applies). In this example, agents prioritise *helping* other agents than collecting blocks, and *exploring* has the lower priority in the example above.

In the MAPC 2022 scenario, there is information that is worth agents remembering and information that we believed not to be worth agents remembering, given the dynamics of the environment. For example, in our implementation, agents remember the position of important components, such as role zones, goal zones, and dispensers. Even though the goal zones disappeared eventually, they were less dynamic than obstacles and blocks. We choose to implement agents that will not remember the position of those things with a higher probability of disappearing or changing position. Further, it is the memory (its beliefs) that make agents, which are all instances of the same code, show different behaviour. This is because those memories will enable the context of other plans, with higher priority, in the agent plan library.

For example, all agents start with the goal of finding a role zone to change their roles according to our strategy, which will enable them to execute a set of predefined actions in the environment of the contest. When an agent knows the position of a role zone, it will move towards that zone, otherwise, it will explore the environment to find a role zone, and then move towards the zone to change its role. The piece of code below shows an example of this strategy:

```
//plan1
+!step(S): my_role(R) & not(role(R)) & roleZone(X,Y)[source(memory)]
        <- !moveTo(X,Y).

//plan2
+!step(S): my_role(R) & not(role(R))  <- !explore.
```

In the example above, first, the agent tries to execute the `plan1`, but if it does not know the position of a role zone then that plan does not apply, i.e.,

it does not have any valid unification for `roleZone(X,Y)[source(memory)]`. In the case `plan1` does not apply, the agent tries to execute the `plan2` (if that context applies), in which the agent creates a goal for exploring the environment, i.e., `!explore`. When that agent finds a roles zone, exploring the environment, it will memorise the information of the coordinates of the role zone it found, i.e., `roleZone(X,Y)[source(memory)]`, then the context of the `plan1` will apply, and then the agent will select that plan during the next step of the simulation, starting to move towards that role zone using that plan, i.e., creating the goal `!moveTo(X,Y)` in which `X` and `Y` are the coordinates to the role zone.

A strategy for implementing the proposed solution, adopted by our team, was implementing plans from lower priority to higher priority, in which higher priority plans have a dependence on beliefs (memories of what the agent is currently doing) achieved by the complete execution of lower priority plans. That means the context of plans with higher priority depends on the memories that only will be obtained during the execution of plans with lower priority. In the example above, the memory `roleZone(X,Y)[source(memory)]` will be obtained by executing the `plan2`, in particular, the plan to achieve the sub-goal `!explore`. Then, after knowing that information, the context of `plan1` will apply and that will be the selected plan in the next step of the simulation, given it has priority over the `plan2`.

Using this development strategy based on the priority of plans, we implemented specific strategies for different activities agents should execute during the matches, among them: **movement and exploration strategies** and a **strategy for dealing with norms**, a **synchronisation strategy based on encounters**, a **strategy for sharing information**, a **strategy for creating groups of agents**, and a **strategy for delivering tasks**. We discuss them below.

Movement and Exploration Strategies. During some test matches, we observed that obstacles, in general, were very disturbing for agents trying to move. Also, the proportion of obstacles increased according to the simulation changed from one round to another. Considering that obstacles should be disturbing for agents from other teams too, we focused on implementing a movement strategy that cleared as less obstacles as possible (different from most of the other teams, from this and previous years of the contest), keeping the obstacles on the grid world to disturb other agents trying to move on the environment.

The first aspect of the movement strategy implemented was the **exploration**, in which agents should explore the environment until being able to do something useful for the team, for example, carrying blocks and helping others to build complex tasks. At the beginning of a match, agents start to explore a particular direction, for example, north – `exploring(n)` – and, aiming to avoid clearing obstacles, they deviate obstacles going to side directions prioritising clockwise, i.e., in case an agent is exploring to the north and there is an obstacle at the north of agent, but there is no obstacle at east, then it would go east until be able to go north again. In case there is an obstacle at the north and east of the agent,

then it goes west (only if didn't come from that direction in the previous step of the simulation). Otherwise, when the agent is blocked by obstacles at the north, east, and west, then the agent executes the clear action targeting the obstacles in the north, allowing it to move to the north. This exploration strategy avoids clearing so many obstacles while also avoids agents to move in circles (or even being trapped at some small portion of the grid world).

A complexity to movement strategies, in the scenario of MAPC 2022, is related to agents moving while carrying blocks. As a coherent simulation, the scenario establishes that obstacles block agents either for obstructing the agent trying to move directly or for obstructing blocks attached to agents trying to move. That means, when agents are moving with blocks attached to them, they have to worry about obstacles obstructing not only their path but also the blocks' path to which they are attached. For example, if the agent has a block attached to the east of it, and the agent is trying to move north, then the agent needs a clear path both north of it and north of the block. However, if the agent has a block attached south of it, and it is trying to move north, then the agent does not need to worry about obstacles blocking the block.

Considering the complexity of moving with blocks, our team thought about two strategies (i) first, we believe to be the more common, is to use the same strategy for exploration, but also clearing obstacles obstructing blocks attached to the agent in order to move; and (ii) second, which we end using, is to rotate the block attached to the agent to the opposite direction the agent is moving, i.e., if the agent is moving north, it rotates the block to the south. The second alternative was more attractive to our team because it aligns with our strategy of clearing fewer obstacles as possible. Also, this strategy of moving with blocks attached to the agent in the opposite direction it is moving also become a very elegant **movement strategy**. Figure 5 shows an example of an agent using this strategy, in which agent 2 approached the dispenser `b1`, collected a block and it is moving away without clearing many obstacles. It makes hard, for other agents, to approach the dispenser than when the agents clear all obstacles on the way, because the dispenser will still be surrounded by obstacles.

Strategies for Dealing with Norms. Analysing the phenoms related to norms, we observed that by limiting agents to carry at most 1 block, we avoid most of the penalties applied when breaking the norms. Also, this strategy aligns with the movement strategy that works only for agents carrying a unique block.

Synchronisation Strategy Based on Encounters. An important aspect for agents to work together in the scenario proposed in the MAPC 2022 was the need for synchronisation of agents, regarding their coordinates. This issue comes from the characteristic of the scenario in which there is no absolute position in the grid world from the perspective of agents. That means, agents start the simulation at different positions on the grid and each agent believes it started at position zero – `position(0,0)`. That means, when agents move and find important elements on the grid, for example, dispensers of blocks, they will have

Fig. 5. Example of movement clearing few obstacles to collect a block.

a different perspective of the position of those elements. In order to coordinate tasks, cooperate and share information it was necessary to implement a strategy for synchronisation.

There would be centralised solutions for synchronisation, but in order to keep our implementation completely distributed, we implement a synchronisation strategy focusing on encounters. That means, when agents encounter each other in the grid world, they are able to communicate and synchronise their relative positions, creating filters for the position of those teammates they encountered during a match. We also used this idea of encounter to create groups of agents, which are dynamic and the process depends on these encounters.

For example, in Fig. 1, agents 3 and 4 are inside each other range of vision, which means they can perceive each other. This is what we call an encounter – when both agents are inside each other range of vision.

The synchronisation strategy occurs as follows: when an agent perceives another agent inside its range of vision, it executes a broadcast message, informing it can perceive the teammate at the coordinates `(XMate,YMate)` regarding its current position `(XMy,YMy)` during the step `S` of the simulation, also keeping a believe `found_mate(XMate,YMate,XMy,YMy,S)` with that information. At the next step of the simulation, `S+1`, the agent verifies if there is any match from the broadcasts it receives from other agents and those teammates found and stored by itself, comparing the relative position as follows:

```
found_mate(XO,YO,XOA,YOA,S)[source(TeamMate)] &
found_mate(XF,YF,XMA,YMA,S)[source(memory)] &
((XF+XO) == 0 & (YF+YO) == 0)
```

in which `XO, YO, XF` and `YF` are the position they found each other, thus if the difference between those values is equal to zero, that means the agent `TeamMate` is the same agent found by it at that step of the simulation. By understanding whom it has found, the agent is able to create a filter using the following equation:

Fig. 6. Example of an encounter.

```
mate_filter(TeamMate,((XF+XMA)-XOA),((YF+YMA)-YOA))[source(memory)].
```

in which it calculates the relative position of the other agent regarding its position.

Note that there are two different pieces of information they are able to infer from this communication process: (i) when agents enter inside the range of vision of each other, they only perceive there is another agent from the same team inside their range of vision. Thus, they communicate the relative position they found the teammate, using that information to understand which agent they have found (both agents execute the same process and they are able to understand they found each other); and (ii) when agents understand they found each other, they are able to create the filter for the relative position of the teammate, using that filter always it is necessary, for example, translating the coordinates of information received from teammates to its relative position using the filter. Figure 6 shows an example of agents 1 and 19 encountering each other, in which the green lozenges are their range of vision.

Strategy for Sharing Information. After creating filters for other agents' coordinates, agents are able to share information regarding the position of important elements. We implement a strategy in which agents share information about the coordinates of dispensers, goal zones, and role zones. Other elements we identify to be very dynamic, and it would not be worthy to share that information, for example, obstacles that could easily be cleared by other agents (from the same team or from the other team).

The implemented strategy for sharing information is as follows: (i) when an agent finds an element it is worthy to share its coordinates, for example, a dispenser, it sends to all agents it knows a filter the position of that element, already applying the filter, translating the coordinates to the relative position of the receiver of that message, also sending the list of agents it is sharing the information; (ii) when an agent receives a message sharing the information of an element from the grid world, with a list of other receivers, it stores the information in its belief base, and it verifies which other agents (who are not in the list) it knows the filter (those it could also share the information and did not receive yet), then it sends the coordinates of that element applying the filter for each receiver, but now without a list of receivers; (iii) when an agent receives a message sharing the information of an element from the grid world, but without the list of other receivers, it only stores that information in its belief base.

That means agents propagate information on two levels. The first level is when an agent shares information about a relevant element itself found in the grid world, and the second level is when agents share information that was shared by agents who found those relevant elements in the grid world. This strategy was used considering the strategy for creating groups of agents we will discuss next, in which agents only share information with groups directly connected by members.

Strategy for Creating Groups of Agents. We create a strategy in which agents form groups dynamically, depending on the encounters that occurs during each match. The basic idea is that agents which encounter each other will belong to an implicit group, and they are able to collaborate. An agent is able to belong to more than one implicit group of agents, for example, imagine that agent `ag1` encounters agents `ag2, ag3, ag4` and `ag5`, then `ag1` is able to collaborate with each of those agents individually and request collaboration from all of them. However, `ag2` only will be in the same implicit group of agent `ag3` if it encountered `ag3` during the match.

We choose to implement this strategy for creating implicit groups of agents because of other decisions we made regarding other strategies, for example, focusing only on delivering small tasks (with less than 3 blocks). Thus fewer agents are required to deliver a task, and we realised that the encounters that occurred during the matches were enough to form groups large enough to deliver those tasks.

Strategy for Delivering Tasks. We implemented a relatively simple task delivery strategy. When an agent is able to contribute to completing a new task, it queries all agents belonging to its implicit group, asking the distance required to them to help deliver the task in a particular goal zone, according to the protocol[6] shown in Fig. 7. Then, when other agents are able to help (they are not helping another agent with another task), they answer the query informing the distance needed to reach the goal zone with the necessary block. Then, the

[6] The inspiration for this protocol comes from the Contract Net Protocol [34].

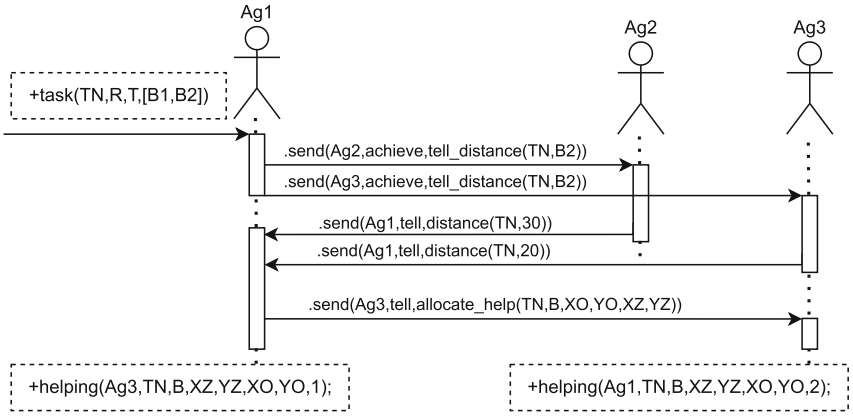

Fig. 7. Allocation Help Protocol.

agent which requested help chooses the close agent to help it, informing the winning agent, and they start moving towards the goal zone to deliver the task.

Figure 8 shows an example in which agent 7 will deliver the `task2` (from the task border at left of the figure) which is worth $40. Note that tasks also have the requirement of a specific position for the agent that will deliver the task, in the case of `task2`, it requires the agent to be north of the block `b0`. Also, we are able to observe in Fig. 8, agent 16 helping to build that task, positioning the block `b1` according to the `task2` requirement. Also, Fig. 3 shows agent 4 delivering the `task3` with help of agent 2.

Fig. 8. Example of a task being delivered.

3.4 Tests

The test strategy was based on using different combinations for the multiple modules described in Sect. 3.2, executing the system, and observing the agents' behaviour and the score agents made during different simulations. During tests, we fixed some issues related to scenarios/situations we did not predict during the implementation. One of the issues fixed after executing some tests was related to strategies for avoiding and deviating from other agents, mainly in the crowded goal zones. Also, we realised that for some goal zones, agents which were not synchronised could try to deliver tasks at the same position, thus we implemented a strategy in which agents wait in a safe zone until their teammate approach to deliver the task they are helping.

4 Results

In the MAPC 2022, the LI(A)RA team ends in 4th place, tied with the GOAL-DTU team. During the contest, we won one round against the FIT BUT team, 2 rounds against the GOAL-DTU team (winning that match), and lose all rounds against GOALdigger and MMD. Our total score was 9 points (regarding the 3 rounds our implementation won). Table 2 shows the final scores for all teams.

Table 2. Final Scores.

Place	Team	Score
1	MMD	30
2	GOALdigger	22
3	FIT BUT	19
4	GOAL-DTU	9
4	LI(A)RA	9

Analysing the matches, our multi-agent system did not score many points against teams with more aggressive adversarial attitudes, which means, teams that implemented strategies for attacking agents from other teams. These attitudes were unexpected, given there is no history of this kind of attitude from other teams in the past, although the scenario is very favourable for this kind of attitude, in which agents can clear other agents taking their energy. Thus, competing with teams exhibiting these attitudes and having not expected such attitudes, our multi-agent system was vulnerable to other teams' attacks.

Attacking other agents during the match has shown to be a very interesting strategy, keeping agents responsible to attack others close to goal zones, in which they could not only take the energy of other agents but also destroy their blocks, which required the agents of other teams to search for blocks again. We intend to explore these attitudes in future participation in the MAPC.

Table 3 shows some estimated metrics about the teams' implementations collected by the organisers of MAPC 2022 and shared with all teams. In Table 3 is possible to note that: (i) our team had a larger number of developers, which

we consider a challenge for this kind of development; (ii) our multi-agent system was developed in fewer hours than most of the other teams, we are the second team which expended fewer hours implementing the system, but GOAL-DTU has already participated in the previous year, then those metrics may be related only to increments made for this year scenario, while we started our implementation this year; (iii) our implementation is the shorter on; however, it is important to mention that it is difficult to count lines of code of different programming languages fairly; and (iv) we are the only team which use Jason platform.

Table 3. Estimated metrics.

Team	Members	Time	Lines of code	Platform	2020(-)
FIT BUT	3	240 h	12000	JAVA(+JADE)	Yes
GOAL-DTU	3	**30 h**	2000	GOAL	Yes
GOALdigger	4	1200 h	10000	GOAL	No
LI(A)RA	**5**	80 h + **40 h**	**1100**	**Jason**	No
MMD	2	896 h	5407	Python	No

5 Conclusion

In this paper, we described the LI(A)RA team's implementation for the MAPC 2022 scenario called Agents Assemble III. Besides summarising the main challenges related to the MAPC 2022 scenario from our perspective, we focused on: (i) describing the technology used to implement our multi-agent system; (ii) detailing the methodology used by our team to implement the multi-agent system, which was a modular and incremental approach; and (iii) describing the strategies we implemented for agents, according to different activities they could execute in the MAPC 2022 scenario.

There were very interesting behaviours that could be observed in the MAPC 2002 contest from other teams, for example, more competitive attitudes in the sense of attacking agents from other teams with the clear events, which surprised our team. We did not implement any strategy for agents to defend themselves from attacks, and we believe those attitudes were decisive in the final scores. Also, we realised our implementation may be shorter than all others, also we dedicate considerably less time to planning and implementing the multi-agent system than the teams with higher scores. Finally, we also did not finish the implementation of all modules we intended during the planning phase, which also may have compromised our scores.

We intend to participate next year, starting the planning phase early, dedicating more time to the implementation phase, also exploring approaches for creating modules of agent-oriented programs already developed over the Jason Platform, as the approach presented in [19].

Table 4. Predicates used in the implementation.

Predicate	Meaning
`my_role(Role)`	a belief describing the role that particular agent should adopt during the match. The variable `Role` will unify with the name of the role that agent will adopt during the match, for example, the role `worker`, i.e., `my_role(worker)`
`maxBlocks(X)`	it stores the information of the max number of blocks agents will carry during the matches, `X` will unify with such max number of blocks. Different agents may be able to carry different numbers of blocks according to the strategies developed
`goingDirection(X)`	it represents a memory of which direction an agent is going, for example, `goingDirection(n)` indicating the agent is going north
`role(X)`	it describes the role of a particular agent during a match
`position(X,Y)`	it describes the current agent position, it is also an agent memory that holds updated, considering there is no absolute position in the MAPC's scenarios
`attached(X,Y)`	it describes that there is something attacked to an agent at coordinates `X` and `Y`
`lastActionResult(X)`	it describes the result of the last action executed by that agent
`lastAction(T)`	it describes the last action executed by that agent
`lastActionParams(L)`	it describes the parameters of the last action executed by that agent
`team(T)`	it describes the team of the agent
`collectingBlocks`	it represents a memory that the agent is collecting blocks
`carryingMaxBlocks`	it represents a memory that the agent is carrying the max of blocks it is capable to carry
`roleAbleBlocks`	it represents that particular agent is able to carry blocks, i.e., it plays a role able to collect and carry blocks
`carryingBlock`	it represents a memory that the agent is carrying blocks
`has_block(T)`	it represents a memory that agent has (it is carrying) a block of the type `T`
`movingToDispenser(X,Y,T)`	it represents the memory that the agent is moving to a dispenser at coordinates `X` and `Y`, and this dispenser has blocks of the type `T`

Table 5. Predicates used in the implementation.

Predicate	Meaning
`movingToRoleZone(X,Y)`	it represents a memory that the agent is moving to a role zone. `X` and `Y` are the coordinates of a role zone the agent is currently moving to it, e.g., `movingToRoleZone(-20,35)`
`movingToGoalZone(X,Y)`	it represents a memory that the agent is moving to a goal zone; `(X,Y)` is the goal zone coordinate the agent is currently moving to it, e.g., `movingToGoalZone(10,25)`
`obstacle_at(X,Y)`	an inference that represents obstacles inside the agent's field of view
`obstacle_cannot_clear_at(X, Y)`	an inference that represents obstacles that cannot be cleared inside the agent's field of view
`collectingBlocks(X,Y,T)`	it represents a memory that the agent is collecting blocks from a particular dispenser at coordinates `X` and `Y` of the type `T`
`roleZone(X,Y) [source(memory)]`	it represents a memory related to the coordinates of a role zone that the agent found during its execution
`roleZone(X,Y) [source(percept)]`	it represents a perception of a role zone inside the agent's field of view
`goalzone(X,Y) [source(memory)]`	it represents a memory related to the coordinates of a goal zone that the agent found during its execution
`thing(X,Y,dispenser,P) [source(memory)]`	it represents a memory related to the coordinates of a dispenser that the agent found during its execution
`thing(X,Y,T,P) [source(percept)]`	it represents a perception related to things inside the agent's field of view. Things can be dispensers, entities, etc.
`closest(goalzone, X, Y)`	an inference that allows agents to find the coordinates of the closest goal zone
`closest(rolezone, X, Y)`	an inference that allows agents to find the coordinates of the closest role zone
`closest(dispenser, T, X, Y)`	an inference that allows agents to find the coordinates of the closest role zone

Table 6. Predicates used in the implementation.

Predicate	Meaning
`task(T,Time,R,E)[source(percept)]`	a perception for a task `T`, in which the agent has available the duration and reward related to that task
`norm(N,I,F,R,Number) [source(percept)]`	a perception for a norm
`cost(Distance,BlockType,TName) [source(TeamMate)]`	informs the distance a teammate needs to help to build a particular task, contributing with the block `BlockType`
`allocate_help(T,B,XZ,YZ,XO,YO) [source(TeamMate)]`	a belief the agent is trying to allocate a particular agent to help it
`mate_filter(TeamMate,XFilter,YFilter)`	a filter that allows agents to translate coordinates to the relative coordinates of other agents
`requested_collaboration(T,XZ,YZ,XO,YO,B)`	a belief for all requested collaboration regarding tasks
`cannot_deliver(TName)`	a belief that the agent cannot deliver a particular task, that means, the agents already reasoned about that task and concluded it cannot deliver it
`helping(Ag,T,B,XZ,YZ,XO,YO,N)`	a memory that the agent is helping another agent to complete a particular task
`found_mate(XMate,YMate,XMy,YMy,S) [source(memory)]`	a temporary belief used to synchronisation strategy, keeping the information of a teammate the agent found in the grid world
`inform_position(T,XT,YT,Parameters) [list(List),source(TeamMate)]`	used for informing the position of elements found by agents in the grid world
`waiting_away(Ag,TName,XMy,YMy,S)`	a belief that the agent is waiting away from the goal zone
`submitting(_,_)`	a memory informing the agent is submitting a particular task

16th Multi-agent Programming Contest: All Questions Answered

LI(A)RA Team
Federal University of Santa Catarina

A Team Overview: Short Answers

A.1 Participants and Their Background

Who is part of your team?
We are 3 undergraduate students, named Marcelo, Michele and Ricardo, from the Computer Engineering program at the Federal University of Santa Catarina (UFSC), under supervision by Professor Alison R. Panisson and PhD Giovani P. Farias. Alison and Giovani have about 10 years of experience in multi-agent systems programming, and the students have bout 1 year of experience. Students are learning about multi-agent systems in undergraduate courses and at the LI(A)RA project, which focuses on teaching multi-agent systems technology.

What was your motivation to participate in the contest?
Giovani and Alison have participated in MAPC in the past, and they always thought about creating a project focusing on teaching multi-agent technologies to students (undergraduate and graduate students), in which they could apply the knowledge from this learning to MAPC problems. One of the more evident motivations is evaluating the multi-agent platforms, languages, and methodologies to develop complex multi-agent systems.

What is the history of your group? (course project, thesis, ...)
LI(A)RA project was created in February of 2022 by Professor Alison at UFSC, in collaboration with the LIA (Academic League of Artificial Intelligence) focused on teaching and exploring multi-agent systems technologies.

What is your field of research? Which work therein is related?
Alison and Giovani are both researchers in the field of multi-agent systems. Alison's main research interest is multi-agent (software agents and humans) communication using argumentation. Giovani's main research interest is multi-task planning.

A.2 Statistics

Did you start your agent team from scratch, or did you build on existing agents (from yourself or another previous participant)?
We started from scratch, about 2 months before the contest.

How much time did you invest in the contest (for programming, organising your group, other)?
Alison has spent about 80 h of programming (during his vacation time), and the group has spent about 40 h discussing strategies, previous papers from MAPC, and organising the infrastructure to develop and test the system.

How was the time (roughly) distributed over the months before the contest?
During about 5 months the group had meetings to talk about strategies and the previous papers from MAPC. Alison implemented the system 2 weeks before the qualification phase.

How many lines of code did you produce for your final agent team?
About 1100 lines of code

A.3 Technology and Techniques

Did you use any of these agent technology/AOSE methods or tools? What were your experiences?

Agent programming languages and/or frameworks?
We used purely Jason.

Methodologies (e.g. Prometheus)?
Usual software engineering methodologies, with incremental and modular components (in the case of Jason agents, an incremental and modular plan library).

Notation (e.g. Agent UML)?
Table of predicates with their meaning and decision-making flowcharts.

Coordination mechanisms (e.g. protocols, games, . . .)?
A complete decentralised mechanism, using communication protocols, for synchronising and organisation (similar to Contract Net Protocol).

Other (methods/concepts/tools)?
Although Jason provides a series of structural programming structures, we opted to make the code more declarative as possible, respecting the original declarative programming paradigm of the language.

What hardware did you use during the contest?
To run our agents in the contest, we used an Avell A52 LIV notebook, with the following specifications:

- Processor: Intel Core i5-10300H (4.5 GHz max clock);
- Graphics Card: NVidia GeForce GTX 1650Ti (with 4GB GDDR6 dedicated RAM);
- Memory: 16 GB DDR4 [2 × 8 GB - Dual Channel] @3200 MHz;
- Hard Drive: SSD M.2 NVME 500 GiB;
- Wireless Card: Intel Dual Band Wireless-9462 + Bluetooth 5.1.

A.4 Agent System Details

Would you say your system is decentralised? Why?
Completely decentralised, no central mechanisms were utilised.

Do your agents use the following features: Planning, Learning, Organisations, Norms? If so, please elaborate briefly.
They use simple strategies organised by priority of plans in their plan library, in which plans are enabled by a mechanism of memory.

How do your agents cooperate?
They synchronise when find each other, requesting help to complete tasks when they are able to participate in those tasks, verifying the best match in the group of agents that can help.

Can your agents change their general behaviour during run time? If so, what triggers the changes?
Yes, they use a mechanism of memory to influence their behaviour. All our agents are instances of the same code.

Did you have to make changes to the team (e.g. fix critical bugs) during the contest?
We opted to keep our original code during the contest without any change.

How did you go about debugging your system? What kinds of measures could improve your debugging experience?
Debugging is normally hard. We basically executed the system, analysed the executions, and did inspections on agents and environment state to find and fix bugs.

During the contest, you were not allowed to watch the matches. How did you track what was going on? Was it helpful?
We did not track what was going on (unfortunately we only were able to organise the system execution during the contest)

Did you invest time in making your agents more robust/fault-tolerant? How?
Yes, identifying possible problems during tests.

A.5 Scenario and Strategy

How would you describe your intended agent behaviour? Did the actual behaviour deviate from that?
Agents dynamically adapted their behaviour according to the result of exploration of the environment and the whether or not they were able to meet other agents at the same time.

Why did your team perform as it did? Why did the other teams perform better/worse than you did?
We consider our implementation was about 35–40% complete, which is resulting from the fact the team started to implement the system very late. We consider we had a very good result considering the state of our implementation and the fact it was the first year the team participated in the contest. One behaviour we did not predict from other teams and it make a great difference was agents attacking others to avoid other teams delivering tasks (We believe it was the first time teams used this kind of *aggressive* strategies) and it worked very well for them (against us).

Did you implement any strategy that tries to interfere with your opponents?
We did not. Others who implemented got an advantage in matches, which seems an interesting direction to pursue during the next contests.

How do your agents coordinate assembling and delivering a structure for a task?
They used a coordination protocol, similar to the contract net protocol, in which one agent was responsible to coordinate the delivery.

Which aspect(s) of the scenario did you find particularly challenging?
Using no absolute position for agents was the most challenging aspect.

What would you improve (wrt. your agents) if you wanted to participate in the same contest a week from now (or next year)?
We would finish our implementation, which was about 35–40% complete.

What can be improved regarding the scenario for next year? What would you remove? What would you add?
We found the scenario very challenging. We would suggest 3D (three-dimensional) tasks, and also agents would have the capability to defend themselves from clear events (some agents would have this capability according to their roles).

A.6 And the Moral of it is ...

What did you learn from participating in the contest?
From the technological point of view, we learned that the platform provides enough to implement this kind of complex multi-agent system to solve complex problems (dynamic, non-deterministic, etc.).
From the software engineering point of view, we were able to use an incremental approach to develop "modules of behaviour", all put together when instantiating agents. All our agents had the same code (same implementation to all our agents), which also is a very interesting achievement.
We found it very difficult to coordinate the team during implementation, but it is because we had a very short time to implement our system (about 2 weeks only)

What advice would you give to yourself before the contest/another team wanting to participate in the next?
Start the implementation as soon as you can, but first study the scenario in detail to plan how to implement your multi-agent system. Also, studying the scenario will provide you with a short of strategies (choices) we will eventually make.

Where did you benefit from your chosen programming language, methodology, tools, and algorithms?
Mostly because of our experience with the language, but it also is very elegant (declarative language) and we would like to check if we could keep it as declarative as possible (we succeed in this).

Which problems did you encounter because of your chosen technologies?
Our system become slow with many agents on regular laptops, and we had to execute it from a better machine (but nothing very powerful like a server). I believe other teams had the same problem independent of the technology they used.

Otherwise, the technology fulfilled our needs.

Which aspect of your team cost you the most time?

Testing. Some situations we would like to test cost many simulations in which we had to wait for the specific situation to happen to verify if our implementation was efficient. We did not explore more sophisticated manners to test our implementation (re-configuring the server, for example).

A.7 Looking into the Future

Did the warm-up match help improve your team of agents? How useful do you think it is?

We did not change our code after the warm-up, but it was useful to verify the connection with the server, the performance of our machine executing the system, etc.

What are your thoughts on changing how the contest is run, so that the participants' agents are executed on the same infrastructure by the organisers? What do you see as positive or negative about this approach?

I believe it would be very positive, giving us better benchmarks in which the infrastructure is not a variable anymore.

Do you think a match containing more than two teams should be mandatory?

I believe it depends on the scenario. 2019–2022 scenarios would have more than a team and it would have been fun. For other scenarios, we are not sure if it would make sense.

What else can be improved regarding the MAPC for next year?

We are very excited about next year's contest and we have no more suggestions. Thank you for organising it and keep it up.

References

1. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional (2000)
2. Boehm, B.W.: A spiral model of software development and enhancement. Computer **21**(5), 61–72 (1988)
3. Boissier, O., Bordini, R.H., Hubner, J., Ricci, A.: Multi-agent Oriented Programming: Programming Multi-agent Systems Using JaCaMo. MIT Press, Cambridge (2020)
4. Bordini, R.H., Dastani, M., Dix, J., Seghrouchni, A.E.F. (eds.): Multi-Agent Programming, Languages, Tools and Applications. Springer, Heidelberg (2009). https://doi.org/10.1007/978-0-387-89299-3
5. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-agent Systems in AgentSpeak Using Jason. Wiley, Hoboken (2007)

6. Bratman, M.: Intention, Plans, and Practical Reason. Harvard University Press, Cambridge (1987)
7. Cernuzzi, L., Cossentino, M., Zambonelli, F.: Process models for agent-based development. Eng. Appl. Artif. Intell. **18**(2), 205–222 (2005)
8. Engelmann, D., et al.: Dial4JaCa – a demonstration. In: Dignum, F., Corchado, J.M., De La Prieta, F. (eds.) PAAMS 2021. LNCS (LNAI), vol. 12946, pp. 346–350. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85739-4_29
9. Engelmann, D., et al.: Dial4JaCa – a communication interface between multi-agent systems and chatbots. In: Dignum, F., Corchado, J.M., De La Prieta, F. (eds.) PAAMS 2021. LNCS (LNAI), vol. 12946, pp. 77–88. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85739-4_7
10. Engelmann, D.C., Cezar, L.D., Panisson, A.R., Bordini, R.H.: A conversational agent to support hospital bed allocation. In: Britto, A., Valdivia Delgado, K. (eds.) BRACIS 2021. LNCS (LNAI), vol. 13073, pp. 3–17. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-91702-9_1
11. Engelmann, D.C., Ferrando, A., Panisson, A.R., Ancona, D., Bordini, R.H., Mascardi, V.: Rv4jaca - runtime verification for multi-agent systems. In: Cardoso, R.C., Ferrando, A., Papacchini, F., Askarpour, M., Dennis, L.A. (eds.) Proceedings of the Second Workshop on Agents and Robots for reliable Engineered Autonomy, AREA@IJCAI-ECAI 2022, Vienna, Austria, 24th July 2022. EPTCS, vol. 362, pp. 23–36 (2022)
12. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an agent communication language. In: Proceedings of the Third International Conference on Information and Knowledge Management, pp. 456–463 (1994)
13. Fowler, M., Highsmith, J., et al.: The agile manifesto. Softw. Dev. **9**(8), 28–35 (2001)
14. George, M., Lansky, A.: Reactive reasoning and planning: An experiment with a mobile robot. In: The Proceedings of the Sixth National Conference on Artificial Intelligence, pp. 677–682 (1987)
15. Madden, N., Logan, B.: Modularity and compositionality in Jason. In: Braubach, L., Briot, J.-P., Thangarajah, J. (eds.) ProMAS 2009. LNCS (LNAI), vol. 5919, pp. 237–253. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14843-9_15
16. Melo, V.S., Panisson, A.R., Bordini, R.H.: Meta-information and argumentation in multi-agent systems. iSys-Braz. J. Inf. Syst. **10**(3), 74–97 (2017)
17. Melo, V., Panisson, A., Bordini, R.: MIRS: A modular approach for using meta-information in agent-oriented programming languages. In: Nineteenth International Workshop on Trust in Agent Societies (2017)
18. Müller, J.P., Pischel, M., Thiel, M.: Modeling reactive behaviour in vertically layered agent architectures. In: Wooldridge, M.J., Jennings, N.R. (eds.) ATAL 1994. LNCS, vol. 890, pp. 261–276. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-58855-8_17
19. Ortiz-Hernández, G., Guerra-Hernández, A., Hübner, J.F., Luna-Ramírez, W.A.: Modularization in belief-desire-intention agent programming and artifact-based environments. PeerJ Comput. Sci. **8**, e1162 (2022)
20. Ortiz-Hernández, G., Hübner, J.F., Bordini, R.H., Guerra-Hernández, A., Hoyos-Rivera, G.J., Cruz-Ramírez, N.: A namespace approach for modularity in BDI programming languages. In: Baldoni, M., Müller, J.P., Nunes, I., Zalila-Wenkstern, R. (eds.) EMAS 2016. LNCS (LNAI), vol. 10093, pp. 117–135. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50983-9_7

21. Panisson, A.R.: M-arguments. In: 2020 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT), pp. 161–168 (2020)
22. Panisson, A.R., Bordini, R.H.: Knowledge representation for argumentation in agent-oriented programming languages. In: 2016 5th Brazilian Conference on Intelligent Systems (BRACIS), pp. 13–18. IEEE (2016)
23. Panisson, A.R., et al.: Arguing about task reallocation using ontological information in multi-agent systems. In: 12th International Workshop on Argumentation in Multiagent Systems, vol. 108 (2015)
24. Panisson, A.R., McBurney, P., Bordini, R.H.: A computational model of argumentation schemes for multi-agent systems. Argument Comput. **12**(3), 1–39 (2021)
25. Panisson, A.R., Meneguzzi, F., Vieira, R., Bordin, R.H.: Towards practical argumentation in multi-agent systems. In: 2015 Brazilian Conference on Intelligent Systems (BRACIS), pp. 98–103. IEEE (2015)
26. Panisson, A.R., Meneguzzi, F., Vieira, R., Bordini, R.H.: Towards practical argumentation-based dialogues in multi-agent systems. In: 2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT), vol. 2, pp. 151–158. IEEE (2015)
27. Panisson, A.R., Sarkadi, S., McBurney, P., Parsons, S., Bordini, R.H.: On the formal semantics of theory of mind in agent communication. In: Lujak, M. (ed.) AT 2018. LNCS (LNAI), vol. 11327, pp. 18–32. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17294-7_2
28. Panisson, A.R.: A framework for reasoning and dialogue in multi-agent systems using argumentation schemes. Ph.D. thesis, Pontifícia Universidade Católica do Rio Grande do Sul (2019)
29. Panisson, A.R., Meneguzzi, F.R., Fagundes, M.S., Vieira, R., Bordini, R.H.: Formal semantics of speech acts for argumentative dialogues. In: 2014 Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems, França (2014)
30. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Van de Velde, W., Perram, J.W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0031845
31. Sarkadi, Ş, Panisson, A.R., Bordini, R.H., McBurney, P., Parsons, S.: Towards an approach for modelling uncertain theory of mind in multi-agent systems. In: Lujak, M. (ed.) AT 2018. LNCS (LNAI), vol. 11327, pp. 3–17. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17294-7_1
32. Schmidt, D., Panisson, A.R., Freitas, A., Bordini, R.H., Meneguzzi, F., Vieira, R.: An ontology-based mobile application for task managing in collaborative groups. In: Markov, Z., Russell, I. (eds.) Proceedings of the Twenty-Ninth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2016, Key Largo, Florida, USA, 16–18 May 2016, pp. 522–526. AAAI Press (2016)
33. da Silveira Colissi, M., Vieira, R., Mascardi, V., Bordini, R.H.: A chatbot that uses a multi-agent organization to support collaborative learning. In: Stephanidis, C., Antona, M., Ntoa, S. (eds.) HCII 2021. CCIS, vol. 1421, pp. 31–38. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78645-8_4
34. Smith, R.G.: The contract net protocol: high-level communication and control in a distributed problem solver. IEEE Trans. Comput. **29**(12), 1104–1113 (1980)
35. Vieira, R., Moreira, Á.F., Wooldridge, M., Bordini, R.H.: On the formal semantics of speech-act based communication in an agent-oriented programming language. J. Artif. Intell. Res. **29**, 221–267 (2007)