Team PUCRS: a Decentralised Multi-Agent Solution for the Agents in the City Scenario

Rafael C. Cardoso*, Ramon Fraga Pereira, Guilherme Krzisch, Maurício C. Magnaguagno, Túlio Baségio, and Felipe Meneguzzi

Faculdade de Informática – PUCRS,
Avenida Ipiranga, 6681 – Porto Alegre, RS, Brazil
E-mail: rafael.caue@acad.pucrs.br
E-mail: ramon.pereira@acad.pucrs.br
E-mail: guilherme.krzisch@acad.pucrs.br
E-mail: mauricio.magnaguagno@acad.pucrs.br
E-mail: tulio.basegio@acad.pucrs.br
E-mail: felipe.meneguzzi@pucrs.br
*Corresponding author

Abstract: The 2016 edition of the Multi-Agent Programming Contest used the Agents in the City as its new scenario, which consisted on the execution of various logistics tasks within a realistic city topology using a number of different vehicle types. The complexity of the scenario and variety of tasks posed a challenging problem suitable for a decentralised multi-agent solution. In this paper we describe the winning solution for the contest, providing insights into how we designed our solution and organised our team, as well as some discussion on a few of our matches.

Keywords: Multi-Agent Programming Contest; Decentralised Multi-Agent Systems; Agents in the City; Contract Net Protocol; JaCaMo.

Reference to this paper should be made as follows: Cardoso, R.C. et al. (201X) 'Team PUCRS: a Decentralised Multi-Agent Solution for the Agents in the City Scenario', *of Agent-Oriented Software Engineering*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Rafael C. Cardoso is a PhD student in Computer Science at PUCRS. He obtained his Master's degree in Computer Science from the same institution in 2014. His main research interests are in Multi-Agent Systems and Multi-Agent Planning. Specifically, he is interested in multi-agent programming languages, benchmarks of agent-based languages, goal allocation mechanisms, multi-agent planners, and coordination mechanisms.

Ramon Fraga Pereira is a PhD student in Computer Science at PUCRS. He obtained his Master's degree in Computer Science from the same institution in 2016. He was recognized as having the second best MSc dissertation in Artificial Intelligence in Brazil (Title: Landmark-Based Plan Recognition), in BRACIS/CTDIAC 2016. His main research interest is in Goal and Plan Recognition using Automated Planning techniques for both Single-Agent and Multi-Agent Systems.

Guilherme Krzisch is a Master student in Computer Science at PUCRS. His main research interest is in Normative Systems; specifically, he is interested in how

to effectively monitor norm violations and in how agents need to change their reasoning in such systems.

Maurício C. Magnaguagno is a PhD student in Computer Science at PUCRS and he received the Master's degree in Computer Science from the same institution in 2016. His research focus on automated planning.

Túlio Baségio is a PhD student in Computer Science at PUCRS and he received the Master's degree in Computer Science from the same institution in 2007. His research interests include Multi-Agent and Multi-Robot Systems, specifically focused on distributed coordination and task allocation mechanisms.

Felipe Meneguzzi is an Associate Professor at the School of Computer Science at PUCRS where he leads the Group on Autonomous Systems. His main research interests are Automated Planning, Plan Recognition, Norm-driven reasoning and their application to Multi-Agent Systems. Having received his PhD from King's College London and worked for Carnegie Mellon University prior to his current appointment, he has authored over 100 peer-reviewed papers and articles, as well as a number of industrial patents. He is currently in the program committee of all major AI and agents conferences, including AAAI, IJCAI, AAMAS, and a number of smaller venues. Felipe has received Google's 2016 Research Award for Latin America and was a runner up to Microsoft's 2013 Microsoft Research Faculty Fellowship.

1 Introduction

The Multi-Agent Contest (MAPC) 2016 introduced a new and complex scenario, *Agents in the City*. This scenario consists of two teams competing to accomplish logistic tasks in the streets of a realistic city. Each team controls 16 autonomous vehicles divided into four different types: car, drone, motorcycle, and truck. Teams win points for having more money than the opposing team at the end of a round and earn money by successfully bidding for and completing logistic tasks. In this paper we focus on describing our team's solution and leave the interested reader to find more details about the scenario in [1].

Although our team was originally formed in 2014, 2016 was the first time that we participated in the MAPC. We chose not to participate in 2014 for two main reasons. First, we did not have enough time to familiarise with the server's API back in 2014. Second, the *Agents on Mars* scenario was in its fourth iteration, giving a disadvantage to any new teams. As soon as the MAPC 2015 was announced, we started to develop our solution, using the multi-agent oriented programming language JaCaMo. Due to the complexity of developing the new *Agents in the City* scenario, the contest was pushed to 2016.

We had many team members over this three year period, and as a result our code shifted considerably throughout the development process. Our team has members from two different research laboratories, SMART-PUCRS and LSA-PUCRS. We started with a centralised solution and gradually decentralised it as much as possible with one key change over the year. Specifically, after receiving advice on how to improve our code from one of the developers of JaCaMo, we decided to completely overhaul the way that we structured the plan library of our agents. The resulting agent code moved from entirely reactive agents to agents that are more proactive, but are still able to react when appropriate.

This paper is structured as follows. In the next section, we explain the design process we used in our implementation: we describe how we managed the team, how we modelled the system, and how we tested our software. Section 3 describes the details of our architecture, with a brief description of JaCaMo and other parts of the architecture that are used throughout our solution. In Section 4, we discuss the main the strategies the agents employ as they play the scenario and explain how they work. Section 5 briefly analyses the results that contributed to our team obtaining first place. Finally, we succinctly answer the questions posed by the organisers for each team in Section 6, and we end the paper with some closing remarks in the final section.

2 System Analysis and Design

In this section we discuss how our meetings were organised and provide some thoughts on collaborative programming with GitHub. We briefly comment on the use of the Prometheus methodology as a modelling tool for our Multi-Agent System (MAS). We end the section providing some information on how we tested our system, using bug tracking and evaluation of preliminary implementations.

2.1 Team Management and Meetings

Our team had weekly meetings ranging from 15 minutes to 3 hours, organised by the team leader. We had three different types of meetings: *progress report*, *strategy discussion*, and *programming meetings*. *Progress report* were short meetings (15 to 30 minutes) where each team member would report on the progress of the task allocated to them. *Strategy discussion* were medium-length meetings (1 to 2 hours) where we did in-depth discussions about possible strategies for a particular feature, such as job evaluation, by writing viable strategies on a whiteboard until we all agreed on which one to implement. *Programming meetings* were lengthy meetings (3 or more hours), usually done when we felt that we were behind schedule, where each team member brought their laptop and focused on implementing some important feature that we needed to proceed. At the end of any of these meetings, we verified what was left to be done and allocated tasks that should be prioritised.

Programming meetings were not very common, we usually had to program on our own free time. We used *GitHub* to host our repository online, making it easier to collaborate and track development, especially because we had a large group of people. Besides using it for source code management, distributed version control, and access control, we also took advantage of collaboration features such as issue tracking, task management, and wikis. When trying different strategies or adding a big feature, we would often create new branches in order to test the code first, before merging it into the master branch.

In Figure 1a, we show a chart with the number of commits sent to our repository, from the first commit on the day that we started programming (May 17, 2015), to the last commit on the last day of the competition (September 13, 2016). This chart indicates that the bulk of the code was done in 2015, when most of the architecture was implemented. Conversely, the chart in Figure 1b indicates that the month before the 2016 competition had the most number of additions to the code, when most of our strategies were implemented. In both charts, there is a large hiatus in activity separating 2015 and 2016, indicative of when the 2015 contest was cancelled.

We initially created GitHub wikis for sharing some important information about the contest, such as agents' roles and their parameters, contest schedule, and how to execute the

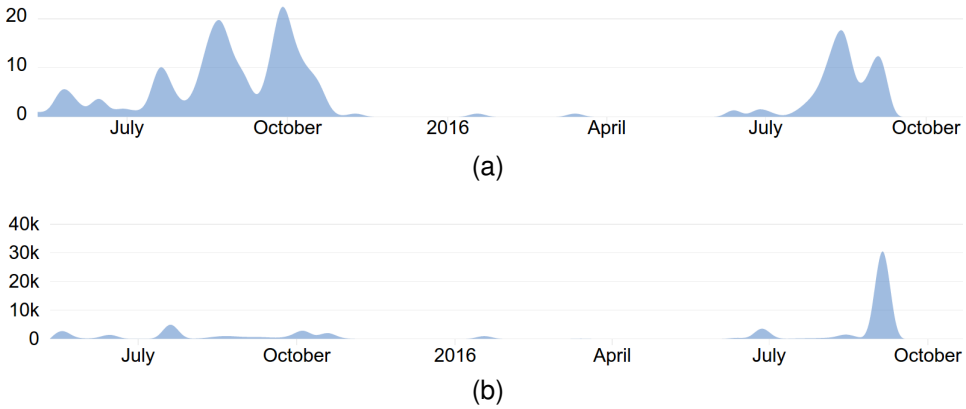

Figure 1 (a) Number of commits from May 17, 2015 – September 13, 2016; (b) number of additions. (Source: GitHub)

server. We quickly noticed the complexity of the problem and decided to write and save our meeting minutes. We used wikis to do so, creating one page for each of the meetings that we had scheduled, and updating it after a meeting ended. Our first recorded meeting minute is from 27/05/2015 (available in our GitHub wiki section), all subsequent meetings can be found in the wiki's sidebar. This was very useful since not everyone was able to attend all meetings, allowing any absent members to quickly catch up on what was discussed and what were the next steps.

2.2 *Modelling Methodology*

The Multi-Agent Programming Contest (MAPC) 2016 introduced *Agents in the City*, a novel and complex scenario. In order to provide a better understanding of this scenario, we opted to use the Prometheus [2] methodology to model the system and our agents. Because of time constraints, we had to intercalate modelling in Prometheus with implementing the system in JaCaMo [3], and through this process we discovered some limitations and conceptual divergences when using Prometheus to model MAS developed in JaCaMo. We discuss these differences in a previous paper [4].

In Figure 2, we show the system overview diagram, modelled using the Prometheus Design Tool. This diagram includes initiator and bidder agents from our task allocation strategy (see Sections 3.4 and 4.5), and an overview of the interaction between vehicle agents and the server's environment. For more information about how the server and its environment work, we refer the reader to [1].

Prometheus was invaluable for the initial development of our solution, but because it does not contain the same concepts found in JaCaMo's environment and organisation dimensions, we decided to stop using it once we got further in the development. For future MAPCs, it would be interesting to investigate the use of the AEOlus methodology [5] to model our MAS. AEOlus extends Prometheus to allow support for the environment and organisation dimensions found in JaCaMo.

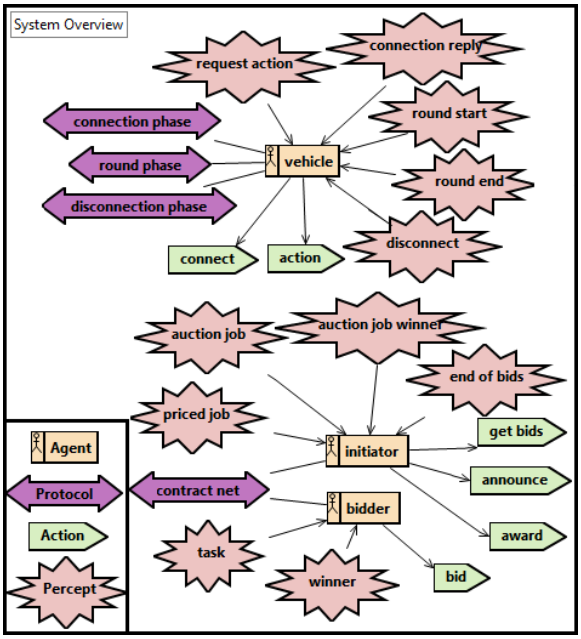

Figure 2 System overview diagram (image adapted from [4]).

2.3 Software Testing

Software testing is important throughout the development of any system, especially when we are considering the inherent complexity and dynamics of multi-agent systems such as in the *Agents in the City* scenario. The purpose of a test phase is to find as many defects as possible in the system. Thus, it would be ideal to test all possible input combinations. However, factors such as the amount of people available, time and, especially, the enormous amount of possibilities, make this impossible for the vast majority of projects, including ours. Due to time constraints, in our project we did not follow the traditional development cycle of well-defined start and end dates for development and testing phases. In fact, these two phases occurred simultaneously throughout our project.

The test phase of our project can be divided into several different parts. We initially had the developers running tests on their own codes. Some developers, that were more experienced with JaCaMo, also performed tests in the code of other developers (these phases would be equivalent to unit test phases and integration test in a traditional development process). However, even while performing tests, programmers continued coding in parallel. Later on, we assigned a team member to work specifically with tests, characterizing what would be called system testing in a traditional development process. Programming and testing took place in parallel, but always in accordance with the availability of team members.

During software testing, we used GitHub's issue tracking system to keep track of the reported bugs during the project. When creating issues using the tracker, it is possible to add labels on them that work as meta data for the issue and that can also be used to filter them. There are some default labels such as bug, duplicate, and enhancement, but the user can also create other labels. As shown in Figure 3a, we used issues to report different aspects

such as bugs, ideas, news, ask questions, request help, and so on. Table 1 provides some data about the number of bugs reported during the testing.

Table 1 Number of bugs reported in GitHub's issue tracker.

Status	#Bugs
Closed	30
Open	11
Invalid	1
Total	42

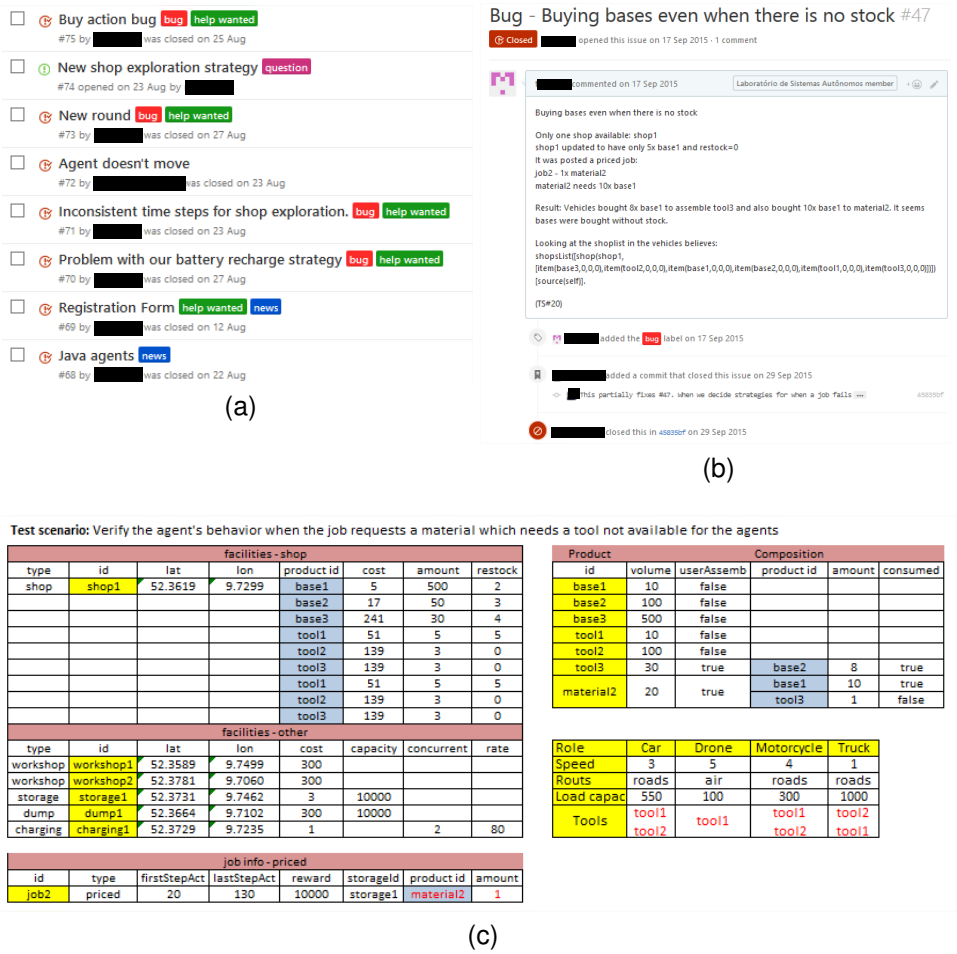

Figure 3 (a) Examples of issues created during the contest; (b) Example of a reported issue; (c) Example of test data used for a test scenario.

The reported bugs were discussed and prioritized during team meetings and informal conversations. As GitHub lacks a specific feature for prioritizing the bugs (although this

could be done through labels such as low, medium, and high), the prioritization list was not kept in the tracker. Another interesting aspect on working with issues in GitHub, is that you can make references to other issues and also for the code used during testing. This was especially useful to us since sometimes the code that was tested could have been changed already.

GitHub's issue tracking system allowed us to record important aspects about the bugs, such as who reported the bug, the time it was created, a description about the error (including how to reproduce it), who worked on it, and its current status. Figure 3b shows an example of a bug reported in our project.

The testing phase occurred dynamically and, most of the time, without documentation about the tests that were performed. However, in some situations the tests were documented through the description of test scenarios and input test data. The following are examples of test scenarios documented during these tests:

- Verify the behaviour of the agents when there is a priced job where the reward is less than the total amount needed to deliver the job.
- Verify the behaviour of the agents when there is an auction job with a short time (impossible) to complete it.
- Verify the behaviour of the agents when a job requests a material which needs a tool that is not available to the agents.
- Verify the behaviour of the agents when there is a job for which agents need to buy bases (items that are not assembled) at different shops to assemble a specific item.

Figure 3c shows an example of input test data used for one of the test scenarios.

It is important to note that most of our tests occurred against the basic scripted team made available by the organizers of the contest, whilst later tests were performed against old versions of our own code. It is also important to comment that small bugs found when programmers were testing their own code were not reported in the system.

3 Software Architecture

This section begins with a brief description of JaCaMo and how we used each of its technologies. Then, we describe several segments of our architecture that are later explored in our strategies: each agent's environment artefact, a centralised team artefact, our map helper, two contract net protocol artefacts, and four Jason internal actions.

JaCaMo [3] is a MAS development platform that explores and exploits the use of three MAS programming dimensions: agent, environment, and organisation. Figure 4 shows how each of these programming dimensions interact with each other. In the top-most level, the organisation dimension is composed of a scheme, a set of missions, and a set of roles. These roles are adopted by the agents that inhabit the agent dimension. In the bottom-most dimension, the environment houses artefacts that relate information about the environment, grouped into workspaces that agents can access. These workspaces can be distributed across multiple network nodes, effectively distributing the MAS.

Jason [6] handles the agent dimension. It is an extension of the AgentSpeak(L) [7] language, based on the BDI agent architecture. Agents in Jason react to events in the system by executing actions on the environment, according to the plans available in each agent's

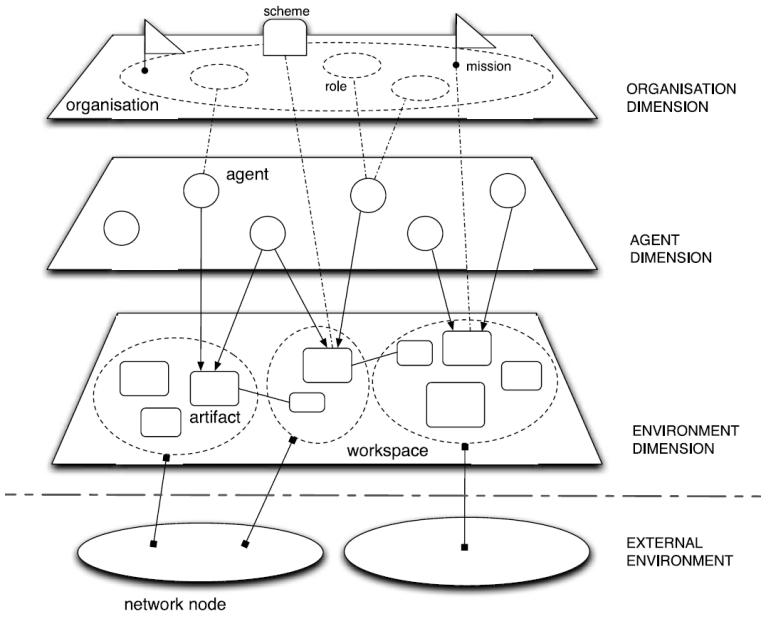

Figure 4 An overview of JaCaMo MAS programming dimensions.

plan library. One of the extensions in Jason is the addition of Prolog-like rules to the belief base of agents. These rules were fundamental in expressing some of the intricacies of this year's MAPC scenario.

CArtAgO [8] is based on the A&A (Agents and Artefacts) model [9], and manages the environment dimension. Artefacts are usually used to represent the environment, but in our case we also use them as an interface between the agents and the server's environment. Artefacts have observable properties – beliefs that are added to the agent's belief base; and provide operations – actions that can be executed by the agents.

Moise [10] operates the organisation dimension, regulating the specification of organisations in the MAS. Moise adds first-class elements to the MAS such as roles, groups, organisational goals, missions, and norms. Agents can adopt roles in the organisation, forming groups and subgroups. Missions are defined to achieve the organisation's goals. The behaviour of the agents that adopt roles to accomplish these missions is guided by norms. Our solution did not use any elements from the organisation dimension, this is something that we plan on exploring in our future MAPCs' solutions.

Project Files

We used the JaCaMo Eclipse plugin to develop our solution. Our project is available at <`https://github.com/smart-pucrs/mapc2016-pucrs`> (pruned code, final repository) or <`https://github.com/lsa-pucrs/mas-pc-pucrs-2016`> (development code, repository with our progress), and contains the following main folders:

- *conf*: server's configuration for simulations.
- *doc*: documentation from the MAPC 2016, does not include any documentation to our code.

- *lib*: assortment of libraries required to run the server and JaCaMo.
- *osm*: OpenStreetMap files containing map data of cities that can be used in simulations.
- *src*: architecture's and agents' source code, divided into four subfolders:
  - *actions*: internal actions described in Section 3.5;
  - *agt*: strategies (see Section 4) employed by Jason agents;
  - *cnp*: CArtAgO artefacts that implement contract net protocol (Section 3.4);
  - *env*: environment architecture programmed in Java and described in Sections 3.1, 3.2, and 3.3.
- *test*: this folder has two executables: `ScenarioConnectToServer.java` and `ScenarioRunServer.java`. The former is used to connect our agents to a server that is already running (e.g., connecting to the contest's server), while the latter first runs the server locally, and then connect our agents and starts the simulation (used for experiments and tests). The JaCaMo project configuration file, `scenario.jcm` is also located in this folder.

3.1 Agent Environment Artefact

To act and receive perceptions in the server's environment we use a CArtAgO artefact called `EISArtifact`. This artefact is responsible for three tasks: register agents, receive (filter) perceptions into observable properties, and send executable actions. When a round starts, all of our agents use this artefact to register with the server by associating them to an entity in the *EnvironmentInterface* of the *MassimServer*. Registering agents is an essential task in the simulation, agents that are not registered in the *MassimServer* are not represented in the environment and can not act in it.

Initially, we were using only one environment artefact for all agents to perceive and act in the server's environment. However, as we progressed to a more decentralised solution, we realised that it made more sense, from a MAS perspective, to have one artefact for each agent. This minimised the bottleneck of sending actions to the server, since each agent would use its own artefact instead of all trying to use the same one. This did not remove the bottleneck entirely, because we were still limited by the number of cores in the computer running our solution. It could provide much better results if the agents were distributed in multiple computers, something that we plan to experiment with in future MAPCs.

At every step of a simulation's round we receive new perceptions, and to deal with them the artefacts can filter any useful perception into an observable property by adding/updating/removing them in the artefact's observable properties. We decided to filter perceptions for two reasons: (1) some perceptions do not change during the simulation, so there is no reason to keep updating them (each addition/update/removal generates an event in Jason); and (2) there were some perceptions that had no use to our agents (e.g., auction jobs), and could be ignored so as to not overload their belief base. We use two sets of filters: match and steps observable properties. *Match observable properties* are perceptions that the artefact receives only once, such as `simStart`, `map`, `steps`, `product`, `role`, and among others. *Steps observable properties* are those that the artefact receives at every step, such as `chargingStation`, `shop`, `storage`, `dump`, `lat`, `lon`, `inFacility`, `step`, and among others.

When our agents decide to act in the environment, they send to their artefact the desired action with the respective parameters (i.e., agent name, action name, and action parameters) that they want to perform in the server. In this scenario, actions can fail due to various reasons, such as wrong parameters, unknown agent (unregistered), uncertainty factor (actions fail with a 1% probability), and among others.

Translator

We developed a `Translator` that contains a set of methods for translating perceptions and actions that the agents receive/send from/to the MassimServer into structures that we use in our solution. In summary, at every step, this set of methods basically translates the data structure of the Massim API into Jason and CArtAgO data structure (e.g, terms, literals).

3.2 Team Artefact

One of the few centralised aspects in our solution is the `Team Artefact`. This artefact is responsible for storing and sharing agents available loads (their capacity to carry items, its value decreases for each item the agent is carrying) and the price of items found in shops. When an agent is located in a *shop* facility, it receives the prices and available quantity of all items within that shop (this information is not available otherwise). We store in the artefact the price of all items that are available to buy in this shop. It is important to emphasise that we store the price of an item only once, we do not update or add this value when another agent revisits that shop, as the items' prices do not change during the simulation.

After an agent adopts a role, we add in the artefact an observable property that corresponds to the load associated with it. We also update the load of an agent when this agent wins a contract to do (part of) a job (this is a projected load, the agent still has in his belief base its true load value), and after delivering a job (i.e., the agent has delivered all items that he had, therefore his available load consequently increases).

3.3 Map Helper using GraphHopper

To obtain routes to locations in the city where agents act and interact, we developed a graph data structure called `MapHelper`. This graph data structure uses the GraphHopper API, an open source Java road routing engine that uses *OpenStreetMap*.

We use `MapHelper` to obtain routes to facilities and locations from the city map in order to avoid executing the environment action `goto` (which also returns a complete route between two locations), because we would have to waste at least one step to execute it, potentially wasting even more steps as this action can fail (e.g., *failed_random* or *failed_no_route*). `MapHelper` provides two types of routes: air (drones) and road (motorcycles, cars, and trucks) routes. Routes contain a sequence of waypoints that go from an initial position to a destination.

These routes are used by our agents to calculate their effectiveness to do a job, that is, the use of `MapHelper` allows agents to estimate the number of steps that would take them to get to a location. These routes still have to be divided by the speed of each vehicle, which we do in the agent's code whenever we call `MapHelper`.

3.4 Contract Net Protocol Artefacts

Our agents use Contract Net Protocol (CNP) [11] to allocate tasks from jobs that are announced in the environment. CNP is a protocol for decentralised task allocation, it has

an *initiator* who announces tasks and awards contracts, and *bidders* who bid for contracts and execute tasks. CArtAgO's source code includes an implementation of CNP, using two artefacts, a `task board` and a `contract net board`.

We extended their implementation with a few extra features such as: a `Bid` class with additional parameters to improve the initiator's bid selection plan which awards contracts; and a CArtAgO internal operation – an operation that can only be executed by the artefact itself – that closes bidding for a contract once all agents have placed their bids. The latter only happens if the contract has not already been closed because the deadline was past. Our agents only place bids once (i.e., they do not update their bids), thus, if all agents have already placed their bid, there is no reason to wait for the deadline, which can save some important time considering that agents also have a deadline to send their actions to the server in each step of the simulation.

One instance of the task board artefact is created by the initiator at the start of the system. All agents focus – while an agent is focusing on an artefact, it is able to see its observable properties and execute its operations – on the artefact as soon as it is available. The artefact has a single operation, `announce`, that allows an initiator to announce new tasks by creating a new observable property `task`, containing information relevant to that particular task. Figure 5 contains a sequence diagram with an overview of how our agents use the CNP artefacts.

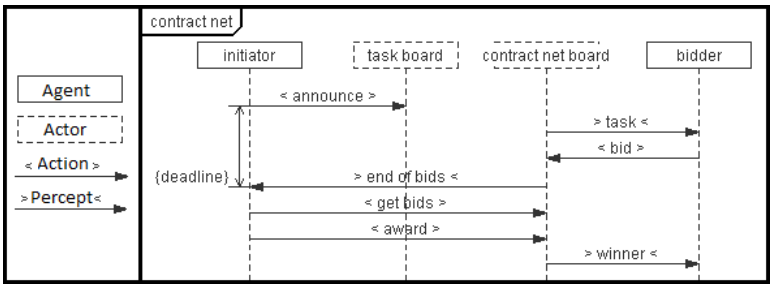

Figure 5 Sequence diagram of how our agents use CNP (image adapted from [4]).

For each new task, an appropriate instance of the contract net board is created, which can then be used to manage its contract. Similarly to the task board, agents will focus on new instances of a contract net board as soon as they are created. A contract net board artefact has a `bid` operation, which agents can execute to place bids on a contract, and a `getBids` operation, used by an initiator to collect the bids placed for a contract. Two internal operations can close and terminate the bidding process over a contract, `checkDeadline` when the deadline is past, and `checkAllBids` when all agents have placed their bids.

3.5 Internal Actions

An action usually performs changes in the environment. That is not the case with Jason internal actions. Internal actions in Jason are executed within the agent, and they can return results that can be used to aid in the agent's reasoning. Jason provides some default internal actions (such as `.send` and `.broadcast` for message passing), as well as allowing the developer to create its own internal actions in Java, such as the ones we describe below.

Closest

The `closest` internal action can be used by the agent to identify the nearest facility, either from the agent position, from the position of another facility, or from specific latitude and longitude coordinates. It is used, for example, to identify the nearest shop, dump, or charge station. This internal action calculates the route to each facility (from a list of facilities received as parameter), identifying the nearest from an initial position (also received as a parameter). It returns the closest facility found.

Get Location

The `getLocation` internal action is used to get the location's latitude and longitude coordinates of a specific facility. It returns that facility's latitude and longitude coordinates.

Route

The `route` internal action returns the route length from an initial position to a specific facility. The route length is calculated from an initial position, which could be: the current agent position; another facility; a set of latitude and longitude coordinates.

Paths to Facilities

The `pathsToFacilities` internal action is used by the agent to calculate the number of steps required from the agent current position to each facility available in a facility list received as a parameter. The number of steps is based in the route length and the agent's speed.

4 Strategies

We start this section by explaining the code in `vehicle.asl`, which contains the code that is loaded by all agents when our solution is executed. Then, we present in-depth discussions on each of our strategies, providing a more detailed description of the following: shop exploration, job evaluation, task allocation, and battery recharge.

In the beginning of `vehicle.asl`, several other `asl` files are included – Jason includes are similar to imports in Java, they add all beliefs, goals, and plans from the file being included. `new-round.asl` and `end-round.asl` contain plans that are used, respectively, in the start and end of a round. We separated common code (to be used by all agents) into several different files: `common-plans.asl`, `common-rules.asl`, `common-actions`, and `common-strategies.asl`. We did this so that agents could also include specific `asl` files related to the vehicle role that they received from the environment, but we never got around to developing strategies for specific types of vehicles, something we plan to do in future MAPCs.

All agents are also bidders, so they also include `bidder.asl`, along with the default `common-cartago.asl` for interfacing with CArtAgO artefacts. We chose to have only initiator (for reasons that we discuss in Section 4.5), thus, only *vehicle15* includes `initiator.asl`. The rest of the code is straightforward, agents register with the server's environment and update some information about themselves in the team artefact. All agents then wait for the first step of the simulation, when our first strategy begins: shop exploration.

4.1 Shop Exploration

We now present a strategy for exploring *shop* facilities at the beginning of a round. This strategy prioritises sending agents to the shop that is nearest to them in order to obtain the prices of available items. The exploration is coordinated with a token ring communication, in which every agent estimates their distance to all shops in the map. Then, they form a proposal containing all of these estimated distances to each shop. Figure 6 illustrates how the shop exploration strategy works.

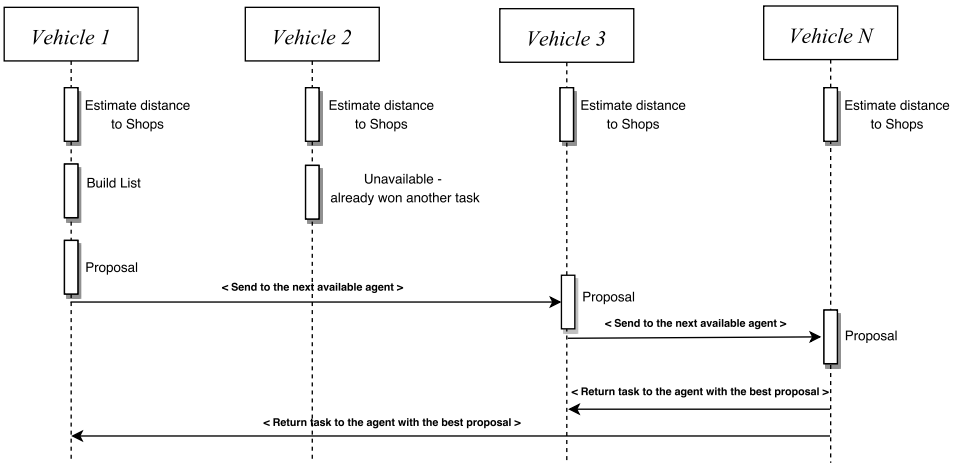

Figure 6 Shop exploration strategy overview.

Only one agent is sent to each shop, in order to minimise any unnecessary battery recharge costs. To decide which agent is sent to explore each shop, we implemented a token ring communication that can occur in several iterations. Each iteration, an agent sends its proposal to explore all available shops (i.e., shops that have not yet been assigned to be explored by another agent) to the next agent in the ring.

An iteration ends once the last agent has formed its proposal, which he then compares to all previous proposals before allocating the exploration of shops to the best proposals. If all shops have been allocated, then the allocation ends and each agent that committed to explore a shop starts to do so. If any shops remain not allocated, then another iteration begins containing only these shops.

4.2 Battery Recharge

Vehicles consume battery when moving in the city; in order to continue moving, they can recharge in any of the charging station available in the map. If their battery becomes empty, they can call the *breakdown service*, which fully recharges its battery after 25 steps. An agent must send the call breakdown service action to the server for each of the 25 steps in order for it to work. As calling this service takes time, and also costs money, we want to minimise these situations.

We implemented a strategy, described in Algorithm 1, which always try to foresee the need to recharge its battery. Whenever a vehicle has a task to go to some facility, it first

checks if its current battery level is enough to go to this facility, and then to the nearest charging station to that facility. Note that it is not enough to just check if the vehicle is able to reach the facility, because then nothing prevents it from not having enough battery to go to other facilities or charging stations.

Algorithm 1 Plan for going to a facility.

```
1: procedure goto(Facility)
2:     if enough battery to go to facility and from there to the nearest charging station then
3:         goto(facility)
4:     else
5:         closestChargingStation ← closestFromFacility
6:         if empty(closestChargingStation) then
7:             closestChargingStation ← closestToMyPosition
8:         end if
9:         if enough battery to go to closestChargingStation and then to facility then
10:            goto(closestChargingStation)
11:            goto(facility)
12:        else
13:            goto(facility)
14:        end if
15:    end if
16: end procedure
```

If the agent reaches the conclusion that it does not have enough battery, it tries to find a route to this facility that includes a charging station on the way for which he does have enough battery to get to. In order to choose the best charging station, that is, one that does not incur a great increase in the route length to the facility, we consider only the ones that are between the vehicle current position and the facility position. If there is no such charging station, we simply choose the closest charging station from the agent's current position. Now, again, the vehicle needs to check if it has enough battery to go the chosen charging station and then to the desired facility; if it has, then it simply follows this route.

However, there are cases where all of these options are not possible, and it is thus impossible for the agent to go to the facility without stopping on the way because its battery was emptied, even if it does recharge in between. In such cases, the vehicle tries to go to the desired facility anyway, calling the *breakdown service* when necessary. The complete code for our battery recharge strategy can be found in `common-strategies.asl` (lines 43 – 123).

4.3 Simultaneous Jobs

Besides the constraints we impose when deciding if our team is going to undertake a job, described in Section 4.4, we also limit the number of simultaneous jobs that our team can handle at the same time. We decided to use this strategy because, as the number of simultaneous jobs increases, and consequently the number of available agents decreases, these remaining agents tend to give worse job estimates. This occurs because faster vehicles, like drones and motorcycles, have a higher chance of giving better estimates than slower vehicles, like cars and, primarily, trucks. Therefore, when our team is already committed

to a given number of jobs, the set of available agents is mainly consisted of these slower vehicles, and it becomes no longer cost-effective to accept new jobs.

We empirically tested different limits for the number of simultaneous jobs; the limit of three jobs at the same time gave the best results with the configuration used in the simulations of MAPC 2016. When this limit is reached, new incoming jobs are dropped without further analysis; we only start to consider taking new jobs when we complete one of the simultaneous jobs that were in progress.

4.4 Job Evaluation

We consider job evaluation to be one of our major strategies that contributed to the good performance of our team in MAPC 2016. We identified two broader types of job evaluation: decide if a job could potentially reward more than it costs; and decide if a new job is more profitable than jobs that we were already pursuing. Because we have a limit of three simultaneous jobs (see Section 4.3), and because we believe that it is too costly to abandon a job, we focused on the first type of job evaluation. During the contest, we ignored *auction jobs* and concentrated only on *priced jobs*, simply because our code for auction jobs was not polished enough in time for the contest.

In Algorithm 2, we have a high-level version of a plan used by the initiator to evaluate new priced jobs. The complete code for priced job evaluation can be found in `initiator.asl` (lines 81 – 150). Our job evaluation consists of multiple conditional tests:

1. We only consider undertaking a new priced job if we are not in the *shop exploration* phase and if we are not in the *endgame* phase.

2. Our team commits to at most three simultaneous priced jobs. Should the team already be at maximum capacity, then the job is ignored. The job is also ignored if we do not have enough available agents for the total number of tasks in the job (i.e., number of different items that comes with the job's perception).

3. Agents in our team do not use assemble actions, as we noticed that buying any assembled item required for a job was faster than trying to assemble them. Thus, we ignore any job that contains assembled items that are not available in a shop. From our tests and matches in the MAPC 2016, we noticed that this would never happen, every item (including assembled items) was available in at least one shop.

4. Next, we calculate an estimate of how many steps it would take to complete the job. We calculate this step estimate by using the distance from the centre of the map, using a car (best results in our experiments), to the closest shop and to the storage that the job has to be delivered. Then, we add these two values and multiply by the number of tasks in the job, which should be equal to the number of agents that will work on the job (we consider the worst case scenario). We add 35 steps to our estimate to improve our chances of not undertaking jobs that could expire before we are able to finish them, and check if the result is lower than the step that the job expires, (`End`) minus the current step of the simulation.

5. The last conditional test is to check if the job offers more money than it costs (i.e., it is a profitable job). In order to check that, we calculate the cost of buying all the items required for the job, plus an estimated battery fee. The battery fee estimation is calculated by multiplying the previously calculated shop step estimate and the storage

step estimate by 10 (the cost of battery for every 1 step of using the `goto` action), then multiplying both values for the number of tasks, and finally adding them together. If the sum of the cost estimate and the battery fee estimate are lower than the money reward offered by the job, then the job passes our evaluation.

Algorithm 2 Plan for evaluating priced jobs.

```
1: procedure pricedJob(JobId, StorageId, Begin, End, Reward, Items)
2:     if not exploring shops and not endgame then
3:         if jobs committed < 3 and number of tasks <= available agents then
4:             verifyAssembled(Items, Assembled)
5:             if (Assembled = 0) or required assembled items are buyable at a shop) then
6:                 calculate_step_estimate(TotalEstimated)
7:                 if TotalEstimated + 35 < End - CurrentStep then
8:                     calculate_price_estimate(Cost, BatteryFee)
9:                     if Cost + BatteryFee < Reward then
10:                        separate_tasks(Items, JobId, StorageId)
11:                    end if
12:                end if
13:            end if
14:        end if
15:    end if
16: end procedure
```

As the map centre information is not promptly available, we calculate an approximation at the start of each simulation. In order to approximate it, we get the average latitude and longitude of existing shop locations. Although this does not give the exact centre of the map, empirically it resulted in a good approximation.

These estimates are rough projections, used to discard conspicuous bad jobs. Real calculations are done during task allocation, should the priced job pass our job evaluation. That is, our bid evaluation does not guarantee that a job is doable. We may still choose to ignore the priced job during task allocation if the job is impossible with the current configuration of available agents at that particular moment in the simulation.

4.5 Task Allocation

We wanted to integrate agent techniques in our strategies in order to demonstrate the advantages of using MAS, especially in a decentralised setting. The *Agents in the City* scenario has many opportunities to use such techniques. Task allocation is one of these opportunities, and has many uses in MAS. We choose to use Contract Net Protocol (CNP) [11] as the task allocation mechanism. Agents use CNP artefacts, described in Section 3.4, to announce new tasks and post bids to contracts.

Our implementation of CNP used during the contest has an important characteristic that makes it different from classical CNPs, the initiator is always the same agent (vehicle15, a truck). Although all agents are able to receive the perception that a new priced job was created, we did not find any particular reason for alternating initiators. However, we have some experimental code that allows bidders to become initiators if they need any help to

fulfil a contract, and plan on adding that feature when we include assemble actions, in future versions of our code.

Initiator

If a priced job has been evaluated and deemed useful, the initiator decomposes the job into several tasks. The number of tasks is equal to the number of different items that the job requests. Then, the initiator announces these tasks concurrently (Jason allows concurrent execution of the agents' intentions), one contract for each task. The deadline (2 seconds) starts counting from this point forward, and the initiator waits for it to close (either by deadline or by all agents placing their bids) before collecting the bids to all existing contracts.

The next step is to select the winner bid and award an agent with the contract. Whenever bidding is closed for a contract, the initiator increases a counter by 1 (starting from 0), and will only start to process the bids once the counter reaches the same number of announced contracts. For each contract, the initiator selects the lowest bid value and add that agent to a winner's list. If an agent is already on the winner's list, its bid will only be eligible to be selected for subsequent contracts if items from these contracts can be bought at the same shop of the first contract that it won.

Finally, when the winning bids of all contracts have been selected, the initiator sends a message to all agents in the winner's list, prompting their task execution plan (see Section 4.6). The number of jobs in progress is incremented, and the initiator goes back to its previous task. After the initiator has received confirmation that a contract has been fulfilled from all agents that were undertaking a priced job, it decrements the number of jobs in progress.

Bidder

As soon as a bidder agent perceives that new tasks have been announced, it starts preparing its bid for that task. Agents that are already undertaking another task, or are busy dumping items from failed job attempts, send a bid value of -1, which represents that it is ineligible for achieving the task (the initiator ignores all -1 bids). If the agent does not have enough load left to carry all of the task's items, it will also send a bid value of -1. Otherwise, if the agent is available, it calculates its bid. The value of the bid is calculated by the sum of the number of steps required to arrive at the closest shop that contains the items of the task and the number of steps from that shop to the storage that the task needs to be delivered.

Bidders that won a contract receive a message with the *winner* perception from the initiator. For all contracts that a bidder receives, it creates a *buyList* belief with the items, and their quantity, that it has to buy for that contract. Then, agents proceed to try and fulfil their contracts (see task execution in the next section). In Figure 7, we show the agent overview diagram for bidder agents, the capability overview diagram for bid procedure, and a corresponding (shortened) JaCaMo code.

4.6 Task Execution

When an agent is awarded with a contract, it activates the plan `go_work`, shown in Listing 1, from its plan library. We simplified some of the code to improve readability, the complete code can be located in `common-strategies.asl` (lines 15 – 41). We collect some beliefs from the agent's belief base in the context of the plan (line 2). Then, the body of the

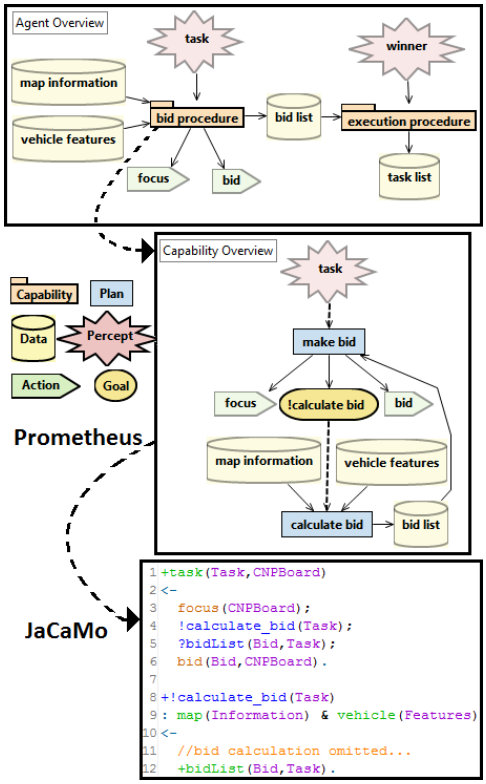

Figure 7 Bidder overview (image adapted from [4]).

plan begins with a subgoal to go to (see Algorithm 1 for more details) the shop specified in the contract.

An agent might have multiple contracts for items that can be bought in the same shop, thus, once it arrives in a shop, the agent will buy all items (from all of its contracts) that can be bought in that specific shop. The `while` loop is used to guarantee that the agent will keep trying to buy the item, should the buy action fail. The buy action can fail in one of two ways: the agent is trying to buy more items than the shop has available (shops restock each step, so it makes sense to keep trying to buy); or because of the 1% chance of random failure that all actions have.

After all items required for the contract have been bought, the agent proceeds to the storage specified in the contract. Similarly to the buy action, if the `deliver_job` action randomly fails, the agent will keep trying to deliver the job. The deliver job action can also fail if: the job has already been completed; or if the agent does not have the correct items for that job. Should the action fail for any of these two reasons, and it is still carrying any items, then the agent will go to a dump (see Section 4.7) facility to dispose of them.

Finally, the agent updates its load in the team artefact (Section 3.2), and makes itself available again by adopting the subgoal `free`. There are several plans for this subgoal, but, in summary, agents will keep sending `skip` actions to the server for as long as they have the `free` subgoal.

Listing 1 Plan for working a contract.

```
+!go_work(JobId,StorageId)
  : buyList(_,_,ShopId) & .my_name(Me) & role(_,_,LoadCap,_,_)
<-
  !goto(ShopId);
  for (buyList(Item2,Qty2,ShopId)) {
    while (buyList(Item,Qty,ShopId)) {
      !buy(Item,Qty);
      !skip;
    }}
  !goto(StorageId);
  !deliver_job(JobId);
  while (failed_random)
    !deliver_job(JobId);
  if (failed_job_status | useless)
    !go_dump;
  .send(initiator,tell,done(JobId));
  updateLoad(Me,LoadCap);
  !free;
.
```

4.7 Dump

As agents have limited carrying capacity, it is important to not let them carry superfluous items, in order to allow them operate at full capacity. When this happens the agent will go to the closest dump facility, and dump all items that it is currently carrying.

The most common cause for this to happen is if another team completed one job before us; in this case, when our agents try to deliver the job (i.e., deliver an item at a storage location), this action will fail (with status `useless` or `failed_job_status`). Another possible cause can be due to an unknown bug in our code, where the agent could not finish its current allocated task.

4.8 Endgame strategy

We implemented the *Endgame* strategy to limit the execution of some strategies in the last steps of a round in the simulation, in order to avoid wasting money with unnecessary actions. This strategy is referenced as *go horse* in our code, a fictitious development methodology that prioritises finishing the project at all costs.

During endgame, the initiator stops announcing any job, since agents will not have enough time to complete the necessary tasks. Therefore, only existing jobs from before the endgame will have its tasks being executed and eventually completed. To save money, agents also stop doing the `dump`. Because no further tasks will be allocated, there is no drawback in agents keeping their items from failed job attempts.

Our default strategy is to enable the *Endgame* strategy in the last 100 steps. However, in our first match with *Flisvos-2016* we detected that this made our agents stop too early (at step 916 they were done with all jobs from before the endgame); for the second match we decreased it to 80 steps, and for the last match we decreased it to 60 steps.

4.9 Post Jobs

We implemented a strategy to post jobs which are guaranteed to give a reward lower than what a team would spend to complete it. Teams that do not reason whether a job is worth its cost (i.e., perform some kind of job evaluation) could potentially accept our jobs and give us an advantage, since they would lose money trying to complete them. As we added this feature on the second day of the contest, we did not have time to implement a more complex strategy; furthermore, as we did not have another team implementation to test against, thus limiting our tests.

In order to avoid flooding the environment with jobs, we select only one agent to keep creating and posting jobs each step. This agent will only post jobs during the *Shop exploration* or the *Endgame* phases, and only agents who are not participating in these phases can be selected (usually trucks). Therefore not impacting its performance and readiness during the important task allocation and task execution phases. This is necessary because posting a job is an action, and we would not want to waste the actions of agents that are committed to completing a job.

We can post either a priced or an auction job, with equal probability. For the priced job we have a *Reward* of "1", and for the auction job we have a *MaxBid* of "1000" and a *Fine* of "1". This job is composed of $\lfloor (Number\,of\,products/3) \rfloor + 1$ product items. For each of these items we need to choose the number of required items of this type; in order to make this job worse, we select a number such that only a truck vehicle would be able to carry all of them at the same time. As we know that a volume of 600 to 900 will guarantee that only a truck would be able to carry everything, we select a random number between this range, and then calculate how many items are necessary to reach this value. Finally, we also choose a random storage for the job to be delivered; one example of a priced and an auction job created by our agent is shown in Example 4.1.

Example 4.1: *PricedJob (Reward=1;Item5(Qty=30);Storage=storage6)*
AuctionJob (MaxBid=1000;Fine=1;Item11(Qty=11);Item13(Qty=24);Storage=storage7)

4.10 Round Change

At each round we need to perform some setup and clean-up. For each agent we reset the charging, dump, storage, and shop facility lists, which will be populated when this information is received from the server when another round starts. We also remove all current beliefs and drop all intentions, desires, and events, using Jason internal actions.

To prepare for the next round we update the map to be used (*london*, then *hannover*, and finally *sanfrancisco*). Close to the contest date we noticed that we had a bug in the map change function of our `MapHelper` that would unable the use of `MapHelper` to find routes when a round changed. We were not able to fix this bug in time for the contest, and thus, we had to reconnect our agents between each round, setting the correct map manually.

5 Result Analysis

In this section, we analyse the performance of our team in two of our matches in the contest. First, we analyse some issues (e.g., connection problem and bugs) that we had in the last round of the match against team *lampe*. Second, we discuss the three rounds of the match

against the most competitive team that we have faced during the contest (*Flisvos-2016*), in which we had our only loss.

5.1 PUCRS vs. lampe

During the third round against *lampe*, a bug occurred that made our agents send skip actions for most steps of the simulation, yet we manage to fix it just in time to win. Figure 8 shows how the money of each team (*PUCRS* vs. *lampe*) fluctuated during the simulation steps. While our team sacrificed a lot of money in the first steps to profit later, *lampe* remained stuck at 50000. Their agents got disconnected and were not able to reconnect in time, thus, resulting in `noAction` from all agents in team *lampe*.

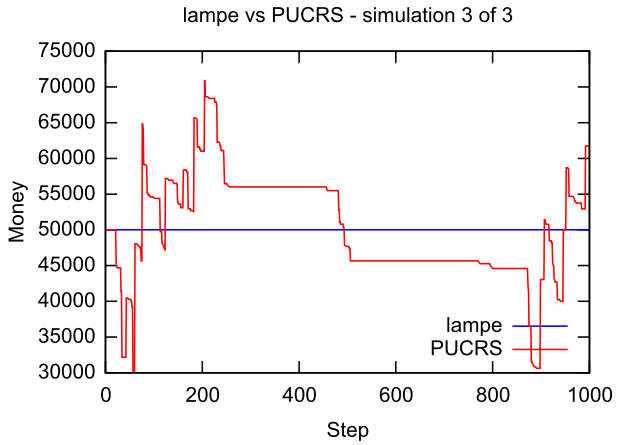

Figure 8 *PUCRS* vs. *lampe*, round 3 out of 3.

Our bug happened around step 260, when some agents alternated their `continue` action with a `skip` action. The continue action is only useful when executed after a `goto`, `charge`, or another `continue` action that preceded either goto or charge. When executed after a skip action, it only repeats the skip. This caused several of our agents to get stuck, they stopped and skipped an action while going somewhere else, and were not able to recover. Because of our limit of three simultaneous jobs, eventually agents from each of our three active jobs got stuck, and that led our whole team to execute skip actions.

We believe that the skip action while moving was sent as a countermeasure to some `noActions` that happened with our agents, which in turn, we suspect were originated from low internet bandwidth at the time of the round. Although we could have won at that time, we decided to reconnect our agents to see if that fixed our bug. Unfortunately, it just made it worse, since some agents were still carrying items for previous jobs, which they did not know how to get rid of (we had not implemented our dump strategy yet), and then started accepting new jobs and ultimately got stuck again. Fortunately, we managed to implement a simple dump strategy (while the simulation was still running) and reconnect our agents at around step 890, which then took around 60 steps for our team to turn the game and eventually win by a margin of 11000.

Even though we were basically playing alone, should our team finish under 50000, which team *lampe* scored by simply not moving, we would have lost and it would probably cost

us our first place. Thus, our on-the-fly implementation of a simple dump strategy allowed our team to come back from defeat in the last 100 steps, and secure a win which should have been easy if not for that bug.

5.2 PUCRS vs. Flisvos-2016

We lost the first round against *Flisvos-2016* for two reasons:

- due to a bug with our recharge strategies where two agents got stuck moving between two charging stations and would never move to their final destination, causing a lock of one of three simultaneous jobs (from that point forward we were only able to do two simultaneous jobs);
- and due to the use of our *Endgame* strategy, in which our agents stopped doing any action that would cost money (e.g., buy items, battery charge, and others) near the end of the simulation.

Namely, our agents stopped trying to do new jobs at step 916, while *Flisvos-2016* has continued working, and eventually surpassed us at step 961, as shown in Figure 9a. Despite being a competitive first round in which our team kept a lead most of the time, we eventually stabilized and lost during the last few steps.

Subsequently, in the second round against *Flisvos-2016*, both teams stopped delivering jobs near step 930, as shown in Figure 9b, but this time we win as the recharging bug did not happen. Although we controlled most of this second round, *Flisvos-2016* did some impressive jobs near the end, and almost managed to surpass our team.

Differently from previous rounds against *Flisvos-2016*, our team dominated the third and final round, winning with a high margin of money, as shown in Figure 9a. This could be the result of our team simply doing better jobs in this third round, but we may also have missed the occurrence of a bug in round 2. Because of the results in the first round, we lowered a bit the number of steps from the end of the round that would activate our *Endgame* strategy. This was immediately employed for the second round (another on-the-fly change), and third round, when we lowered it a bit more.

6 Team Overview: Short Answers

6.1 Participants and their background

What was your motivation to participate in the contest?

Our motivation was to improve our knowledge about agent technologies and to put them into practice in a complex multi-agent scenario. We also wanted to compare and check how effective a JaCaMo implementation could be against other agent development platforms.

What is the history of your group? (course project, thesis, . . .)

This was a side project to everyone in our team. As much as we wanted to add our PhD and MSc techniques to our agents, the scenario was simply too complex to add yet more layers of complexity on top of it. Furthermore, the deadline for each step would severely limit the usage of our approaches (e.g., multi-agent planning).

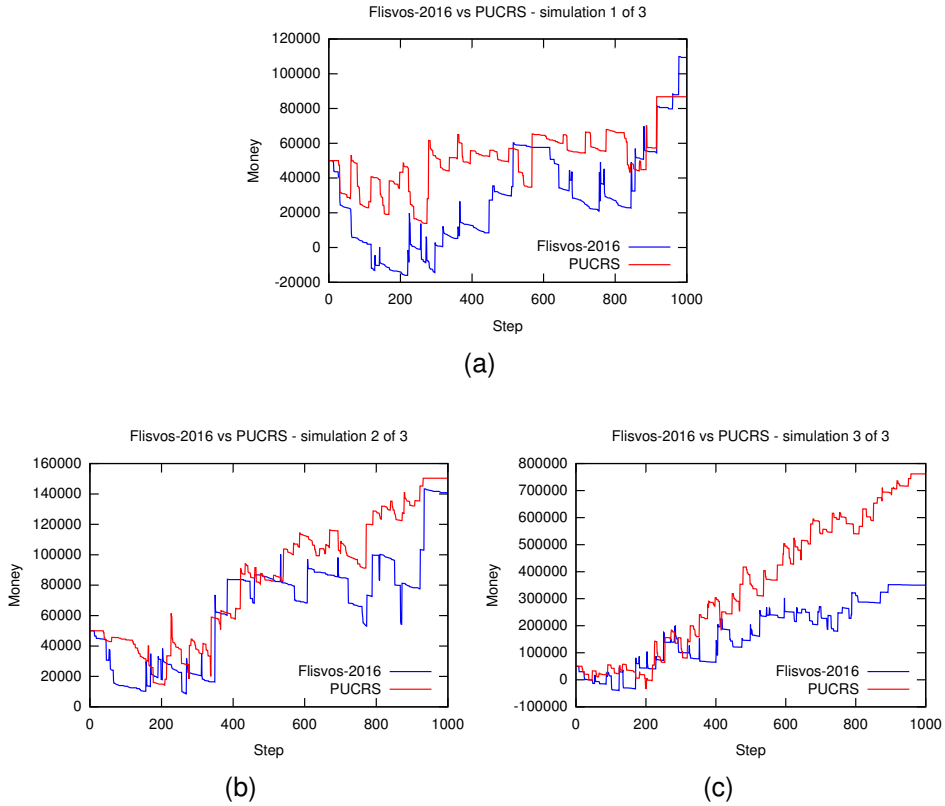

Figure 9 (a) *PUCRS* vs. *Flisvos-2016*, round 1 out of 3; (b) *PUCRS* vs. *Flisvos-2016*, round 2 out of 3; (c) *PUCRS* vs. *Flisvos-2016*, round 3 out of 3.

What is your field of research? Which work therein is related?

All of our members are from Computer Science - Artificial Intelligence. Research topics include but are not limited to: Automated Planning, Multi-Agent Planning, Plan Recognition, Machine Learning, and Task Allocation. We used the JaCaMo (see Section 3) platform to develop our agents, which some of us were already using in our research.

6.2 The cold hard facts

How much time did you invest in the contest (for programming, organizing your group, other)?

Approximately 252 man hours. 156 in 2015 and 96 in 2016. Versions from 2015 and 2016 differ significantly, moving from 2015's purely reactive plans to a mixture of proactive and reactive plans.

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

2191 total. 1166 lines of agents' code, 118 of configuration, and 907 of Java code. Measurements were taken by using Linux `wc` command plus `sed` for removing blank

lines. Please note that comments and extra lines that we used to improve code readability were not removed in this count.

How many people were involved?

A total of 13 people were involved. 11 team members and 2 collaborators. We had at most a group of 4 to 6 people active in any given week, who would then cycle in and out with others.

When did you start working on your agents?

We started working on our agents on May of 2015. A significant shift in the way that we dealt with the environment and in the way our agents reasoned happened at the end of 2015, but we only started implementing these new changes around August of 2016. Instead of using one artefact that would communicate with the environment, receive actions and send perceptions to agents, we added one artefact for each agent in order to decentralise the information and execution loops. The biggest change was related to the way that we were writing agents' plans. Initially, we made them entirely reactive to each step of the simulation, but after some suggestions from one of the developers of JaCaMo, we completely overhauled our plans to be declarative, making our agents a lot more proactive while still keeping some elements of reactivity.

6.3 Strategies and details

What is the main strategy of your agent team?

Our main strategy is task allocation (see Section 4.5). By using contract net protocol the agents themselves can decide who is best for the tasks that a job requires.

How does the team work together? (coordination, information sharing, ...)

Coordination in our shop exploration strategy is decentralised using a token-ring message passaging mechanism. Coordination for job allocation and execution is based on contract net protocol. An initiator is responsible for evaluating jobs, announcing tasks, and allocating tasks to bid winners. We also use an artefact that all agents can access to share important information that is unique to them and otherwise unattainable.

What are critical components of your team?

The main components of our team are: each agent's environment artefact, the contract net protocol artefacts, and the team artefact. The main strategies employed by our team are: shop exploration, task allocation, and battery recharge.

Can your agents change their behaviour during runtime? If so, what triggers the changes?

Our agents can change their behaviour based on the step that the simulation is currently in, after a specific strategy concludes, or upon receiving feedback from the last action that was executed. Initially we ignore all jobs, agents decide who will explore shops, while idle agents post jobs and send skip actions. After shop exploration has ended, agents now have access to all items' prices, and thus they can start evaluating new jobs. If a job failed during a deliver action, agents dump their items before trying to accept new jobs. Close to the end of the match, agents adopt the `Endgame` strategy stance (see Section 4.8), which limits some of the plans that they can use.

Did you make changes to the team during the contest?
During the contest we did mostly bug fixes and optimisation of parameters/options. We did implement the dump strategy (see Section 4.7) for the second day of the contest which we did not have ready for the first few rounds of day one. We also added a simple post job strategy (see Section 4.9 for the second day.

How do you organize your agents? Do you use e.g. hierarchies? Is your organization implicit or explicit?
A truck agent act as the initiator. The rest of our organisation is defined implicitly by contract net bid winners'.

Is most of your agents' behaviour emergent on an individual or team level?
Mostly individual, although we do have a few team aspects for control purposes, such as which agents are awarded with more than one task, and which ones are free to take on new jobs. For example, a bidder agent might be eligible to win more than one contract, which can occur concurrently, but the initiator agent will make sure that it only wins more than one iff the bidder agent is able to carry all necessary items, and iff these tasks have the same destination.

If your agents perform some planning, how many steps do they plan ahead?
Our agents plan their routes before placing their bids for tasks. We developed experimental code that can plan the full route, including any recharging stops, but the code was not ready in time for the contest. Instead, we use an estimate of the route that ignores the need to recharge battery for the agent's bid. Since JaCaMo does not have any lookahead planning mechanism [12], we used the GraphHopper library to plan our routes in advance.

If you have a perceive-think-act cycle, how is it synchronized with the server?
Synchronisation is done through each agent artefact (see Section 3.1), on a step-by-step basis. These artefacts receive all of the perceptions from the server, filter the ones related to their agent, send them to it, the agent then reasons about it, chooses an appropriate course of action, send the action to their artefact, who finally sends it to the server.

6.4 Scenario specifics

How do your agents decide which jobs to fulfill?
The contract net initiator first checks if the number of tasks (different items required by the job) is smaller than or equal to the number of available agents (agents without a job). It proceeds to test if all assembled items can be found in at least one shop. Then, the initiator evaluates if the number of steps that the job will be active is within a calculated estimative. After that, it estimates the cost of doing the job, and if that estimated cost is less than the reward offered, the job will accepted (read more details in Section 4.4).

Do your agents make use of less used scenario aspects (e.g. dumping items, putting items in a storage)?
Our agents use dump (see Section 4.7), call service breakdown (see Section 4.2), and post jobs (see Section 4.9) actions.

Do you have different strategies for the different roles?
Roles have an impact in each agent's bid during task allocation for jobs. Roles that move faster tend to have a lower route, making their bid more favourable, as long as they can carry all items needed for the task. We also decided to use a truck as the initiator for our contract nets.

Do your agents form ad-hoc teams for each job?
The team responsible for a job is formed of all contract net winners related to that job. These agents should, in theory, be the best ones available for the job.

What do your agents do when they do not pursue any job?
They can explore shops, dump items, and post jobs. Although if an agent is not pursuing a job, we prefer them to stay still to save battery (money) while waiting for a new job.

6.5 And the moral of it is …

What did you learn from participating in the contest?
Coordinating a development team with more than 4 people can be very difficult. The different number of perspectives are most of the time very advantageous, but can make it difficult to choose a specific approach to implement. We have strengthened our knowledge of agent technologies, and many of us have learned several new things from our strategy discussions.

What are the strong and weak points of your team?
We believe our strong points are how we evaluate potential jobs (see Section 4.4), the use of contract net for task allocation (see Section 4.5, and our battery recharge strategy (see Section 4.2). Our weak points are the round changing bug, the battery recharge bug, no use of assemble actions, and the fact that we ignore auction jobs.

How viable were your chosen programming language, methodology, tools, and algorithms?
The Prometheus methodology was useful in providing an overview of our system before and during development. We did not use it as much in later strategy development stages as a result of conceptual divergences between Prometheus and JaCaMo (see [4]). We did not have any major problems programming our agent team in JaCaMo, everything we wanted to do was possible and very easy to program, despite some difficulties with debugging the interaction between agents and artefacts, that were the source of some of our bugs. We did not use Moise, the organisation layer of JaCaMo, because we did not have enough time to model an organisation for this scenario, but we believe this may be possible and potentially useful in future agent teams. The use of Github for source code management, access control, and collaboration features such as bug tracking and wikis, made it a lot easier to manage our team more efficiently.

Did you encounter new problems during the contest?
We encountered more problems than we expected to. There are a lot of random elements in this scenario, making it harder to properly test all of our team's code. We managed to fix most of them between matches (sometimes even between rounds), while for the most complex ones we had to develop workarounds (see Section 5).

Did playing against other agent teams bring about new insights on your own agents?
We were more occupied in looking out for possible bugs in our code, and trying to fix them, instead of analysing the behaviour of other teams. We did notice that some teams had very active agents, which made us try a few adjustments such as increasing the number of simultaneous jobs, and lowering some of our job estimates.

What would you improve if you wanted to participate in the same contest a week from now (or next year)?
A week from now: focus on fixing bugs, and perhaps tweaking some of our parameters and options. Next year: definitely develop an assemble strategy that works, start to consider auction jobs, and possibly distribute our agents in several machines (1 to 1 would be great).

Which aspect of your team cost you the most time?
Switching from a purely reactive code to a more proactive one cost us a lot of time. The scenario had a few changes as well (new actions added, bug fixes, parameters changed) that also had an impact. From our strategies, the use of contract net protocol and our battery recharge strategy cost us the most time.

What can be improved regarding the contest/scenario for next year?
Teams that do assemble will always lose to teams that do buy only, since buying is much faster than assembling (once a team delivers a priced job, the opposing team cannot complete it any more). We hope this is fixed for next year, since assemble is a much more interesting problem for agents to solve. A simple solution would be to have some jobs with assembled items that could not be bought from any shops. Perhaps by adding a random parameter with a small chance for assembled items to spawn (or to not spawn) in shops.

Why did your team perform as it did? Why did the other teams perform better/worse than you did?
We believe our job estimates, simultaneous job execution, task allocation, and our battery recharge strategy were the deciding factors in how we won first place. Most teams had bugs in their code (we also had several), some were more severe than others. We have reasons to believe that the only match we lost was due to a bug in which two of our agents got stuck for many steps in a recharge loop between two charging stations (see Section 5).

7 Conclusion

The Agents in the City scenario provided us with a complex challenge, one that put many of our skills to the test, such as team work, agent-oriented programming, and decentralised techniques for coordination of agents. As such this project allowed team members with previous experience with JaCaMo development to improve their proficiency in JaCaMo programming, as well as provided a working introduction to the platform for the team members without such background.

JaCaMo was used by the winner of the last two previous MAPCs, in 2013 [13] and 2014. Ours was the third consecutive win of a team that uses JaCaMo to program their agents. We

were able to intuitively program all of our strategies and take advantage of the agent and environment dimensions available in JaCaMo. However, we faced some difficulties when we had to debug our code, and sometimes we had to change our approach completely in order to avoid the bug instead of trying to fix it, since we could not find any reliable way of fixing it. This could be, partially due to some of our inexperience with JaCaMo programming, and partially due to the complexity of the scenario, but nevertheless, JaCaMo's debugging features could be improved, especially the link between agents and artefacts.

Although we developed a strategy for evaluating and placing bids to auction jobs, as well as a strategy for determining if our team should try to assemble items instead of buying them, we were unable to fix some game breaking bugs in time to include them for the contest, and thus, we do not describe them in this paper. For future contests we plan on adding these strategies, as well as fixing the bugs that we found in our current strategies during the contest's matches, such as the bug in our round change and battery recharge strategies.

Acknowledgements

We are grateful for the support given by CAPES and CNPq. Túlio thanks the support given by the Federal Institute of Rio Grande do Sul (IFRS) – Campus Feliz. We would like to thank the contributions given by fellow teammates: Alison Roberto Panisson, Anibal Heinsfeld, Artur Freitas, Guilherme Azevedo, and Tabajara Krausburg. The team also extends their thanks to all the JaCaMo developers (Olivier Boissier, Rafael Bordini, Jomi Hubner, Alessandro Ricci), especially to Rafael and Jomi for their feedback.

References

[1] Tobias Ahlbrecht, Jürgen Dix, and Niklas Fiekas. Multi-agent programming contest 2016. *Int. J. Agent-Oriented Software Engineering*, X(X):X, X.

[2] Lin Padgham and Michael Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley & Sons, Inc., New York, NY, USA, 2004.

[3] Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with jacamo. *Science of Computer Programming*, 2011.

[4] Artur Freitas, Rafael C. Cardoso, Renata Vieira, and Rafael H. Bordini. Limitations and divergences in approaches for agent-oriented modelling and programming. In *Workshop on Engineering Multi-Agent Systems (EMAS-16)*, Singapore, 2016.

[5] Daniela Maria Uez and Jomi Fred Hübner. Environments and organizations in multi-agent systems: From modelling to code. In *2nd International Workshop on Engineering Multi-Agent Systems*, pages 181–203, 2014.

[6] Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.

[7] Anand S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world*, MAAMAW '96, pages 42–55, Secaucus, NJ, USA, 1996.

[8] Alessandro Ricci, Michele Piunti, Mirko Viroli, and Andrea Omicini. Environment programming in CArtAgO. In *Multi-Agent Programming: Languages, Tools and Applications*, Multiagent Systems, Artificial Societies, and Simulated Organizations, chapter 8, pages 259–288. Springer, 2009.

[9] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, 2008.

[10] Jomi F. Hübner, Jaime S. Sichman, and Olivier Boissier. Developing organised multiagent systems using the moise+ model: programming issues at the system and agent levels. *Int. J. Agent-Oriented Software Engineering*, 1(3/4):370–395, 2007.

[11] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Comput.*, 29(12):1104–1113, December 1980.

[12] Felipe Meneguzzi and Lavindra De Silva. Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. *The Knowledge Engineering Review*, 30:1–44, 1 2015.

[13] Tobias Ahlbrecht, Jürgen Dix, Michael Köster, and Federico Schlesinger. *Multi-Agent Programming Contest 2013*, pages 292–318. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.