

Multi-Agent Systems

Helge Hatteland
s144457

Oliver Fleckenstein
s144472

DTU



Kongens Lyngby 2017

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

The goal of the thesis is to develop and implement a full-scale multi-agent system targeting a realistic and nondeterministic environment. The environment is provided by the organizers of the annual Multi-Agent Programming Contest (MAPC), having created a scenario which takes place in several of Europe's famous cities, namely *Agents in the City*. The scenario is related to each team being a contractor, having a set of vehicles at their disposal to solve jobs that involve acquiring, assembling and delivering items at specific locations.

The system is developed using Jason integrated with CArtAgO, often referred to as JaCa, while the agents are implemented using the agent-oriented programming language AgentSpeak.

The purpose of this project is to familiarize the reader with the development of multi-agent systems, solving non-trivial coordination tasks and learning about the challenges that arise in complex nondeterministic environments. The second purpose is to successfully implement one or more solutions to the MAPC, and find good metrics for testing and evaluating these on their own and against each other.

Preface

The thesis was prepared at DTU Compute in fulfillment of the requirements for acquiring an BSc in Engineering (Software Technology).

The project was conducted in 2017 from February 25th to the 1st of July for 15 ECTS points, with Jørgen Villadsen and John Bruntse Larsen as supervisors.

The thesis concerns the development and evaluation of a multi-agent system using Jason in combination with the CArtAgO framework. The multi-agent system targets the 2017 Multi-Agent Programming Contest scenario, *Agents in the City*.

Familiarity with logic programming, such as Prolog, is an advantage when reading the report. Knowledge about artificial intelligence is also helpful, although the necessary concepts are introduced in the thesis.

Acknowledgements

We would like to thank our supervisors Jørgen Villadsen and John Bruntse Larsen from DTU Compute for guidance throughout the project, providing helpful feedback and support.

We would also like to thank Lars Fleckenstein and Sara Bjørnevik for extensive proofreading and feedback.

Contents

Summary	i
Preface	iii
Acknowledgements	v
1 Introduction	1
1.1 Aim and Scope	2
1.2 The History of AI	2
1.3 Thesis Structure	3
2 Multi-Agent Systems	5
2.1 Introduction	5
2.2 Definitions	5
2.2.1 Environment	6
2.2.2 Intelligent Agent	7
2.3 Agent-Oriented Programming	8
2.3.1 Logic Based Agents	8
2.3.2 Reactive Agents	10
2.3.3 Belief-Desire-Intention	11
2.4 AgentSpeak and Jason	13
2.4.1 Jason Programming	14
3 CArtAgO	19
3.1 Introduction	19
3.2 Artifact-Based Environments	20
3.3 Standard Artifact Operations	22
3.3.1 Creating and Discovering Artifacts	22
3.3.2 Using and Observing Artifacts	22

3.3.3	Linking and Unlinking Artifacts	23
3.4	Artifact Programming	23
3.4.1	Integration of CArtAgO	25
3.4.2	Security Aspects	27
3.5	Application to Multi-Agent Systems	27
3.5.1	Agent Coordination	27
3.5.2	Task Synchronization	28
3.5.3	Resource Programming	28
4	Problem Analysis	29
4.1	The Multi-Agent Programming Contest	29
4.2	Scenario Description	30
4.2.1	Jobs	30
4.2.2	Items	31
4.2.3	Roles	32
4.2.4	Facilities	32
4.3	MASSim Environment	33
4.3.1	Actions	34
4.3.2	Percepts	36
4.3.3	Properties	37
4.4	Problem Simplifications	38
4.5	Choice of Solution	39
5	Implementation	41
5.1	Development Process	41
5.2	Server Communication	41
5.3	Percept Filtering and Organizing	42
5.4	Agent Logic	43
5.4.1	Rules	44
5.4.2	Plans	45
5.5	First Iteration	47
5.6	Contract Net Protocol	47
5.7	Second Iteration	49
5.8	Third Iteration	50
5.9	Communication Protocols	51
5.9.1	Give/Receive Protocol	51
5.9.2	Assemble Protocol	51
5.10	Fourth Iteration	52
5.11	Fifth Iteration	54
5.12	Evaluating Jobs	55

6	Results	57
6.1	Evaluation	57
6.2	Configuration	58
6.3	Solution 1	58
6.4	Solution 2	59
6.5	Solution 3	60
6.6	Solution 4	61
6.7	Comparison	63
6.8	Solution 5	64
6.9	Job Evaluation	66
6.10	Matches	67
7	Discussion	73
7.1	Pitfalls of AgentSpeak	73
7.1.1	Performance Issues	73
7.1.2	Delayed Percepts	75
7.2	Side-Effects of CArtAgO	76
7.2.1	Percepts and Perceiving	76
7.2.2	Operation Semantics	77
7.2.3	Operation Invocation	77
7.3	Future Work	78
7.3.1	Freeing Resources	78
7.3.2	Competition Application	79
8	Conclusion	81
A	Benchmark Data	83
B	Multi-Agent Environment Details	89
B.1	Environment Actions	89
B.2	Environment Percepts	97
C	Source Code	105
C.1	Jason MAS Configuration File	105
C.2	Jason Source Code	106
C.3	Java Source Code	116

CHAPTER 1

Introduction

A multi-agent system is comprised of several interacting agents, observing and acting upon an environment. By coordinating and synchronizing intelligent agents, it is possible to find solutions to problems that an individual agent or a monolithic architecture cannot. Intelligent agents are often characterized by the properties of *autonomy*, *proactiveness*, *reactivity* and *social ability* [?]. As a result, the individual agents are capable of goal-based reasoning, while being able to coordinate their efforts toward a common solution and react to sudden changes.

Implementing efficient reasoning and coordination is however often considered the most challenging task concerning multi-agent systems, hence many tools, architectures and other technologies have been introduced to facilitate the development of such systems. These include the Belief-Desire-Intention (BDI) architecture, the AgentSpeak programming language which builds upon it, the implementation of AgentSpeak called Jason, and the JaCa extension, which is an integration of the CArtaGo (Common ARTifact infrastructure for AGents Open environments) framework into Jason. By exploiting these tools and the concepts that follow, an attempt will be made to develop a highly sophisticated multi-agent system capable of solving non-trivial tasks which rely on a high level of coordination.

The multi-agent system described in the following chapters targets a predefined

nondeterministic environment, provided by the organizers of the annual Multi-Agent Programming Contest (MAPC). The scenario, *Agents in the City*, is related to each team being a contractor, having a fleet of vehicles at their disposal to move around a city and solve jobs that involve acquiring, assembling and delivering items at specific locations. All of these jobs requires several agents to work together.

1.1 Aim and Scope

The overall aim of the project is to gain extensive knowledge in terms of developing and implementing multi-agent systems capable of competing in the MAPC. The market for artificial intelligence, and as an extension multi-agent systems, is growing rapidly, with many predictions about what the future of these fields will bring. Everything from expert systems able to diagnose patients [?], to multi-agent systems controlling airport baggage transportation [?] are already affecting peoples everyday lives. No matter what, artificial intelligence have and will continue to have a huge influence on the world, being the reason for this choice of project.

One of the initial project goals was to use the Jason implementation of AgentSpeak, which called for further research on technologies and frameworks building on this. As JaCaMo was the platform used by the last year's winners of the Multi-Agent Programming Contest this was one of the chosen areas of focus. JaCaMo is a combination of **Jason**, **CARTAgO** and **Moise**, where Moise is used for programming multi-agent organizations. While organizations are not first priority, CARTAgO on the other hand, is used for programming environment artifacts. Artifacts are entities shared among the agents in an environment, and can be highly relevant for e.g. coordination purposes. As a result, CARTAgO became a part of the project, making the development easier by exploiting tools that already exist.

1.2 The History of AI

The field of AI, or artificial intelligence, research was founded as an academic discipline at the Dartmouth Conference in 1956. The proposal for the conference built on the basis of the conjecture that: *every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it* [?]. The conference is widely renowned for giving birth to AI,

giving it a name, a purpose and its first success.

The following years were an era of discovery and most people were simply astonished by the capabilities of AI. This included computers that could solve algebra word problems, prove theorems in geometry and learn to speak English. The optimism flourished and million dollar grants were received. There were however limitations, and in the mid 70s AI research were subject to financial setbacks, due to among other issues, limited computational power, intractability of problems and *Moravec's paradox*. The latter entails how computers are efficient at solving tasks which humans consider to be complicated or difficult (e.g. proving theorems), while failing at tasks humans consider to be intuitively simple (e.g. recognizing a face) [?].

After some ups (1980-1987) and downs (1987-1993) the field of AI has become half a century old and has finally achieved some of its oldest goals, being successfully applied throughout the technological industry. While this has not necessarily been a result of higher intelligence, it has been a combination of more computational power, better algorithms and deductive reasoning. Today, even faster computers are accessible, advanced machine learning techniques, not to mention the vast amounts of data, i.e. *big data*. By 2016, the market for AI related products, including hardware and software, reached more than 8 billion dollars [?], being yet another success for the field of AI, and according to the predictions, certainly not the last.

1.3 Thesis Structure

The thesis is structured as follows:

- Chapter 2 gives an introduction to multi-agent systems and provides relevant background knowledge.
- Chapter 3 introduces the CArtAgO framework from both a theoretical and practical viewpoint.
- Chapter 4 describes and analyzes the Multi-Agent Programming Contest scenario, identifying key problems.
- Chapter 5 explains the development and the implementation of the project.
- Chapter 6 displays the results from the test simulations, and compares the different solutions against each other.

- Chapter 7 discusses the experiences developing and implementing the solution.
- Finally, Chapter 8 concludes the project.

CHAPTER 2

Multi-Agent Systems

2.1 Introduction

In general, multi-agent systems are used to model and control environments, often simulating the real world, where more than one agent plays a role. While the only difference from single-agent problems is the fact that there are multiple agents, it calls for new interesting and complex problems, such as coordination and task sharing. While many single-agent problems (games, pathfinding, and more) have been solved at a better-than-human level, multi-agent scenarios still provide big challenges, e.g. agent coordination.

2.2 Definitions

Before going in detail about multi-agent systems, it is important to define what an *agent* really is, and why it is considered intelligent. There is no universally agreed upon definition of an agent, since different problems require domain-specific elements to be accurately modelled. However, the most abstract concept of an agent, can be seen in Figure 2.1. The agent is able to perform *actions* on the environment using its actuators, while the changes in the environment are

perceived through its sensors. The internals of the agent define how the agent reacts to different percepts, and thereby which actions it should perform next [?].

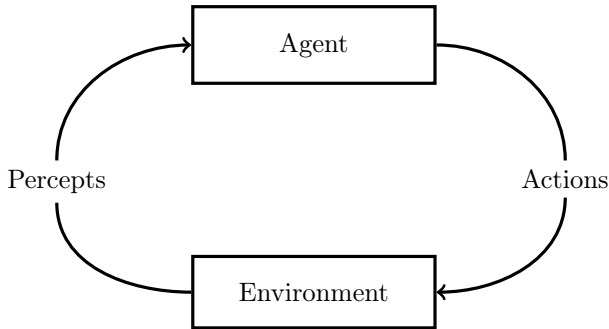


Figure 2.1: A model of an agent and its environment. An agent acts upon the environment through actions, and perceives the state of the environment through percepts.

2.2.1 Environment

The environment in this figure is also rather abstract. In the field of AI, environments can model entirely different domains, having several different properties. The properties a problem can have are divided into the following categories:

- *Single-Agent vs Multi-Agent:* When modelling environments, they can either be regarded as single-agent or multi-agent. This distinction might seem simple enough, but there are some things to consider. What should be viewed as an agent? In a self-driving car, should the car consider other cars as agents, or simply as objects in the world that move around? It might be enough to model other cars as object, and simply design the self-driving car to avoid them, but one could also argue that all cars are agents. By defining them as agent, cars could communicate and coordinate their driving, improving traffic overall.

Agents in a multi-agent environment can either be *competitive* or *cooperative*. In games like chess, each player (agent) is trying to win the game, therefore they are competitive. In a situation with multiple self-driving cars, they should try to work together to minimize the risk of collision, making them cooperative.

- *Partially Observable vs Fully Observable:* An environment is fully observable if an agent at any time can see the entire state of the world. It is often

more convenient to work with a fully observable environment, as there is no need for the agent to keep an internal state of the world. Partial observable environments are missing some of this information, either because of noise or missing percepts. An environment can also be *unobservable*, meaning that the agent knows nothing about the environment.

- *Static vs Dynamic*: If an environment stays the same unless an agent acts upon it, the environment is considered static. If the world can change without the influence of an agent, it is considered dynamic. A static environment is easier for an agent to work with, as it does not have to continuously observe the environment to know which state it is in.

In multi-agent environments, the environment can be static, but look dynamic from each agent's point of view. While an agent is considering what to do, other agents might act on the environment and change it.

- *Episodic vs Sequential*: In an episodic environment each state does not depend on the previous one. In other words, agents' actions does not have any permanent effect on the environment, and they can essentially restart themselves in each step. In sequential environments on the other hand, each step depends on the previous one, entailing that agents' actions affect future states of the environment. As a result, agents might have to make plans for the future to reach their goals.
- *Deterministic vs Stochastic*: In a deterministic environment each action will only have one effect on the environment, which might be known before executing the action. If an action has multiple possible outcomes, the environment is said to be stochastic. This brings uncertainty into the problem. Actions will often have a probability assigned to each possible outcome to model this. When the environment is stochastic, agents will have to be able to adapt and reconsider their plans depending on the actual outcomes of their actions.
- *Discrete vs Continuous*: A discrete environment is limited to a fixed set of actions, percepts, and states. Continuous environments on the other hand can have any number of these. Many games can be modelled as discrete problems, while real world problems are often continuous.

2.2.2 Intelligent Agent

The abstract concept of an agent does not describe how to handle all the different properties of an environment. An intelligent agent is defined having the following properties:

- *Autonomy*: An intelligent agent should have a notion of autonomy, e.g. being able to autonomously choose which actions to perform to reach a specific goal.
- *Proactiveness*: The agent has to be proactive, meaning that it must consider the future and make plans for how to achieve its goals. By making compositions of plans, the agent is able to solve more complex problems and reach high-level goals, not necessarily achievable by a single action.
- *Reactivity*: The agent also has to be reactive, being able to perceive the environment and respond accordingly. Often, this must be done fast, as changes in the environment could potentially be harmful to the agent. For example with self-driving cars, regardless of its initially planned trajectory, if a person steps in the way, it should do something to avoid him. However, if agents only react to changes, they will not be able to achieve long term goals, so this must work in balance with being proactiveness.
- *Social ability*: Lastly, intelligent agents should have a social ability, i.e. the ability to communicate with other agents or objects. By understanding what other agents' goals are, the agents can cooperate to efficiently solve all their desired goals. However, the agents should be able to distinguish cooperative agents from competitive ones, realizing if they want to help or cause damage.

2.3 Agent-Oriented Programming

So far, agents have simply been considered as a function, receiving input from the environment (percepts) and returning output (actions). There are four classic ways to model the internals of agents: logic based, reactive, belief-desire-intention (BDI), and layered architectures. The main focus will be on the BDI model, as this is the leading approach to agent programming and also the model used in this project. However, as the BDI model builds on top of the previous approaches, these will also be mentioned to give the full picture of agent-oriented programming. Given that the layered architecture is not used in the project, it will not be explained in-depth, other than it can be thought of as a combination of the three other techniques, arranged in a layered structure.

2.3.1 Logic Based Agents

The traditional approach to agent-oriented programming was to use logic in decision making. This can be done by representing the world through a symbolic

representation of the environment and the agent's goals. Then the representation can be manipulated until the goal state is achieved. Each predicate would be some belief about the world, e.g. the agent's location or the temperature of the room. This manipulation is done using logic deduction, which finds the applicable actions that proves the correct solution.

To illustrate how this representation works, consider a grid world in which an agent moves around and can vacuum the cells. Each of the cells in the world is represented symbolically in logic by a $Cell(x,y)$ predicate, where x and y is the coordinates of the cell. The location of the agent is represented similarly by $In(x, y)$, and dirty cells is represented by $Dirt(x, y)$. The set of actions which the agent can perform would be $A \in \{MOVE(D), SUCK\}$, where $D \in \{NORTH, SOUTH, EAST, WEST\}$.

Each action will have a set of preconditions which has to be satisfied before the action is valid and can be executed, along with a set of postconditions, i.e. the effects on the world. The $MOVE(D)$ action would have the preconditions that the agent believe it has a location, and there exist a cell in the direction D from this location. The postconditions would be the change of the agent's beliefs such that the old $In(x, y)$ predicate is removed, and a new one is added with the updated location. The agent can only have one In belief at any time, as it can only be at one place at any time. The action scheme for $MOVE(NORTH)$ can be seen in Figure 2.2 as an example.

ACTION : $Move(North)$
 PRECONDITION : $In(x, y) \wedge Cell(x, y + 1)$
 POSTCONDITION : $\neg In(x, y) \wedge In(x, y + 1)$

Figure 2.2: An example of the MOVE action with the direction NORTH. Similar actions can be defined for all the other directions.

If the agent is currently standing in a cell containing *Dirty*, the SUCK action can be used to remove this. The action scheme for this can be seen below. From this scheme, if the agent is not currently standing on a dirty cell, this action is not applicable. However, one could argue that the cell does not need to be dirty, for the agent to be able to perform the SUCK action. The *Dirty* predicate from the precondition could be removed, allowing the agent to do a SUCK action in any cell. As the only effect of the suck action is to remove the *Dirty* predicate from the current cell, doing SUCK on an already clean cell does not change the state of the environment, and can therefore be considered equivalent with doing nothing.

The goal for this vacuum agent is to remove all the *Dirt* predicates from the

ACTION : *Suck*
 PRECONDITION : $In(x, y) \wedge Dirty(x, y)$
 POSTCONDITION : $\neg Dirty(x, y)$

Figure 2.3: The SUCK action, which will clean the cell which the agent is currently standing on.

world. For the agent to find a solution, it needs a set of rules to guide it towards its goal. For example a rule like: if I am standing in a cell and that cell is dirty, do SUCK, written as: $In(x, y) \wedge Dirty(x, y) \rightarrow SUCK$. If this is not the case, the agent should do a MOVE in any direction in order to move itself onto a dirty cell. How the agent will move around can be implemented in many different ways, using different strategies. For example, one approach be to check if there is a dirty cell north of the agent, and move north if this is the case. This could be expressed as:

$$\exists x_0, y_0, x_1, y_1 (In(x_0, y_0) \wedge Dirty(x_1, y_1) \wedge y_1 > y_0) \rightarrow MOVE(NORTH)$$

Similar rules for other direction can be made as well. Another strategy is to visit all cells systematically, or even just do a random walk until all cells are clean. When the agent has reached a world state where it believes all cells are clean, the agent has achieved its goal.

One of the issues with the logical approach, is that the problems quickly become intractable when complexity increases. Because it relies on simple rules, it requires many iterations to find solutions to more complex problems. This limitation led to new approaches to agent programming, which did not rely as heavily on logic and deduction.

2.3.2 Reactive Agents

One of the properties of the intelligent agent defined earlier, is that the agent should be able to react to the environment. Developing such agents is facilitated by integrating a reactive architecture. The approach is built upon the idea, that intelligence is linked to the environment in which agents live and act, and that intelligence emerges from various simpler rules about how an agent should react.

There are two characteristics that distinguish this agent architecture from the others. The first is its decision making process, which is defined through be-

havioral rules. Instead of taking all the information perceived about the environment into consideration, the agent has rules to handle every single percept. It has a SEE function to perceive the environment, a set of rules, and an ACT function to perform actions on it. The rules will map every output from the SEE function to an action, which will be given as a input to the ACT function. This avoids any complex symbolic representation of the world, and avoids all complex reasoning and deliberation. Thereby, agents can be easily developed creating rules like *percept* \rightarrow *action*.

The second characteristic is the way this architecture handles multiple percepts at the same time. Doing so can cause multiple rules to be applicable in a given situation. To avoid this, rules are organized into a hierarchy, meaning some rules should always be prioritized over others, and therefore its action should be executed first. For instance, if the agent is in some catastrophic situation, it should try to solve this first, before trying to achieve its goals.

However, even though this approach does handle some problems well and can be useful for many agent systems, especially in episodic environments, it has trouble with solving complex tasks which requires long-term planning. By only reacting to the environment as it is, it does not consider how the future will look, and does therefore not create plans for how to achieve its goals.

2.3.3 Belief-Desire-Intention

The belief-desire-intention (BDI) architecture is an attempt at modelling practical reasoning. It builds upon the logical and reactive approaches, which by themselves both had some faults. It tries to balance the proactive (goal directed) and reactive (event driven) elements of reasoning, using the best parts of the two previous approaches. This is done by taking the simplicity of representing the world using logic, and combining this with rules from the reactive approach. This allows for a simple way of modeling the environment, while being able to react quickly to sudden changes. Figure 2.4 illustrates how the BDI architecture is linked together.

Beliefs

The BDI architecture uses *beliefs* to model the state of the environment, similar to how the logical agent architecture does. The agent perceives the environment, receiving percepts and revise these using a *belief revision function* (BRF). The BRF maps the percepts into new beliefs, merging them with the old ones and removing incorrect or outdated beliefs as a result. For instance, if the agent perceives its location, previous beliefs about its location should be discarded.

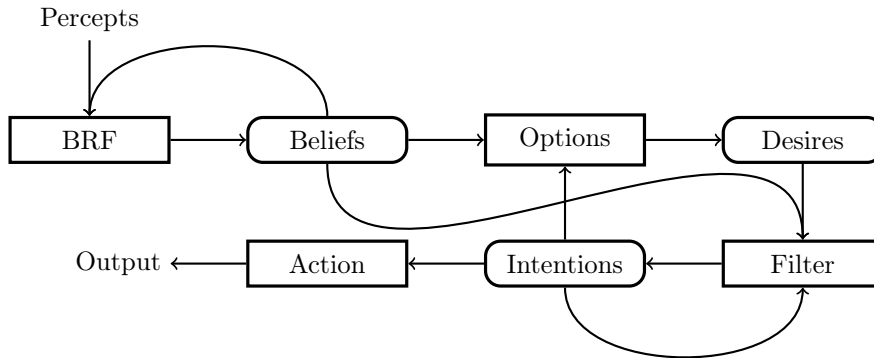


Figure 2.4: Overview of the BDI model and its functions. Illustrates the connections between the individual elements in the architecture.

Desires

By evaluating the agent's current beliefs, together with its current intentions, an *option* generating function is used to determine which options are available to the agent. These options represent the *desires* the agent wants to accomplish. While desires cannot be acted on by themselves, an agent can commit to fulfill one or more of its desires, thus becoming *intentions*.

Intentions

An intention is an actual goal, which the agent can attempt to achieve. This is done by finding relevant *plans*, being the recipes on how to solve goals. Each plan comprise a sequence of actions (or other plans) and a specific goal being achieved by following the plan. As a result, complex goals can be solved by creating compositions of several plans.

Deliberation

By utilizing a *filter* function, the agent is able to decide which desires to commit to. The function takes all the agent's current beliefs, desires and intentions into account, evaluating which of the intentions to proceed with. It should be able to continuously act towards an intention, but at the same time, be able to dynamically change which intention to pursuit. If the intention is accomplished, becomes impossible to achieve, or if the intention is no longer a desire, it should be dropped. This is the biggest challenge with the BDI architecture - finding the perfect balance between continuing with and reevaluating intentions.

If the agent reevaluates its intentions too often, it wastes resources that could otherwise have been spent on figuring out how to fulfill the intentions. On the other hand, if the agent never reevaluates them, it may end up pursuing

intentions, which given the current state of the environment, are no longer relevant or can be achieved in a more efficient manner. How often an agent should reevaluate its intentions depends on the dynamics of the environment, and on how reactive the agent has to be.

The last step in the BDI architecture is the *action* function, determining which action to perform in order to fulfill its current intention. The action is selected according to the available plans, while plans are selected according to their specified preconditions. Finally, the agent uses its actuators to perform the given action on the environment.

Once the new state of the environment is perceived, the entire deliberation process restarts with the agent's revised beliefs, desires, and intentions. This procedure will continue until no new desires are generated.

2.4 AgentSpeak and Jason

The *AgentSpeak* programming language is designed to develop multi-agent systems using the BDI model. It was originally designed as an abstract agent-oriented programming language, used to understand and describe agents with a BDI architecture formally.

Jason is an implementation of the abstract AgentSpeak language in Java. The language allows for an easy way to represent beliefs in logic, and to describe plans for how to achieve intentions. The agent's BDI architecture is hidden behind the scenes, implicitly adding the behavior of perceiving, updating the belief base and automatically evaluating desires and intentions. This allows for an easy and fast way of setting up a multi-agent system.

While the Jason language does provide the BDI behavior out of the box, many of the components can be changed or extended using Java to fit the developers need. The *Agent* class is exposed for developers to extend and overwrite functions as needed. For instance, the intention selection function, *selectIntention*, defaults to polling intentions from a queue, where the first element will always be the current intention. If the developer wants to prioritize specific intentions, this could be done by changing how it polls intentions from the queue. The other functions, such as the *option* function or *belief revision function*, can also be extended on demand.

Jason is also very powerful in the way it allows agents to communicate with each other. By using the internal *send* action, agents can easily share their beliefs, ask

for information or plans for how to achieve intentions, or even ask other agents to achieve one of their desires by sending a message. These functionalities are distinguished by a keyword, specifying one of the following: **tell**, **askOne**¹, **askHow** or **achieve**. The sender must also specify the receivers of the message with a name, a list of names, or simply use the internal *broadcast* action to send the message to every agent in the system.

2.4.1 Jason Programming

Jason's syntax is based on logic, and share a lot of characteristics with the logic programming language Prolog. Like Prolog, Jason has literals that are known to be true or false, and has rules to evaluate more complex queries.

Beliefs

In Jason beliefs are described through the use of literals. Continuing from the example introduced in subsection 2.3.1 with the vacuum agent, the environment is described as follows.

```
cell(0,0).  
cell(0,1).  
cell(1,0).  
cell(1,1).  
  
in(0,0).  
dirty(1,1).
```

In this example, the agent believes that there are four cells, and that it is located in cell (0,0). In addition it also has the information that cell (1,1) is dirty. Jason will always start from the top of the source file when it searches for a belief, and will stop searching when the first match is found (the same is the case when searching for plans and rules). Jason also supports negation of literals, which can make agents believe that something is explicitly false. This is done with the \sim operator.

For example, if the agent cleans a cell, it could either just remove the dirty literal, as it no longer believes the cell to be dirty, or it could explicitly express that it believes the cell not to be dirty. This could be written as \sim dirty(1,1), when cell (1,1) has been cleaned. It is not always necessary to use this negation operator, but there might be a difference from believing that a cell is not dirty, and not knowing anything about the state of the cell. For example, in partially

¹This will return the first answer to the question. Jason also support an **askAll**, which returns all the possible answers to a query.

observable environments, agents might have to explore the cell before knowing if the cell is dirty or not.

Jason also has negation as failure. By using the **not** operator, agents can check if something is not in its belief base. For example, **not in(1,1)** means the agent does not believe it is in (1,1). Note that this is different from explicitly believing that the agent is not in (1,1), as the \sim operator expresses.

Annotations

Beliefs can also have annotations, e.g. used to describe where the information came from. By default, beliefs have a **source** annotation. This tells the agent if the belief was perceived, sent from another agent, or added by itself (referred to as a mental note). Any information can be added as annotations. An example of this is a timestamp for when the belief was added. Annotations can be written after the belief using [], e.g. **in(0,0)[source(percept)]**.

Belief annotations are not necessarily just information-holders, and the agents' architecture can be extended to provide the semantics. As an example, an **expires** annotation can be introduced to remove beliefs upon adding others, resolving contradictions in the belief base. For instance, consider two beliefs **summer** and **fall**. While it does not make sense for the agent to believe both to hold at the same time, the first belief can be set to expire in case of the second belief. Using the new annotation, this can be achieved by replacing **summer** with **summer[expires(fall)]**.

Other than the beliefs' annotations, there are a couple of annotations used for plans, including **atomic** and **priority**. The **atomic** annotation ensures that once the plan is selected for execution, it will be executed in subsequent reasoning cycles until it is finished. The **priority** annotation on the other hand, allows the developer to specify a priority among other plans, prioritizing the execution of the one with the highest priority.

Rules

Rules are used to infer information based on the agents' beliefs, and to introduce more complex preconditions in the plans' context. Rules in Jason are written using the **:-** operator. Consider the following example.

```
man(david).
woman(suzie).
parent(david, suzie).

father(X, Y) :- parent(X, Y) & man(X).
```

The agent believes that **david** is a man, **suzie** is a woman, and that **david**

is a parent to **suzie**. By using the **father(david, suzie)** rule, the agent is able to conclude that **david** is the father of **suzie**. This example also introduces variables to Jason. Anything starting with an uppercase character will be considered a variable in Jason, just like in Prolog. The use of variables allows for more general and reusable rules to be created. When testing **father(david, suzie)**, Jason will unify **X** with **david** and **Y** with **suzie**, continuing to evaluate whether **max(david** is also the case. If all clauses evaluate to true, the result of the test itself evaluates to true. If the arguments are switched around, testing **father(suzie, david)**, the rule evaluates to false, since neither **parent(suzie, david)** nor **man(suzie)** is found in the agent's beliefs.

Goals

What have been presented so far has all been standard to logical programming languages. But Jason also includes a notation for *goals* and *plans*, which as mentioned earlier is essential for agent programming. There are two types of goals in Jason: *achievement goals* and *test goals*. Achievement goals describe something the agent wants to achieve, and is written using the **!** operator. This could for example be **!own(house)** or in the vacuum world **!clean(1,1)**. These are regarded as desires by the BDI model, being something the agent wants to achieve.

Test goals on the other hand, are used to retrieve information from the agent's belief base. This is done with the **?** operator. Considering the previous example, it might be interesting about whom **david** is a father of, thus testing **?father(david, Child)**. Similarly, it could test **?woman(Person)** where in both cases, the variables **Child** and **Person** will unify with **suzie**. Test goals are more or less the application of rules and beliefs, where the unification of variables is more interesting than the evaluation to true or false.

Plans

Plans are very essentials to Jason, being used to achieve goals. A plan is comprised of the following components:

- a goal - the postcondition of the plan;
- a context - the precondition of the plan; and
- a body - the sequence of actions to carry out.

In Jason this is represented using the following syntax:

```
+!goal : context <- body.
```

Continuing in the vacuum agent world, the following code shows a plan for solving the goal `clean`, which includes solving another goal `go(1,1)` followed by performing a suck action.

```
+!clean <- !go(1,1); suck.
```

The previously mentioned plan does not specify a context, thus it is always applicable. Since it is not relevant to go to a cell the agent is already in, the plan can be extended to the following:

```
+!clean : in(1,1) <- suck.  
+!clean <- !go(1,1); suck.
```

If the agent is not already located in (1,1), it will skip the first plan and use the other. Note that the order of which the plans are written matters, since Jason always chooses the first applicable plan from the top of the source file.

Events

When a new belief is added to an agent's belief base, it will fire a *triggering event*. These events can be handled by prefixing the + operator to the literal. The following is an example of such an event:

```
+hello <- .print("Hi").
```

Jason also include the - operator, which can be used to handle failures. If it is impossible to find any applicable plans for a goal, or if the plan fails during execution, this failure event will be triggered. Failure events are used for error handling, or to continue on an alternative path.

```
+hello : false <- .print("Hi").  
-hello <- .print("Bye").
```

In this example, triggering `hello` will not find a valid event due to the inclusion of `false` in the event's context, thus failing and printing `Bye`.

Lists

The last thing to mention is lists. Lists in Jason are similar to those in Prolog, and consist of a head and a tail. The head is the first element in the list, and the tail is the remaining elements. An empty list is written as `[]`, and the first element can be accessed by using the `|` to split the head element from the tail. The best way to explain this is to show the *member* rule, checking if an element is in the list. This rule succeeds if the first element is equal to the element being

checked, and otherwise recursively call the rule on the rest of the list. The rule will fail when this list is empty, and it can no longer match any of the cases.

```
member(Head, [Head | Tail]).  
member(X,    [_ | Tail]) :- member(X, Tail).
```

This illustrates how a list can be iterated and how to access each element. Lists are an essential data structure, used extensively in general logic and Jason programming.

CHAPTER 3

CArtAgO

3.1 Introduction

CArtAgO (Common ARTifact infrastructure for AGents Open environments) is a general purpose framework facilitating the programming of environments for multi-agent systems. The framework is based on the Agents & Artifacts (A&A) meta-model for modelling and designing multi-agent systems, introducing *agents*, *artifacts* and *workspaces* as a first-class abstraction.

The A&A abstractions are taken from human cooperative working environments where: the *agents* are the computational entities performing goal-oriented activities, *artifacts* are the resources and tools which can be dynamically constructed, used, and manipulated by agents, and finally *workspaces* are used to structure artifacts which agents from various platforms can join to access.

As a result, CArtAgO provides a simple programming model to design and implement agent computational environments, which is not bound to any specific agent model or platform. However, the framework is especially effective when integrated with agent programming languages based on a strong notion of agency, and in particular those based on the BDI architecture, e.g. Jason.

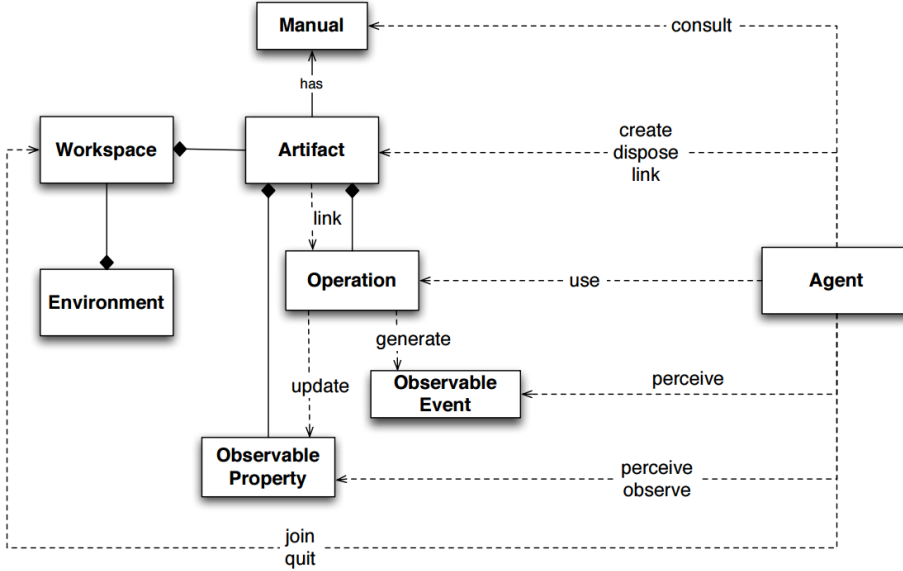


Figure 3.1: A&A meta-model (from [?, p. 8]). Illustrates how an agent can interact with artifacts.

3.2 Artifact-Based Environments

Figure 3.1 provides an overview of the main concepts characterizing artifact-based environments. The environment is composed of a dynamic set of artifacts, which agents in the environment can share and exploit. These artifacts can be organized in one or more workspaces, representing resources with different functionality. From an object-oriented viewpoint, artifacts in multi-agent systems are analogous to classes, defining the structure and behavior of the concrete instances. However, to make its functionality available and exploitable by agents, the artifacts rely on operations and observable properties.

Operations represent computational processes executed inside artifacts which can be triggered by either agents or other artifacts. These operations are used to retrieve and manipulate data in the artifact, and generate observable events the agents can react to. Observable properties represent state variables, whose value is perceivable by agents observing the artifact. While observing an artifact, the agent will be notified when the value of an observable property is updated, and when the execution of an operation generates a signal. By doing so, agents can respond to the specific changes in the environment they are interested in.

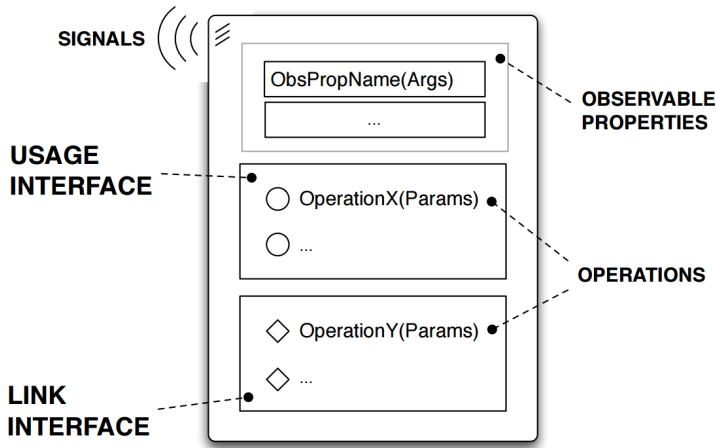


Figure 3.2: Abstract representation of an artifact (from [?, p. 9]).

While the overall set of artifact operations available to agents comprise the *usage interface*, linked artifacts can trigger the execution of operations in one another being exposed in the *link interface*. This allows for a separation of concerns without affecting usability, where the only requirement is that an agent links the artifacts beforehand. After doing so, link operations may be executed similar to operations executed by agents, allowing the realization of distributed environments where the linkability extends through different workspaces in different network nodes.

For an agent to know what functionality an artifact provides and how to exploit its usage interface, the artifact can be equipped with a manual. The manual is a machine-readable document to be consulted by agents, which is particularly useful in open systems where the agents dynamically decide which artifacts to use according to their goals, and dynamically discover how to use them.

Artifact operations represent external actions provided to agents by the environment. As a result, the repertoire of external actions available to an agent is defined by the set of artifacts that populate the environment. Given that the set of artifacts can be changed dynamically by agents themselves, instantiating new artifacts or disposing existing ones, the actions repertoire can be dynamic as well. By dynamically selecting which properties and events to observe, the artifact-based environment's complexity is reduced opposed to e.g. directly modelling percepts inside agents as beliefs.

3.3 Standard Artifact Operations

This section gives a short introduction to the artifact primitives provided by the CArtAgO framework. These primitives can be categorized into three main groups: creating and discovering artifacts, using and observing artifacts, and finally linking and unlinking artifacts.

3.3.1 Creating and Discovering Artifacts

Artifacts are meant to be a dynamic component in the artifact-based environment, thus it should be possible to create, discover, and dispose artifacts at runtime. To do so, three primitives are provided: `makeArtifact`, `lookupArtifact` and `disposeArtifact`. The `makeArtifact(ArtifactName, ArtifactClass, InitParams)` method instantiates a new artifact with the given `ArtifactName` of type `ArtifactClass` using the provided `InitParams`.

Agents in the appropriate workspace are then able to retrieve the artifact's id by using `lookupArtifact(ArtifactName)`, being its unique identifier. Furthermore, the artifact id can be used when executing operations (explained in the following section) and when disposing artifacts. To dispose an artifact, the `disposeArtifact(ArtifactId)` primitive is used, removing the artifact with the specific `ArtifactId` from the workspace.

3.3.2 Using and Observing Artifacts

An artifact can be used in two ways, either by executing its provided operations or by perceiving its observable properties and events. To use an artifact, an agent can simply call an operation by its name together with its associated signature, and then the target artifact can be inferred based on this. However, in scenarios where multiple artifacts in the workspace define operations with identical names, or the workspace simply contains multiple instances of the same artifact, an artifact id has to be provided to specify the target. For instance, if `ArtifactId` is a variable containing the target artifact's id, and `exampleOp` is the name of the operation to execute, in Jason this would be achieved using the following annotation: `exampleOp[artifact_id(ArtifactId)]`.

To observe an artifact, CArtAgO provides a `focus(ArtifactId)` primitive, which allows the agent to receive notifications when a signal is generated or an observable property is updated. Opposite to `focus`, `stopFocus(ArtifactId)`

is provided to stop observing an artifact. As a result, agents are able to dynamically choose what to perceive based on the artifacts they observe, and they are able to automatically update their beliefs about the environment. Signals on the other hand, are not related to observable properties, and act like messages which are processed asynchronously on the agent side.

An agent is also able to retrieve the value of an observable property without actually observing the artifact. This is done using `observeProperty(PropertyName)`, returning the value of the specific property as feedback. By using this primitive, the agent does not have to be continuously aware of the state of an artifact, while still being able to access its state when needed.

3.3.3 Linking and Unlinking Artifacts

Agents are able to link two artifacts together, where the first artifact is the linking artifact and the second artifact is the linked one. By doing so, the linking artifact is able to execute operations on the linked artifact. As a result, simple artifacts can dynamically be composed into more complex artifacts, creating whole networks of artifacts which can be distributed among several workspaces. To support linking of artifacts, two primitives are provided: `linkArtifacts(LinkingArtifactId, LinkedArtifactId)` and its opposite `unlinkArtifacts(LinkingArtifactId, LinkedArtifactId)` which respectively links and unlinks two artifacts.

3.4 Artifact Programming

CARTAGO provides a Java-based API for programming artifacts and artifact-based environments. This API simply utilizes Java classes and basic data types, for instance is an artifact created by defining a Java class extending the `cartago.Artifact` library class. By doing so, the defined class inherits the expected structure and behavior of an artifact. Artifacts are however not constructed like ordinary java Objects, but uses an optionally definable `init` method to initialize its fields and observable properties.

Observable properties are defined through `defineObsProperty`, taking a key-value pair where the key specifies the name of the property, and the value its initial value. An observable property can then be retrieved and modified using `getObsProperty` and `updateObsProperty` respectively. The artifact's fields can be used to define internal non-observable variables.

Operations are defined by methods annotated with `@OPERATION` and `void` return type, using its method parameters as both input and output operation parameters. Output parameters are represented by the parameterized class `OpFeedbackParam`, whose value should be set during execution of the operation.

Examples of these features can be seen in Listing 3.1, which is an example of a simple counting artifact. The artifact defines an observable property `count` during initialization and provides two operations, one for incrementing the count, and one for retrieving it. When the `incCount` method is executed, all agents observing the artifact will be notified of two events; that the value of the observable property `count` has been updated, and the signal `tick`. In AgentSpeak, this corresponds to the triggering events `+count(X).` and `+tick.` respectively.

```
public class Counter extends Artifact {

    void init() {
        defineObsProperty("count", 0);
    }

    @OPERATION void incCount() {
        ObsProperty prop = getObsProperty("count");
        prop.updateValue(prop.intValue() + 1);
        signal("tick");
    }

    @OPERATION void getCount(OpFeedbackParam<Integer> count) {
        count.set(getObsProperty("count").intValue());
    }
}
```

Listing 3.1: Artifact Example

Furthermore, operations can be composed of one or more atomic computational steps, where the execution of atomic steps inside an artifact is mutually exclusive. This allows for implementing long-term operations and can be used as an efficient coordination mechanism. To break down the execution into multiple steps, the API introduces `await` together with guards. Guards are defined by methods annotated with `@GUARD` and `boolean` return type. A guard can then be used as a condition in an `await` statement as shown in Listing 3.2.

```
public class Guarded extends Artifact {

    private int internalCount;

    @OPERATION void guardedOp(int waitCount) {
        internalCount = 0;
        signal("start");
        await("countGuard", waitCount);
    }
}
```

```

        signal("complete");
    }

    @GUARD boolean countGuard(int waitCount) {
        return internalCount >= waitCount;
    }

    @OPERATION void helpOp() {
        internalCount++;
    }
}

```

Listing 3.2: Guard Example

Operations with the `@OPERATION` annotation constitute an artifact's usage interface. Similarly, the operations annotated with `@LINK` constitute its link interface. Operations are defined identically regardless of annotation, only difference being the operation's scope and how it is executed. There is however a third type of operation, which can not be found in any of the artifact's interfaces. These operations are annotated with `@INTERNAL_OPERATION` and can only be executed by the artifact itself, or from another operation. Internal operations are executed asynchronously and are therefore very useful for background tasks or timers.

3.4.1 Integration of CArtAgO

CArtAgO has been developed with respects to orthogonality in terms of different multi-agent system technology, facilitating the integration of any agent programming language and platform. As a result, agents implemented using different programming languages, different technologies and running on different platforms can work together in the same multi-agent system by sharing common artifact-based environments. In other words, CArtAgO allows for creating heterogeneous systems, by simply extending the various agents' action repertoire with those provided by the artifacts.

Given that these actions are available to agents across multiple platforms, they can be considered as an external component. This also applies to the agents' percepts, having the extension of observable properties and signals generated by the artifacts. Although dependent on the agent programming language, observable properties are mapped into beliefs about the state of the environment, while signals are mapped into beliefs about the occurrence of observable events. The concrete realization can of course vary, but their semantics remain the same.

To demonstrate the integration of CArtAgO into an agent programming language, two Jason agents are defined below which use the previously mentioned

artifacts to coordinate their behavior. As a purely illustrative example, one agent (A) wait for a specific count to start a **guardedOp**, while the other agent (B) increments the count until the **guardedOp** starts, and subsequently executes **helpOps** until it is complete.

Listing 3.3: agentA.asl

```
{ include("focusArtifact.asl") }

!testArtifacts.

+!testArtifacts <-
  makeArtifact("myGuarded","Guarded");
  !focusArtifact("myCounter").

+tick <-
  .print("tick perceived").

+count(X) : X = 3 <-
  .print("starting guarded op");
  guardedOp(X);
  .print("guarded op complete").
```

Listing 3.4: agentB.asl

```
{ include("focusArtifact.asl") }

!testArtifacts.

+!testArtifacts <-
  makeArtifact("myCounter","Counter");
  !focusArtifact("myGuarded");
  while (not stopInc) {
    incCount; .wait(10);
  }.

+start <- +stopInc; !helpWithOp.
+complete <- +stopHelp.

+!helpWithOp : stopHelp.
+!helpWithOp <-
  .print("helping with op");
  helpOp; .wait(10);
  !helpWithOp.
```

Listing 3.3 and Listing 3.4 show the implementation of the two agents A and B respectively. Agent A executes **guardedOp** when a count of 3 is received, denoted by the precondition **X = 3**, which is also the amount of **helpOps** required to complete the **guardedOp**. **focusArtifact.asl** is included in both agents, which simply contains plans for looking up and focusing artifacts (Listing 3.5). Agent B is the one doing most of the operations, and reacts to the signals sent by agent A's execution of the **guardedOp**.

When a **+start** event is triggered, the agent stops incrementing the count and starts executing **helpOps** until the **+complete** event is triggered. At this point the **guardedOp** is complete, and the agents have successfully coordinated their actions. As a final note, the **.wait(10)** actions are necessary to allow agent A to respond in time. The result of executing these two agents in a default environment can be seen in the following output:

```
Jason Http Server running on http://192.168.0.15:3273
[agentA] tick perceived
[agentA] tick perceived
[agentA] tick perceived
[agentA] starting guarded op
[agentB] helping with op
[agentB] helping with op
[agentB] helping with op
[agentA] guarded op complete
```

As expected, the count is incremented three times before the `guardedOp` starts, while three `helpOps` are required for the `guardedOp` to complete.

```
+!focusArtifact(Name) <-
  lookupArtifact(Name, Id);
  focus(Id).
-!focusArtifacts(Name) <-
  .wait(10);
  !focusArtifacts(Name).
```

Listing 3.5: `focusArtifact.asl`

3.4.2 Security Aspects

Given that artifacts can be dynamically created and disposed by agents across distributed networks of workspaces, it is important to be able to set some constraints. This is achieved by having each workspace adopt a Role-Based Access Control mechanism, specifying roles for different agents and which actions different roles may or may not execute. These roles and policies are created and modified using one of the default workspace artifacts called **security-registry**, accessible by both human administrators and agents themselves. For instance, two roles could be defined, **admin** and **user**, where **admins** may dispose artifacts and **users** may not, thus preventing unauthorized agents from performing harmful actions. Similarly, the access to the workspace itself can be restricted, defining roles who may or may not join the workspace.

3.5 Application to Multi-Agent Systems

The CArtaGo framework introduces additional options for solving the typical problems that arise during the development of multi-agent systems. This includes coordinating agents, synchronizing tasks, and programming resources, which the following sections will explain how CArtaGo handles.

3.5.1 Agent Coordination

Agent coordination is one of the main aspects to developing multi-agent systems, which is why CArtaGo provides tools to ease the process. By exploiting the fact that artifacts are shared and concurrently used by agents, they can be used as coordination mechanisms. An example of such coordination was shown in

subsection 3.4.1, where two agents are able to coordinate their actions by the use of two artifacts. Coordination artifacts are especially useful to adopt an objective approach, where the coordination policies are defined by a state and a set of rules which should be encapsulated and separated from the agents.

3.5.2 Task Synchronization

A barrier synchronization mechanism is an example of an objective coordination artifact, where the state is the number of agents waiting at the barrier, and the set of rules are e.g. how many agents must arrive before they are released, or how many agents are released at the time. This can be achieved by having a single **synch** operation, which halts the agents' execution until a condition is fulfilled, after which the guard releases the agents. By implementing such an artifact, task synchronization can be ensured without having to rely on communication protocols, and without requiring agents to be aware of one another.

3.5.3 Resource Programming

Artifacts can not only be used to encapsulate a state and a set of rules, but any kind of data or functionality, allowing the dynamic creation of resources. These resources can first of all be data containers, e.g. implementing an artifact as a database with insert and select operations. Secondly, an artifact can provide new functional capabilities, e.g. simulating a software library by extending an agent's action repertoire with the artifact's usage interface. Doing so results in a high degree of reuseability (reusing the artifacts wherever needed), flexibility (using artifacts among heterogeneous agents) and dynamic extensibility (creating and disposing the artifacts at runtime). At the same time, by executing computationally heavy operations on the artifacts, agents are relieved of this burden, allowing for even increased agent performance.

Problem Analysis

4.1 The Multi-Agent Programming Contest

The system developed in this project targets the annual Multi-Agent Programming Contest (MAPC).¹ The competition was established to stimulate research within the field, and does so by creating problems that engage people. Also because the competition is easily accessible it allows for anyone interested to test their capabilities.

The competition has been held every year since 2005, and during these twelve years, the competitors have had to tackle five different scenarios. The last two years the challenge is named *Agents in the City*. This simulates several agents moving around on a map, earning as much money as possible. In order to maximize the earning potential, the agents will have to work together by the means of coordination. The simulation will run for a specific number of steps, usually 1000, and the winner of the simulation is whoever has the most money at the end.

¹The contest can be found at <https://multiagentcontest.org>

4.2 Scenario Description

The simulation takes place in one of Europe's famous cities, e.g. London or Rome. Throughout the map, several types of facilities and vehicles are placed randomly, where each vehicle is an agent. The colors of the vehicles denote their team, and the type of vehicle their role. There are several types of facilities, denoted by a pin on the map, where each facility has a different color, symbol and purpose. Figure 4.1 shows a snippet of the scenario taking place in Paris.

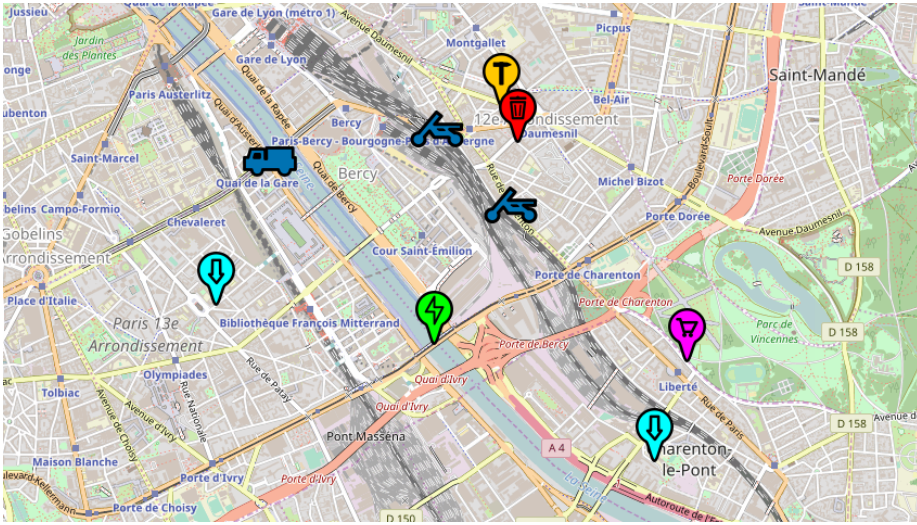


Figure 4.1: Snippet of the scenario

4.2.1 Jobs

To earn money, the agents have to complete jobs. The server will post different jobs when the simulation is running, which teams are able to solve. All jobs comprise acquiring, assembling and delivering some items to a specific facility before a deadline. Which items and how many depends on the job, as does the delivery location. There are four types of jobs in this years scenario: *standard jobs*, *auctions*, *missions*, and *posted jobs*.

- **Standard Jobs**

Standard jobs consist of delivering items to a facility.

- **Auctions**

Both teams can bid on auctions. The amount they submit is what they will get for completing the job. The team with the lowest bid wins the opportunity to do it. However, if they are not able to complete it within the allotted time, the team gets fined.

- **Missions**

Missions are given to all teams, and have to be completed within a deadline to avoid a fine.

- **Posted Jobs**

Posted jobs are similar to standard jobs, only created by opposing teams. The delivered items are given to the posters of the job, while paying the reward.

4.2.2 Items

Items are what all jobs are about. All items have a specific volume, but can have a different price depending on where they are sold. Some items can be bought at shops and collected at resource nodes, while the remaining items have to be assembled in workshops. There is also a special type of items, called tools. To assemble certain items, specific tools are required. Each tool is associated with specific roles, entailing that each vehicle has a distinct set of tools to utilize.





Role	Speed	Load	Battery	Travel
 Drone	5	100	250	air
 Motorcycle	4	300	350	road
 Car	3	550	500	road
 Truck	2	3000	1000	road

Table 4.1: MAPC Roles

4.2.3 Roles

The scenario introduces four roles: *car*, *drone*, *motorcycle*, and *truck*. These roles control how much load an agent can carry, how fast it moves, its battery capacity, and which tools it can use. As a general rule, faster moving vehicles can carry less and have to recharge more often. For instance, motorcycles move faster than cars and trucks, but can carry less and has a lower battery capacity. Drones has the additional advantage of being able to fly, moving in straight lines (euclidean distance), while the other roles have to follow roads. An overview of the different roles' specifications can be found in Table 4.1.

4.2.4 Facilities

There are six different types of facilities: *charging station*, *dump*, *resource node*, *shop*, *storage*, and *workshop*. All their locations are always known, except for the *resources nodes*', which first have to be discovered. Each type of facility has a specific purpose to fulfill. A detailed description of these are given below, along with an outline in Table 4.2.







Facility	Purpose
 Charging station	Charging vehicles
 Dump	Dumping items
 Resource node	Gathering items
 Shop	Buying items and tools
 Storage	Storing items and delivering jobs
 Workshop	Assembling items

Table 4.2: MAPC facilities with their map icon and overall purpose.

- **Charging Station**

The vehicles have limited battery capacity, and to move around, charge is consumed. To recharge a vehicle, an agent has to move to a charging station, where its battery is filled at a given rate, depending on the charging

station. If an agent runs out of charge, it has to utilize its solar panels to recharge at a extremely low rate.

- **Dump**

Dumps allow agents to get rid of their items, removing them from the scenario indefinitely.

- **Resource Node**

Resource nodes can be utilized to gather items, where each resource node has one associated item available for gathering. An item is gathered after several attempts (gather actions), where having multiple agents doing it at once will speed up the process. To locate a resource node, an agent has to be within a given proximity.

- **Shop**

Shops have a subset of all the items and tools available for sale. Each shop has different prices and quantities of their associated products and a restock value, denoting how many steps are required before one of each product is restocked.

- **Storage**

Storage facilities can be used by agents to store their items and tools, allowing them to be retrieved at a later point. Each storage facility has a limited capacity, denoting the total volume of items and tools which can be contained at any time. Furthermore, storage facilities are also the delivery location for jobs, where the required items for completing a job is delivered.

- **Workshop**

Last but not least, the workshops, being the facilities where tools are utilized to assemble parts into even larger items. Multiple agents can be used to assemble an item, which is sometimes a necessity depending on the different tools required.

4.3 MASSim Environment

The MAPC provides the Multi-Agent System Simulation platform (MASSim), which is a predefined environment consisting of a set of applicable actions and a set of perceivable percepts. Actions and percepts are sent between agents and the environment using XML files. This is handled by the *Environment*

Interface Standard (EIS)², a Java-based interface standard for connecting agents to controllable entities in an environment. As a result, agents can easily be connected by associating each agent with an entity on the environment side. How this is done is illustrated in Figure 4.2. Agents are then able to send actions to the environment by utilizing their associated entity, and EIS will handle the conversion from Java objects into XML and into applicable actions on the environment.

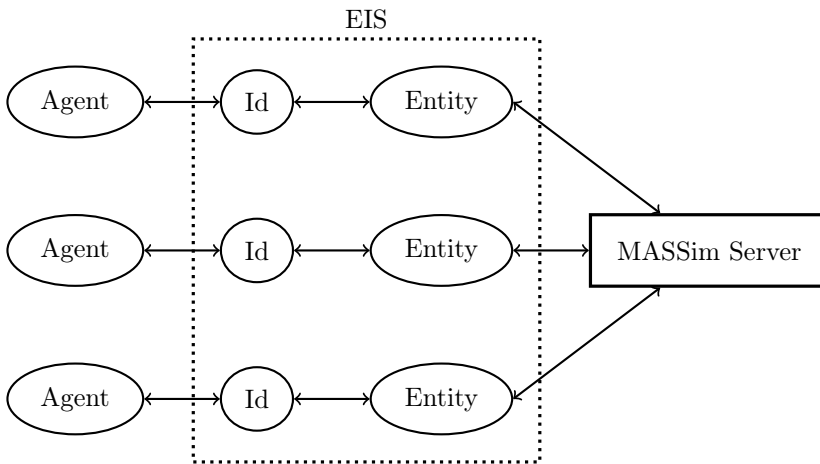


Figure 4.2: Overview of how each agent interacts with the MASSim environment running on a server using the Environment Interface Standard.

The MAPC can be executed on a single machine or multiple machines, since the MASSim platform runs as a server, either local or remote, receiving all its data through the EIS. Tests have been conducted locally, while the official contest will be held on a remote server. This allows all contestants to connect from their own computer.

4.3.1 Actions

The agents affect the environment by performing actions on it. They must do exactly one action in each step of the simulation, where the default action is `NOACTION`. Some actions take parameters to specify exactly what the agent wants to achieve, e.g. `goto(shop1)` or `buy(item1,6)`. All actions however, return error messages if the circumstances in which the actions are executed

²Detailed description and source code can be found at <https://github.com/eishub/eis>.

does not correspond with the environment's requirements. For instance, agents have to be in a shop to use the BUY action.

Furthermore, the simulation can be configured to nondeterministically replace an agent's action with the RANDOMFAIL action, making it possible for all actions to fail at any point. Doing so, makes the environment nondeterministic (stochastic), forcing the systems to be able to dynamically adjust to these errors. The following list explains some of the most important actions. Details about all the actions can be found in Appendix B.1, along with their failure codes.

- **GOTO**

Agents can use the GOTO action to move around the map. This action takes either a coordinate pair, latitude and longitude, specifying the location they want to move to, or the name of the facility they want to go to. This also means that the task of finding a path to the given location is abstracted away, allowing for a higher level of plans. Each GOTO action will consume 10 of the agent's charge.

- **BUY**

Agents also have a BUY action, which allows them to specify an item and an amount they want to purchase in a shop. For this action to succeed, the agent has to be located in the particular shop, and must have the given amount of the item available for sale, and the agent must have enough capacity to carry the items.

- **ASSEMBLE**

An agent can assemble items using the ASSEMBLE action, by specifying the item it wants to assemble. This action requires the agent to have all the required items in its inventory for the process to succeed. This means that the agent itself must carry the items, or that an assisting agent carries them. Other agents can assist an assembly by using the ASSIST_ASSEMBLE action, and specifying which agent to assist.

- **DELIVER_JOB**

To deliver a job, agents have to be at the correct location and use the DELIVER_JOB action. If the agent has any items to contribute towards the completion of the job, it will get a **successful_partial** message back, and if the job is completed a **successful** message, along with the reward for the job. If the agent has no items to contribute with, the environment will return a **useless** error message.

4.3.2 Percepts

Agents receive information about the environment through percepts. The MAS-Sim environment defines all the necessary percepts to allow agents to observe the state of the environment, as well as the states of the individual agents. As with actions, percepts are sent as XML messages from the environment to agents, which are all handled by the EIS. There are two methods for the agents to receive percepts: either by manually retrieving the newest percepts on demand, or by receiving each percept as a notification, handling them on the fly. The first method is preferred when the percepts are received at regular intervals. However, when percepts arrive at arbitrary intervals notifications are preferred.

The MASSim environment defines many different percepts, including the static information, dynamic information, agent-specific percepts and job percepts. The following list will give an explanation to each of these categories, while a complete overview of all percepts is provided in Appendix B.2.

- **Static Information**

The initial percepts include general information about the current simulation, such as how many steps it is going to be, which city it is taking place in, and which teams are competing. The initial percepts also include the *role* percepts, containing details for each of the roles, as well as the *item* precepts. The *item* percepts comprise all the items available in the simulation, including which items are needed for assembling them (if any). This information will never change throughout the simulation.

- **Dynamic Information**

At each step, an update of the dynamic information is sent to each agent. This first of all includes the *step* percept, informing the agents of the current step. Secondly, agents receive the current information about all the facilities, including the amount of items for sale in each shop, as well as their price and when the shop will be restocked.

- **Agent Percepts**

All agents will in each step get an update on how much charge and load they have together with their current location. If they are in a facility, the name of the facility will also be included. Furthermore, information regarding the agents' last action is perceived, containing the name of the action, the parameters given, and the result of the action, which can be useful for handling errors.

- **Job Percepts**

The last important percepts from the MASSim environment are the job percepts. For all types of jobs, start and end time, delivery location, reward and items required for completing the job is included. Auctions and missions include extra information about the fine for not completing the job. In addition, auctions also include the current lowest bid, when the auction is going to end, and the maximum reward. Lastly, posted jobs include the same information as standard jobs.

4.3.3 Properties

As mentioned in section 2.1, environments can model very different domains, being characterized by several different properties. In case of the MASSim environment, it is characterized by the following properties:

- **Multi-Agent**

The competition is based on having two or more teams compete against each other, where each team comprise multiple agents. As a result, the environment consists of both multiple cooperative agents (within a team) and multiple competitive agents (across different teams).

- **Fully Observable (with one exception)**

The locations of agents and facilities, except for resource nodes, are observable at any time. As a result, the environment is technically partially observable, but is for any other case fully observable.

- **Dynamic**

The agents that form the opposing teams cannot be controlled, and will continuously act upon the environment. As a result, the environment can change at any time, thus making it dynamic. Jobs are also a dynamic part of the environment, being posted at different points of the simulation.

- **Sequential**

Given that the simulation is based on a real life scenario, the environment is sequential. Each action executed on the environment is reflected in all future steps, hence each step depend on the previous one. For instance, if an agent at some point moves to a facility F , then the agent will be stationed at F in all subsequent steps, until it moves elsewhere.

- **Stochastic**

All actions have multiple outcomes due to `randomFail`, where the probability of successfully executing an action is $1 - p$ given that p is the

probability of a random failure. For instance, with $p = 0.15$ the probability of successfully executing an action is $1 - 0.15 = 0.85$, or 85%. As a result, the environment is stochastic, and agents will have to adapt to these random failures and reconsider their plans when necessary.

- **Discrete**

Even though the competition is based on a continuous environment, the simulation is modelled as a discrete problem, where there is a fixed set of actions, percepts, and states.

4.4 Problem Simplifications

To allow for faster development, some initial simplifications in terms of what the solution should be able to handle and how easy it is to solve a given job, have been considered. This is done by adjusting the server configuration, containing variables for determining the difficulty of jobs and much more. A list of all the relevant parameters is shown below.

Jobs

<code>productTypesMax:</code>	The maximum amount of different items a job can require.
<code>difficultyMax:</code>	The maximum difficulty of a job.
<code>timeMin</code>	The minimum amount of steps a job must be completed within.
<code>missionDifficultyMax:</code>	The maximum difficulty of a mission.

Shops

<code>minProd:</code>	The minimum number of different items a shop sells.
<code>amountMin:</code>	The minimum amount of an item available.
<code>restockMax:</code>	The maximum number of steps before one additional item of each type is added (up to the initial amount).

Items

<code>baseItemsMax:</code>	The maximum number of different base items.
<code>volMax:</code>	The maximum volume of an item.

<code>valueMax:</code>	The maximum price of an item.
<code>maxReq:</code>	The maximum number of required items for assembly.
<code>reqAmountMax:</code>	The maximum amount per required item for assembly.
<code>toolsMax:</code>	The maximum number of tools required for assembly.
<code>toolProbability:</code>	The probability that an item will require tools to assemble.

Adjusting these variables allows for vast simplifications, e.g. making items free without taking up space. You can even make jobs require a single item, and give the shops an infinite supply of that item. Doing so does however provide unrealistic results in terms of the competition, which is why adjustments have been kept to a minimum. As a result, the `toolProbability` was first of all set to zero, removing the possibility of items requiring tools to be assembled. Using tools is not the first concern, and neither is handling shopping errors.

Shopping errors arise due to many agents attempting to buy the same items simultaneously, or simply because there is a shortage of some specific items. This is especially the case in terms of resources, which are items that can be gathered from resource nodes. Resources are sold in limited quantities in the shops, given that they are available elsewhere. Gathering resources is not a priority either, hence the amount of items available in shops was increased to account for this limitation. This simplification was done only for the first solution, and was removed again when it was capable of handling the issue.

It is also important to note that this years competition features 28 agents in each team opposed to 16 last year. While the number of agents has almost doubled, a new configuration taking the increased number of agents into account has not been made available. As a result, it is apparent that the configurations are not up to scale with the scenario, and until they are, reasonable modifications will be made.

4.5 Choice of Solution

Given the increased number of agents, solving task efficiently relies on a high level of coordination in combination with problem decomposition. By dividing tasks into several subtasks, it is ensured that all agents are utilized, not wast-

ing resources by doing nothing. It can however be difficult to achieve a good distribution in terms of task delegation, which is why the Contract Net Protocol (CNP) (section 5.6) is introduced. The protocol is used for sharing tasks, allowing agents to bid using a predefined heuristic which takes the agent's load, speed, and distance to a given facility into account. As a result, the agent who can carry the most, is fastest, and is closest to the destination wins the responsibility of completing the given task, which is the strength of and why the CNP was chosen.

A natural way of dividing jobs into subtasks is by having each subtask consist of delivering one of the job's required items to the delivery location, solving partial jobs at the time. Another possibility however, is when a task requires the acquisition of items from several shops, each subtask comprise the acquisition of items from one distinct shop. These subtasks are however not as easily completed, since once the items have been retrieved, they have to either be given to the agent responsible for solving the task, or used by assisting the agent with its assembly, both which call for coordination.

Agent coordination is usually done using protocols, i.e. predefined procedures stating the rules and methods of a given transaction, but can also be achieved using CArtAgO's artifacts. Operations can be implemented to block an agent's execution, until some predefined rules are fulfilled or the state of the artifact is satisfiable. By doing so, CArtAgO can be very helpful in terms of coordinating the vast number of agents, while separating the coordination procedures from the agents.

Implementation

5.1 Development Process

The solutions to the MAPC have been developed and implemented using an agile approach, having several iterations with more features and more advanced agent logic. The first concern was to implement a solid and modular framework for handling server communication, thus perceiving the environment and performing actions on it. The second concern was to filter and organize the overwhelming amount of percepts received. The third, but at least as important concern, was to find a way to efficiently assign tasks to the different agents, hence implementing a Contract Net Protocol. Finally, different strategies on how to solve jobs most efficiently was revised, introducing problem decomposition and hierarchical planning.

5.2 Server Communication

As previously mentioned in section 4.3, actions are executed on the environment, and percepts are perceived using the *Environment Interface Standard*. To allow this behaviour, the EIArtifact maintains an instance of the **Environment**

Interface and provides operations for registering agents and executing actions, while simultaneously receiving all the percepts. Some of the percepts are exposed as observable properties, which agents that focus on the `EIArtifact` will be able to observe and trigger events when they are updated. This is especially important for the `step` percept, indicating when agents are allowed to execute a new action on the environment.

Most percepts however, are passed on to various information artifacts, i.e. data containers, where the relevant information in the percepts are extracted and stored. This is done using two internal operations, `perceiveInitial` and `perceiveUpdate`, where `perceiveInitial` is, as the name would suggest, only used initially for perceiving the static information such as which items or roles are defined. The `EIArtifact` furthermore listens to the `step` percept by attaching an `AgentListener` (an interface provided by `EIS`) to one of the entities. This listener reacts to the `simStart` and `step` percepts, ensuring the `perceiveInitial` and `perceiveUpdate` operation are executed when necessary, perceiving all the relevant information.

5.3 Percept Filtering and Organizing

To filter and organize all the different percepts perceived, several artifacts have been created, each being responsible for handling a share of the percepts. First of all, it is important to distinguish the static percepts from the dynamic ones, since all static information will be received in every step, regardless of the fact that it never changes. Secondly, only a fraction of the percepts are agent specific, meaning that most of them are perceived n times in a scenario with n agents. To avoid perceiving the same information several times, all the common percepts are collected in a set, removing duplicates.

This set of percepts is sent to different types of artifacts which filter them according to which percepts they are assigned to handle. Each percept is then parsed differently depending on the type, and the information is stored in the artifacts. To simplify this process, `MASSim`'s class definitions have been exploited, creating some custom extensions as needed. By doing so, the artifacts achieve a separation of concerns, while being able to define operations to retrieve and manipulate the information needed by the agents. Table 5.1 contains an overview of all the different artifacts, and which kinds of percepts they are responsible for perceiving.

Some percepts are however needed more frequently than others, i.e. used to decide which intentions to execute or which goals to achieve. These percepts

AgentArtifact	For dynamically perceiving agent specific percepts such as an agent's charge, load, etc.
DynamicInfoArtifact	For dynamically perceiving map info such as the current money, timestamp, etc.
FacilityArtifact	For dynamically perceiving all the different facilities such as shops, storages, etc. These percepts are dynamic due to the change in a shop's assortment or a storage's content.
ItemArtifact	For statically perceiving all items and tools at the start of a simulation.
JobArtifact	For dynamically perceiving new jobs, auctions, missions, etc.
StaticInfoArtifact	For statically perceiving map info such as the specifications of the different roles, the length of the simulation, etc.

Table 5.1: Overview of info artifacts.

usually include the agent specific ones, and are made as observable properties, giving the possibility of using the values in rules and as preconditions for selecting plans. To prevent agents from observing all the other agents' values, an AgentArtifact is created. Each agent then only focuses on their corresponding AgentArtifact.

As a result, for n agents, $n - 1$ additional artifacts are needed, but considering each agent has v values, their belief bases are only extended by v beliefs opposed to $n \cdot v$ beliefs, i.e. reducing the amount of beliefs by $(n - 1) \cdot v$. To illustrate, consider a scenario with 28 agents, where each agent has 12 agent-specific values, that is 204 fewer beliefs in each agent's belief base at the price of 27 additional AgentArtifacts, which at the same time helps separate the agents' knowledge.

5.4 Agent Logic

The general approach for implementing the agent logic, has been to utilize hierarchical planning. This entails the need to define the simplest of actions, and by creating compositions of these, build up more advanced high-level plans. The simplest plan the agents are able to do, is executing an action on the environment. This is mostly handled on the server side, having an internal action (the `jia.action` below) request the execution of a specific action on the EIS. It is

however important to send the actions in the correct step, which is ensured by suspending the intention after the request using `.wait`. This is implemented as follows:

```

+!doAction(Action) : .my_name(Me) <-
    jia.action(Me, Action); .wait({ +step(_) }).

```

Before implementing a strategy for earning money and solving jobs in the MAPC domain, the agents' need some general rules and plans to allow for problem decomposition, facilitating the use of hierarchical planning.

5.4.1 Rules

Rules are used throughout the implementation to abstract calculations away from plans and to retrieve important information. This includes finding an agent's speed or max load, which both can be extracted from the role definitions. Agents also have rules for testing whether they have sufficient charge to move, or if they can reach their destination without charging first. Given that a `goto` action consumes 10 points of charge, a `canMove` rule can be formalized from this requirement:

```

canMove :- charge(X) & X >= 10.

```

To test if an agent has enough charge to reach its destination, it can use the rule below. Using the length of the route along with its speed, the agent can calculate how many steps it will take to complete this route. By using the step estimate together with its charge and charge threshold, the agent can check whether its level of charge is sufficient. The threshold is included to ensure that the agent is able to reach a charging station afterwards.

```

enoughCharge :- routeLength(L) & enoughCharge(L).
enoughCharge(L) :- speed(S) & charge(C) &
    chargeThreshold(Threshold) & Steps = math.ceil(L / S)
    & Steps <= (C - Threshold) / 10.

```

Note that the rule has been split into two cases, allowing the calculation to be done for any route. All the defined rules can be found in Appendix C.2.

5.4.2 Plans

The agents have many different plans for solving goals in the environment. Some are more complex and require a lot of coordination, while others are relatively simple, but nonetheless essential for the agents. In this section, some of the most important plans are presented.

Getting to Facilities

The most important part of the scenario is having the ability of moving around the map and visit facilities. To do so, plans for getting to a facility have been implemented. These plans rely on the *inFacility(F)* belief predicate, describing which facility the agent is currently in. The idea behind the plans are simple, and the Jason implementation of it can be seen below. If the agent believes it is in the facility, the goal has been achieved. This is the base case for the first **getToFacility** plan. If the agent does not believe it is in the facility, it has three different ways of achieving its goal. As Jason always starts with the first plan from the top, it will start by testing whether the agent can actually move, using the **canMove** rule. If this is not the case, it will do a **recharge** action, charging up until it is able to move again, continuing with its intention.

```
+!getToFacility(F) : inFacility(F).
+!getToFacility(F) : not canMove

    <- !doAction(recharge); !getToFacility(F).
+!getToFacility(F) : not enoughCharge & not isChargingStation(F)
    <- !charge; !getToFacility(F).
+!getToFacility(F)
    <- !doAction(goto(F)); !getToFacility(F).
```

The next plan checks if the agent has enough charge to reach its destination, and if it is currently going towards a charging station. If none of these are the case, it should intend to charge first, before continuing to the facility. This will ensure that the agent does not run out of charge before reaching the facility. If none of the previous plans are valid, nothing is stopping the agent from going towards the facility, simply executing one **goto** action at the time using plan recursion.

Buying Items

Retrieving items is also an important part of the scenario. This is primarily done by buying items at shops. To do so, three *buyItems* plans have been defined, taking a list of items where each item is a literal with a name and an amount, i.e. **map(Name, Amount)**. The base case for this plan is when the list is empty, and there are no more items to be bought. The second plan simply removes the first element from the list, given that the amount to buy is zero, continuing

recursively with the rest of the list.

The last plan is the only plan that actually performs the `buy` action. Note that the agent is assumed to be in a shop when using these plans, and the `inShop` rule is used to retrieve the name of the shop. The agent uses this name to ask the `ItemArtifact` for the amount available of a given item in a given shop using the `getAvailableAmount` operation. By passing the desired amount to buy as well, the `AmountAvailable` variable is the minimum of the desired amount and the available amount. After the `buy` action has been executed, the plan proceeds by recursion on the rest of the list and on the same item with an updated amount. The ordering is to allow the shop to restock before trying to buy the remaining amount.

```
+!buyItems([]).
+!buyItems([map(Item,      0)|Items]) <- !buyItems(Items).
+!buyItems([map(Item, Amount)|Items]) : inShop(Shop) <-
    getAvailableAmount(Item, Amount, Shop, AmountAvailable);
    !doAction(buy(Item, AmountAvailable));
    !buyItems(Items);
    !buyItems([map(Item, Amount - AmountAvailable)]).
```

Retrieving Items

By dividing plans into as small and simple cases as possible, more complex plans can be developed using a hierarchical approach. For example, to retrieve items from any shop, the `buyItem` and `getToFacility` plans can be combined. As seen in the `retrieveItems` plan, retrieving items from a given shop is done by first getting to the shop, followed by buying the items.

```
+!retrieveItems(map(Shop, Items)) <-
    !getToFacility(Shop);
    !buyItems(Items).
```

Delivering Items

In a very similar way, the agents can use the `getToFacility` plan in combination with the `deliver_job` action to deliver items for jobs. This is achieved by the `deliverItems` plan, going to the facility, followed by executing the action.

```
+!deliverItems(TaskId, Facility) <-
    !getToFacility(Facility);
    !doAction(deliver_job(TaskId)).
```

These are some of the most essential plans, which all agents need in order to complete jobs. All agents also include different plans for how to complete other intentions, such as charging. Plans for actions requiring coordination is more

extensive, e.g. assembling items, and will be discussed in the later sections.

5.5 First Iteration

The first iteration of the multi-agent system focus on getting jobs divided into partial jobs, greatly simplifying some of the problems. This entails splitting jobs into subtasks, delivering a share of the items at the time. By doing so, the agents use quite a few steps to solve each job, but at the same time solving many in parallel. It is worth noting, that for this strategy to work, the problem was simplified by removing the tool requirement, avoiding the need of agent coordination.

Each partial job consist of acquiring the required items, assembling them if necessary, and delivering the items to a specific location. Such a plan can easily be expressed in Jason, as seen below, dividing the process of solving a partial job into three smaller and simpler goals.

```
+!solve_job(Id, Items, DeliveryLocation) <-
  !retrieve(Items);
  !assemble(Items);
  !deliver(Id, DeliveryLocation).
```

To decide which agent gets the responsibility of solving which job, a *Contract Net Protocol* has been implemented allowing agents to bid on who can solve each job most efficiently.

5.6 Contract Net Protocol

Contract Net Protocol (CNP) is a task-sharing protocol in multi-agent systems, where agents form the *contract net* and where each agent may take the role of a *contractor*. This protocol is used to select which agent is most suited to complete a task or subtask, by having all available agents bid on the announced task based on their current values (load, speed, charge). As a result, the agent bidding the lowest, i.e. can complete the task with the least cost, will win the responsibility of solving the task.

Having integrated the CArtAgO framework into the solution, the CNP has been developed using two types of artifacts: a `CNPArtifact` for receiving and main-

taining bids, and a `TaskArtifact` for creating and announcing `CNPArtifact`s. The `CNPArtifact`s are announced by defining an observable property containing the name of the announced task, an arbitrary amount of arguments and finally the unique id of the `CNPArtifact`. By using an observable property, the task is received by all agents. The implementation is shown in Listing 5.1, where the artifact is made, the id of the `CNPArtifact` is appended to the arguments, and the observable property is defined.

```
@OPERATION
private void announce(String property, Object... args)
{
    try
    {
        String cnpName = "CNPArtifact" + (++cnpId);

        ArtifactId id = makeArtifact(cnpName, "cnp.CNPArtifact",
            ArtifactConfig.DEFAULT_CONFIG);

        List<Object> properties = new LinkedList<Object>(Arrays.
            asList(args));

        properties.add(id);

        defineObsProperty(property, properties.toArray());
    }
    catch (Throwable e)
    {
        logger.log(Level.SEVERE, "Failure in announce: " + e.
            getMessage(), e);
    }
}
```

Listing 5.1: Method for creating and announcing `CNPArtifact`s

To exemplify the use of the `TaskArtifact`, assume that it has just received a job from the server requiring a set of `Items` to be delivered to a `Storage`. This could be done by calling `announce("job", Items, Storage)`, which will instantiate a new `CNPArtifact` and define the observable property with the contents. Agents focusing on the `TaskArtifact` will then have `job(Items, Storage, CNPId)` added to their belief base. This will trigger an event, in which they bid using the `CNPArtifact` associated with the given `CNPId`.

When a `CNPArtifact` is created, a timer is started by executing an internal operation, indicating the period of time the artifact accepts bids. After an agent has given a bid, the agent executes the artifact's `winner` operation, using the timer as a guard before evaluating to true or false indicating whether the agent has won. If two or more agents share the same lowest bid, the winner is whoever placed its bid first. From an agent's point of view, the bidding process

has the following sequencing:

```
...
bid(Bid)[artifact_id(CNPId)];
winner(Won)[artifact_id(CNPId)];

if (Won)
{
    ... // Complete the task
}.
```

Tasks often involve many subtasks, which is why bidding is only done on one subtask at the time, re-announcing the rest of the task afterwards. By doing so, the CNP greedily picks one agent for each subtask, where all subtask are to be coordinated accordingly.

5.7 Second Iteration

The second iteration of the MAS system focused on a new strategy, relying on much more coordination between the agents. The goal was to move away from solving partial jobs in parallel and instead define smaller tasks, like buying items or moving them from one location to another. The goal of this solution is to have agents do jobs more efficiently by introducing a higher level of coordination.

To do so, the solution defines a specific role for trucks. Each shop has a truck assigned to it at the beginning of the simulation, after which the trucks are tasked with buying the items required for each job. The other agents are then set to retrieve the items from the trucks instead of the shop, meaning that they no longer had to be concerned about buying items. The trucks that are not assigned to any shops, given that there are more trucks than shops in the simulation, are assigned to workshops. These workshop trucks will then focus on assembling items, again simplifying the tasks the other agents have to do.

The remaining tasks comprise the retrieving and delivering of items. Similarly to the first iteration, all subtasks are posted using the CNP, distributing them across the agents. Given the initial simplifications, the subtasks mostly involve giving and receiving items to and from the trucks. To coordinate this behavior, protocols have been established.

The transaction of items require the giver and receiver to do the **give** and **receive** actions in the same step respectively. By utilizing Jason's internal *.send* action, this can easily be achieved by sending messages back and forth.

A truck and another agent can easily communicate which step they want to exchange items. The details of these protocols are explained in subsection 5.9.1.

This iteration also introduced the *announcer* agent. The announcer does not represent any agent in the environment, but serves the purpose of defining subtasks and announcing them to the other agents. The *initializer* was also introduced, creating all the necessary artifacts in the beginning of the simulation. This agent is necessary due to the fact that artifacts can only be made by agents or artifacts themselves. The initializer removes itself once it has fulfilled its purpose.

5.8 Third Iteration

Using trucks as mediators for solving jobs in the second solution proved to be a huge bottleneck, which led to a different approach in the third iteration. Jobs are still being divided into subtasks, where the agents use the CNP to distribute their workload. However, this solution defines different subtasks, being an attempt at backwards reasoning. Backwards reasoning entail considering how to achieve the initial goal by possibly introducing new goals, repeating the process for each additional goal.

The initial goal is to deliver a set of assembled items to a location. To do so, the assembled items must be acquired, which is done by assembling a set of base items. Furthermore, the base items must also be retrieved, buying them from shops. This calls for two different subtasks, the first is similar to the first solution, dividing the acquisition of assembled items into partial jobs. The second however, divides the retrieval of items into one subtask for each shop to visit. These subtasks are defined as **assembleRequests** and **retrieveRequests** respectively.

The winner of an **assembleRequest** is responsible for assembling and delivering a subset of items. In order to do so, the required base items must be retrieved, creating a shopping list and announcing **retrieveRequests** for each shop. Agents that are assigned the responsibility of retrieving items from a shop, are considered as assistants, assisting with the assembly. To do so, the agents meet at a workshop, initiating the assemble protocol, explained in subsection 5.9.2. Once all items have been assembled, the assembler can deliver the items to complete its task.

5.9 Communication Protocols

To coordinate the agents in terms of assembling, giving and receiving items, the agents have several communication protocols at their disposal, allowing them to agree upon a step of execution.

5.9.1 Give/Receive Protocol

The give and receive protocols are between two agents, where one of them initiates the protocol (initiator), and the other one accepts it (acceptor). As a result, the acceptor has the final say in which step the transaction will take place.

The give protocol is initiated by telling the acceptor which items the initiator would like to give. The initiator then waits for the acceptor to respond with **readyToReceive** and in which step it is ready. This step is then the agreed upon transaction step, and is always a step in the future to account for already pending actions. Both agents wait for the transaction step before executing the give and receive actions.

The receive protocol is identical to the give protocol, only difference being the replacement of give with receive and vice versa.

5.9.2 Assemble Protocol

The assemble protocol is based on having one agent in charge of assembling an item, possibly being a part of a subtask. To assemble the item in the most efficient manner, the assembler requests help from other agents to retrieve the required items, assisting with the assembly afterwards. The assembler will therefore retrieve as many items as it can, and post the remaining items to the Contract Net Protocol. Every other agent then bids on these tasks, sending a message if they have been assigned to the task to tell the assembler they are assisting.

After the assembler and every assistant have retrieved the necessary items, they meet at a workshop, it sends a message to the assembler with the **assistantReady** literal and its name as soon as it arrives. When the assembler is at the store, and have received this message from every assistant, the actual assemble process

can begin. This is first done by coordinating which step they will begin. The assembler sends a step to every assistant, which is the step it will do the **assemble** action. Every assistant will do the **assist_assemble** action simultaneously.

An assistant will stop doing this action if it does not have any items left in its inventory, or if it receives an **assembleComplete** signal. This signal is sent by the assembler once it has all the necessary items to complete its task. Subsequently, all agents remove their beliefs associated with the assembly, thus concluding the protocol. The assembler proceeds by delivering the items to the required location, completing the given task.

5.10 Fourth Iteration

The fourth iteration was an attempt to deploy the same strategy as in the previous iteration, but with a much more dynamic approach. This was done by having each task divided into even more subtasks, and having the agents check their individual states before continuing towards their goals. As a result, the agents do not have to rely on every action succeeding, being more robust in terms of a competition setting. At the same time, this solution greatly emphasizes that agents should always be able to solve their tasks individually, not having to rely on others to achieve their goals.

This was a huge disadvantage with the previous solution, where agents sometimes relied on assistance from other agents to solve given subtasks, thus allowing for deadlocks in cases where all agents require help from one another. To prevent agents from taking jobs they cannot solve on their own, a load requirement is calculated for each job, only bidding on jobs for which they meet this requirement. Doing so has a downside however, since many of the jobs require heavy items only trucks have the capacity to carry. As a result, the agents take on fewer jobs, leaving the smaller vehicles idle.

The overall algorithmic functionality is explained by the following steps:

1. When a job is announced, the fastest agent who meets the load requirement gets the responsibility of solving it.
2. If the agent is unable to carry all the items at once, the task is announced further, in which free agents may take the responsibility of solving parts of it. If no other agents are free, the task is divided into several subtasks, completing one subtask at the time by delivering partial jobs.

3. To solve a subtask, a shopping list is created, telling which shops to visit and which items to buy in each shop. The shopping task is announced further as well, in which free agents may take the responsibility of retrieving the items from one of the shops. Similarly to the previous step, if no agents are available, the task is divided into several subtasks, retrieving the items from one shop at the time.
4. Depending on whether the agent has received assistance or not, the assembling procedure requires coordination. If the agent carries all the required base items by itself, it can proceed to a workshop and start assembling the items. If not, the agent initiates the assemble protocol, coordinating the assembly with its assistants.
5. Once all items of the given task or subtask has been assembled, the agent can proceed to the delivery location and deliver the items.

To ensure dynamic behavior, the agent attempts to reannounce the remaining workload after completing a subtask, unless there is only one subtask to go. To illustrate the difference between this solution and the previous one, consider how items are assembled:

```
+!assembleItems([]).
+!assembleItems([map(  _,      0) | Items]) <-
  !assembleItems(Items).
+!assembleItems([map(Item, Amount) | Items]) <-
  getRequiredItems(Item, ReqItems);
  !assembleItem(Item, ReqItems);
  !assembleItems([map(Item, Amount - 1) | Items]).

// Recursively assemble required items
+!assembleItem(  _,      []).
+!assembleItem(Item, ReqItems) <-
  !assembleItems(ReqItems);
  !doAction(assemble(Item)).
```

Listing 5.2: Third iteration's implementation of item assembly.

This implementation assumes that the items are successfully assembled in each step, simply assembling one quantity of an item at the time while calling itself recursively with a decremented amount. In scenarios where `randomFail` is a possibility, this solution will not be adequate, since the failures will cascade making subsequent assemblies fail and eventually making the agent unable to solve its task. The more dynamic approach on the other hand, has been developed to take random failures into account by consistently trying to achieve some postcondition before continuing.

```

// Postcondition: Items in inventory.
+!assembleItems(Items) : hasItems(Items).
+!assembleItems([map(Name, _) | Items]) :
    jia.getReqItems(Name, []) <-
    !assembleItems(Items).
+!assembleItems([map(Name, Amount) | Items]) :
    jia.getReqItems(Name, ReqItems) <-
    !assembleItem(map(Name, Amount), ReqItems);
    !assembleItems(Items).

// Postcondition: Item = map(Name, Amount) in inventory.
+!assembleItem(Item, _) : hasItems([Item]).
+!assembleItem(map(Name, Amount), ReqItems) : hasItems(ReqItems) <-
    !doAction(assemble(Name));
    !assembleItem(map(Name, Amount), ReqItems).
+!assembleItem(map(Name, Amount), ReqItems) <-
    !assembleItems(ReqItems);
    !doAction(assemble(Name));
    !assembleItem(map(Name, Amount), ReqItems).

```

Listing 5.3: Fourth iteration’s implementation of item assembly.

The `assembleItem` plan is used to assemble an amount of a specific item, being its postcondition to have these assembled items in the agent’s inventory. Assuming correct implementation and that the postcondition holds, the `assembleItem` plan can be used several times in the `assembleItems` plan to guarantee that after executing the higher-order plan, several items are in the agent’s inventory, being its postcondition. By making sure that all low-level plans holds some postcondition, compositions of these can be used to make complex high-order plans while still ensuring the correct and dynamic behavior a nondeterministic scenario requires.

5.11 Fifth Iteration

The fifth iteration is a simple attempt at utilizing tools, building on the previous dynamic version. This is achieved by having all agents acquire the tools they can use in the beginning of the simulation, as long their capacity holds. Vehicles such as drones can carry a limited number of tools, and to account for this limitation, the agents will prioritize the acquisition of role-specific tools, i.e. tools usable by only one type of vehicle.

Once all the tools have been acquired, the solution continues by solving jobs as mentioned in the previous section, with the addition of coordinating tools. Before initiating the assemble protocol, the agent checks which tools are required for the assembly, inspecting its own and its assistants’ inventories to see if any of

the tools are missing. The agent proceeds by announcing a `toolRequest` with the missing tools, allowing other agents with the appropriate tools to take part in the assembly.

5.12 Evaluating Jobs

So far, all solutions have focused on solving as many jobs as possible, with no regards of evaluating which jobs to solve. Instead, each agent has bid on all announced jobs while idle, making the job selection process essentially random. With several solutions running, the next step was to implement a heuristic function for choosing jobs, making agents capable of solving the best jobs available. At the same time, solving missions should be prioritized, as not doing so will result in a fine, along with the addition of handling auctions. Auctions have higher rewards than standard jobs, being significant in terms of earning potential.

Two different approaches for this functionality were considered: estimating the total earnings for each job, or estimating how many steps it would require to complete the job, and thereby give a reward per step value. The former is a simpler way to evaluate jobs, as it only requires looking at the cost of buying all the necessary items and subtracting this from the reward. Furthermore, jobs can be filtered by introducing a certain threshold, avoiding the announcement of jobs whose reward is insignificant.

The latter approach is more complex, as it requires estimating how many steps it takes to complete jobs. This estimate is based on how many items must be assembled, how many items must be bought and how many shops must be visited. Additionally, the distance between all the facilities must be taken into account, and whether or not the agent has to charge in between. While this is not necessarily easy to estimate, an amount of steps can be calculated assuming a single agent will solve the job on its own. A more accurate estimate would take the actual agents solving the job into account, but this cannot be known beforehand having a decentralized solution. However, since the system only needs to compare all jobs on the same terms, this simplification is valid.

To solve an auction job, the team first have to win the rights to do so. This is done by having an agent bid on the auction, using the `bid_for_job` action, specifying the id of the job and the bid. By utilizing the heuristics evaluating job rewards, a bid can be calculated. The agent will then decide whether or not to bid depending on the profit being above a certain threshold. Agents will not bid on every auction, as winning an auction without completing it will result in a fine. To prevent taking on more auctions than the system is capable of

handling, a soft limit is introduced, limiting the number of active auctions to five at any time. The limit is soft, since it does not prevent missions and well rewarding jobs from being started.

Results

6.1 Evaluation

The metrics for evaluating the solution have been based primarily on the amount of money acquired at the end of the simulation, along with the number of jobs the agents were able to complete. These results depend on the configuration used, which should also be taken into account during comparisons. When a simulation is over, the MAS will output a file with each team and their corresponding **score** and **ranking**. Along with the rankings, the solutions will output a log file with the amount of money the team had in each step, and how many jobs was completed during the simulation. This log file is the base for the evaluation of each solution in this chapter. Please note that the results shown only include a data point for every 50 step.

The goal is to earn as much money as possible, however as the rewards for jobs varies, the primary focus for the five iterations has primarily been on solving as many jobs as possible. Therefore, no function was used to specifically choose jobs. Later, focus was changed to optimize which jobs to solve, increasing the earning potential. As the jobs in this year's MAPC scenario is very different from last year's scenario, the solutions will not be compared with the results from last year.

Each iteration has been tested extensively through several simulations. The results displayed in the following sections is selected from the benchmarks, being regarded as the average output of a given solution. As the scenario introduces multiple random factors, there have been some slight variations in the results, causing deviations of at most a few thousands. However, the number of jobs completed has been very consistent.

6.2 Configuration

All simulations have been tested using the same configurations, unless otherwise specified. The configuration is taken from the MAPC as of June 1st 2017, with tools and random failures deactivated, as this was not a focus for the first solutions. This configuration had 28 agents on each team, four drones and eight of the other types, starting with a capital of 50000. The amount and location of each facility is randomly generated by the server, but by using the same random seed, the facilities in the simulations stays the same. The number of items generated given by the MAPC is based on last years version, which only had 16 agents on each team. It was therefore chosen to increase the items' availability, to account for the extra agents.

6.3 Solution 1

The results from the first iteration of the MAS is shown in Figure 6.1. This, along with all the data displayed in this chapter, can be found in Appendix A. The first solution was able to make profit of almost 40,000 during a simulation by completing 39 jobs, giving a good baseline for future iterations. However, there is some interesting information learned from this data. First of all, the agents actually lost money during the last 300 steps, possibly due to agents not being able to complete jobs in time. Furthermore, the agents do not consider when the simulation is going to end, and starts jobs regardless of being able to complete them in time. As a result, money is spent in vain on items that are never used.

Agents have to invest money in items to be able to complete jobs. As a consequence, the amount of money has to decrease before it can increase, which is especially apparent in the beginning of the simulation. Jobs vary in both difficulty and reward, where those requiring more items, hence also more steps, will result in a larger profit once completed. While the graph shows an im-

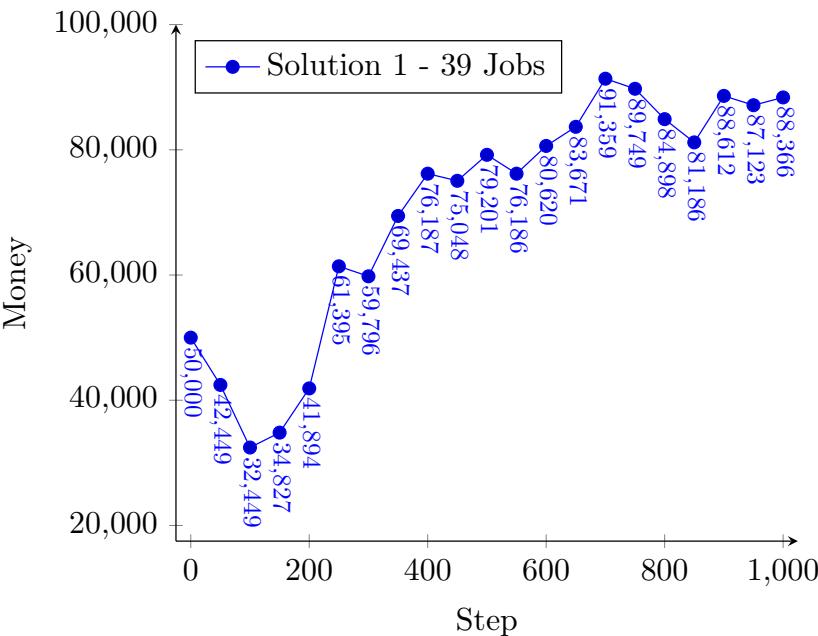


Figure 6.1: Money earned by the first solution of the MAS.

mense increase of nearly 20,000 from step 200 to step 250, this is most likely due to several jobs being completed at once, solving several tasks in parallel. This implementation does not consider which jobs to take, simply committing to completing the first ones that become available. Choosing the jobs that give the highest rewards could therefore help improve the system.

This solution solves several jobs at once, by splitting tasks into subtasks. However, if all subtasks are not completed, i.e. all items are not delivered, no profit is granted. As a result, if the last subtask of a job is not completed in time, all previous subtasks have been solved in vain. The same applies in terms of errors, where if one of the agents fail, all agents doing the same job practically fail, wasting time and money trying to complete an unsolvable job.

6.4 Solution 2

The second solution introduced bottlenecks at both the shops and the workshops. By having all items go through trucks, the trucks became overloaded

with work, only being able to give one type of item to a single agent at the time. The idea is that by having vehicles stationed at the shops, the system is able to buy items quicker than its opponents, thus ensuring the resources to solve a given task. While this limits the opposing teams, it also limits the system itself by having to pass on all the items bought. Similarly, by having trucks responsible for assembling, the other vehicles would not have to take part as assistants. As a result, the agents would only be tasked with simply carrying items between shops and workshops, and workshops and storage facilities.

However, the potential advantages does not outweigh the disadvantages, given that the solution had difficulties solving a single job. Because of this, further development was stopped and discarded for a better strategy, hence no results have been produced.

6.5 Solution 3

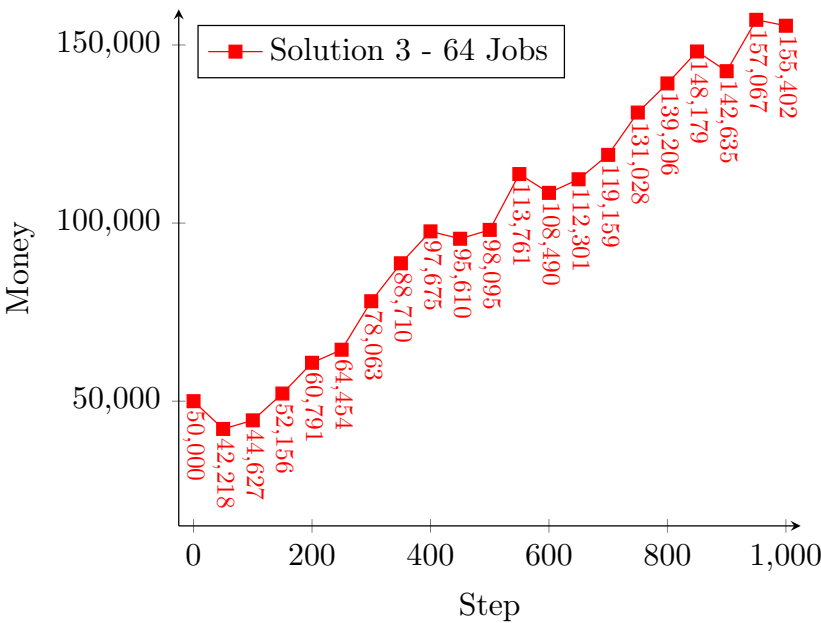


Figure 6.2: Money earned by the third solution of the MAS.

The results of the third iteration is shown in Figure 6.2. This time, the agents were able to triple the initial capital, ending on 155,000 by completing 64 jobs. Having a profit of 105,000, this solution earns more than twice as much compared

to the first solution. The results reflect the benefit of concentrating the resources towards fewer jobs at the time, avoiding situations where one subtask is solved too slowly for the system to earn any profit.

This version relies on agents being able to assist each other, having to coordinate their tasks. In previous solutions tasks have been independent of one another, allowing agents to focus on one subtask at a time, and continue with a new, unrelated task once it is complete. However, since agents have to wait for each other before assembling items, the solution has some limitations. The agents have to wait for all of them to arrive at the workshop before assembling, thus multiple agents may have to wait for the last. Instead, this can be optimized to allow agents to give their acquired items to other agents, relieving themselves of their duties. This assumes the other agents have the capacity to carry the given items.

6.6 Solution 4

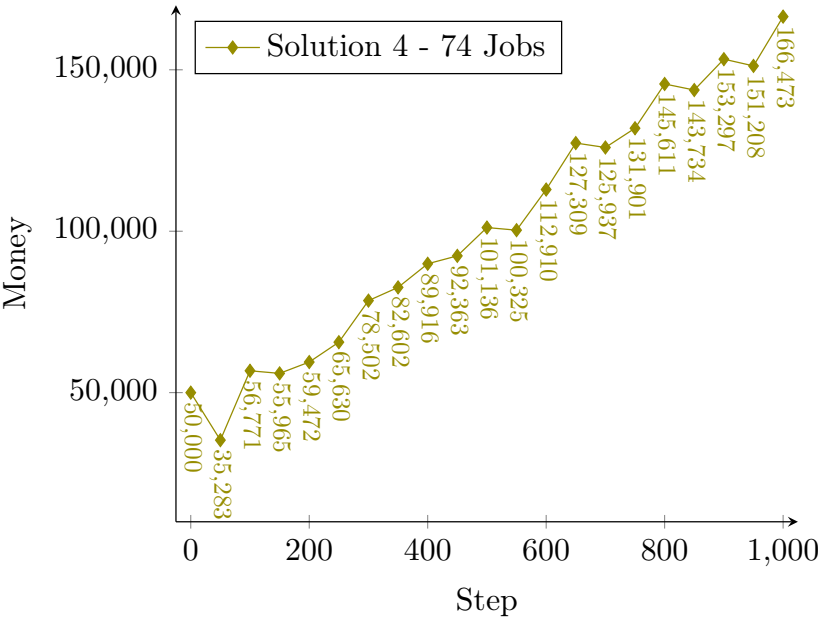


Figure 6.3: Money earned by the fourth iteration of the MAS.

The results of the fourth iteration is shown in Figure 6.3. As seen from the figure, the solution solves significantly more jobs, having an increase of about

15% from the previous solution. This is a result of dynamic behavior, being able to handle possible failures, while reusing leftover items. The solution does however not evaluate possible earnings from the various jobs, or take the agents' current items into account when choosing which jobs to solve, thus the results are very dependant on the solution coincidentally making the correct decisions.

The aim of this iteration is however not about solving an increased number of jobs, or earning more money per job, but about still earning money regardless of failures. To demonstrate its capabilities of failure handling, a comparison with various percentages of `randomFail` can be seen in Figure 6.4. With a `randomFail` of 1% and 28 agents on the team, a random failure will occur every 3.6 step on average for an arbitrary agent. Furthermore, with a `randomFail` of 10%, each agent will fail 100 actions on average during a 1000 step simulation, resulting in 2800 failed actions in total for 28 agents.

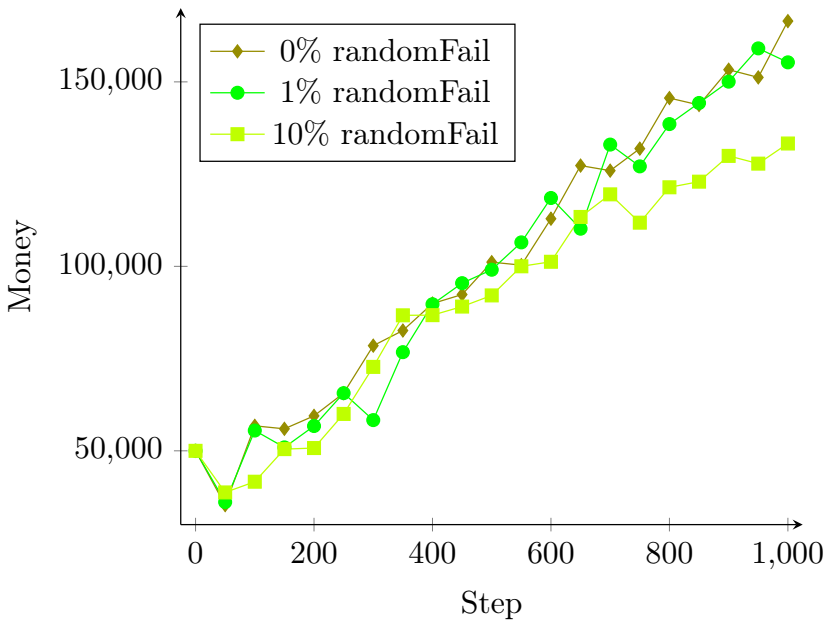


Figure 6.4: Comparison of solution 4 with various percentages of `randomFail`.

Despite this vast amount of failures, the error prone versions manage to keep up with the error free. Although the one with a 10% chance of failure falls off at the end, the solution with 1% chance of failure seems to be just as good as the one without any failures. The impact of the random failures will of course vary, for instance in a situation where five agents are assembling together, a random failure from one of the agents may result in all of the agents failing, failing not one, but five actions in practice. On the other hand, multiple failures e.g. going

to a workshop, will have no impact if the agent is to wait for another agent at the workshop anyways.

Given multiple reruns, ensuring that the results are consistent, it can be concluded that the system is capable of adjusting to random failures, facilitating the interaction with nondeterministic environments.

6.7 Comparison

Figure 6.5 includes a comparison between the solutions created so far. Here, the differences between the solutions are more apparent. One of the strengths of the first solutions is its ability to solve many jobs in parallel. As it will try to solve more jobs, it needs to invest in more items as well, which is why solution 1 uses more money then the other solutions at the beginning of the simulation. However, this also becomes the solution's weakness, as taking too many jobs at the same time prevents it from solving them all in time.

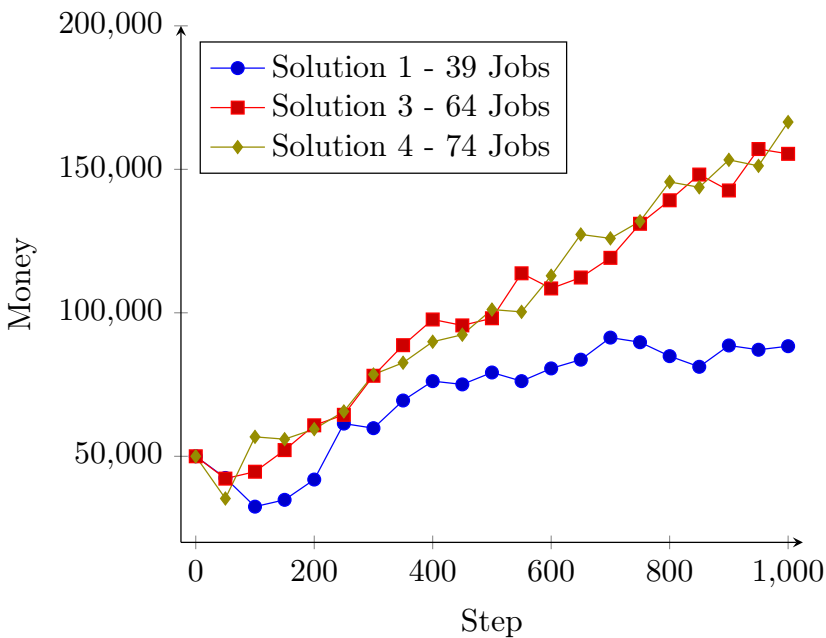


Figure 6.5: Comparison between the different solutions. All these simulations have been executed with the same configuration.

The third solution does not solve as many jobs at the same time, focusing on completing the jobs it has already started. This means it solves fewer jobs in parallel, but solves them faster. From this data, it seems to be a more efficient strategy, as solution 3 is solving 25 more jobs than solution 1. Solution 4 did best overall, solving a total of 74 jobs, beating solution 3 on the finish line. This is most likely due to solution 4 being more robust against errors, ensuring that failures are handled correctly.

6.8 Solution 5

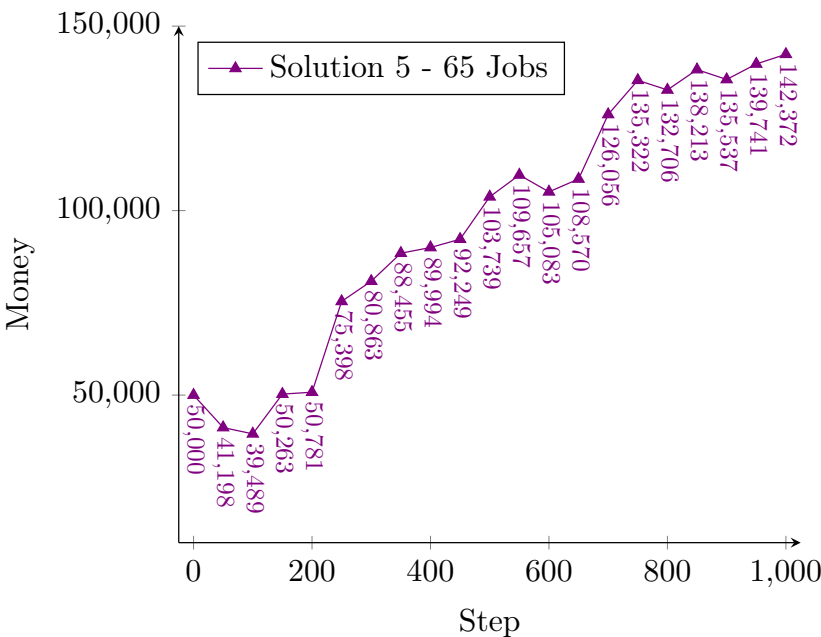


Figure 6.6: Money earned by the fifth iteration of the MAS.

The fifth iteration is the first attempt at utilizing tools, and the results of doing so are shown in Figure 6.6. This solution solves about 10 fewer jobs than the previous solution, earning approximately 25,000 less. This is a result of tools making jobs more difficult to solve, requiring a higher degree of agent coordination. Additionally, the agents have to spend quite a few steps in the beginning of the simulation to acquire tools, while solution 4’s agents starts solving their first job as soon as it becomes available. Acquiring tools does not only take time, but has a cost as well, not earning any significant profit prior to

step 250.

At the same time, additional resources in terms of agents are needed to solve jobs, where some required tools are missing. For instance, consider a group of agents consisting of trucks, cars and motorcycles, having retrieved all base items needed for a job. These base items are to be assembled, requiring a set of tools including a tool only usable by drones. To assemble the items, the agents depend on a drone being available, having to wait with the assembly if all drones are occupied solving other jobs.

While solution 4 emphasizes that agents should be able to solve the jobs they take individually, this is no longer the case having included tool requirements. Once again, this introduces the possibility of deadlocks, where several groups of agents await some resource to become available, while in fact, waiting for each other. To prevent such scenarios, instead of assigning one agent to a job at the time, groups of agents could be assigned, ensuring that this group is able to solve the job independently. By doing so, the tools could also be bought on demand while retrieving items, not having to waste additional money and capacity on excess tools.

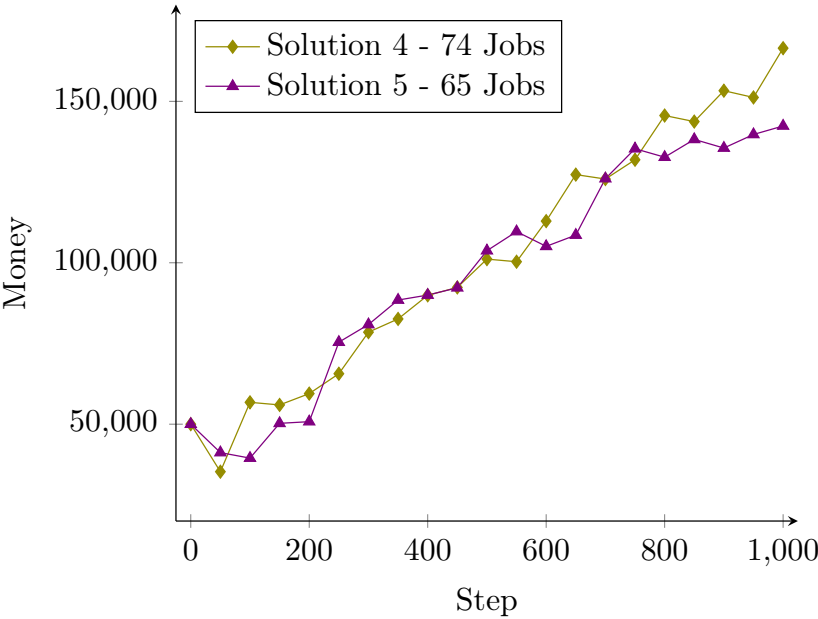


Figure 6.7: Comparison of solutions 4 and 5.

Regardless of these disadvantages, the results of this solution is not far from the previous, as can be seen in Figure 6.7. Similar to solution 4, the results are

dependent on which jobs are coincidentally chosen, avoiding situations where the agents have to wait on one another. Furthermore, which roles can use which tools will also have an impact on the results. If many of the tools are only usable by drones, the drones will not be able to carry all tools, due to the limited number of drones in the simulation compared to other agent types.

6.9 Job Evaluation

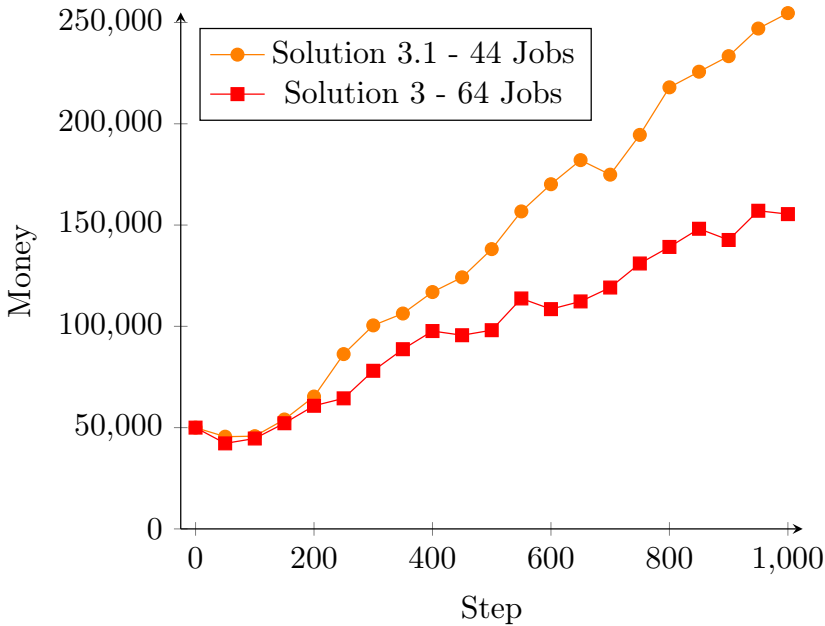


Figure 6.8: Results from solution 3.1, which uses a job heuristic to chose the best jobs to solve. This has resulted in doubling the money earned during a simulation compared to solution 3.

By implementing a more clever strategy to choose jobs, a massive increase in the amount of money the team earned was found, even though they were able to complete far fewer jobs. Figure 6.8 shows the results of including job evaluation to solution 3. This version uses the simpler heuristic function, only considering the reward compared to price. While the results of this inclusion is only shown for solution 3, the feature has been added to the newer solutions as well, but without being thoroughly tested and therefore not included. However, similar improvements can be expected, and have so far been confirmed by the

few simulations run.

As shown in the graph, this addition made the system able to actually double the profit, earning an additional 100,000. It should be noted, that as it is solving auction jobs, which no opposing team is bidding on, the rewards for these jobs are likely higher than they would be in a competitive match. While the team is earning much more, this is done by solving fewer jobs (44 now compared to 64 before), likely because the difficulty of the jobs increases with the reward.

6.10 Matches

The MAPC is designed to have two or more teams compete against each other in a match. To test the solutions in a real contest scenario, they have been pitched against each other in order to evaluate their performance. This is done using the same configuration as the previous simulations, only with multiple teams instead of one.

During a competitive match, the solutions generally perform worse than they do by themselves. This is likely due to the fact, that there are the same amount of items and jobs available to the teams, and each job can only be solved by one team. When multiple teams attempt to solve the same job, all teams except one will end up having wasted their time and resources solving the job. The amount of resources available remain the same, making it more difficult to get the required items to solve the jobs.

Solution 1 vs. Solution 3

The first match pitched the first and third solutions against each other. The results of this can be seen in Figure 6.9. It is clear that both solutions have more problems in a competitive setting. Solution 1 solves 20 jobs, nearly half compared to the single team scenario, while solution 3 only solves 39 jobs, 25 jobs less than before. As a result, solution 1 is not able to make a profit, while solution 3 barely reaches 100,000, earning 50% less than when it was on its own.

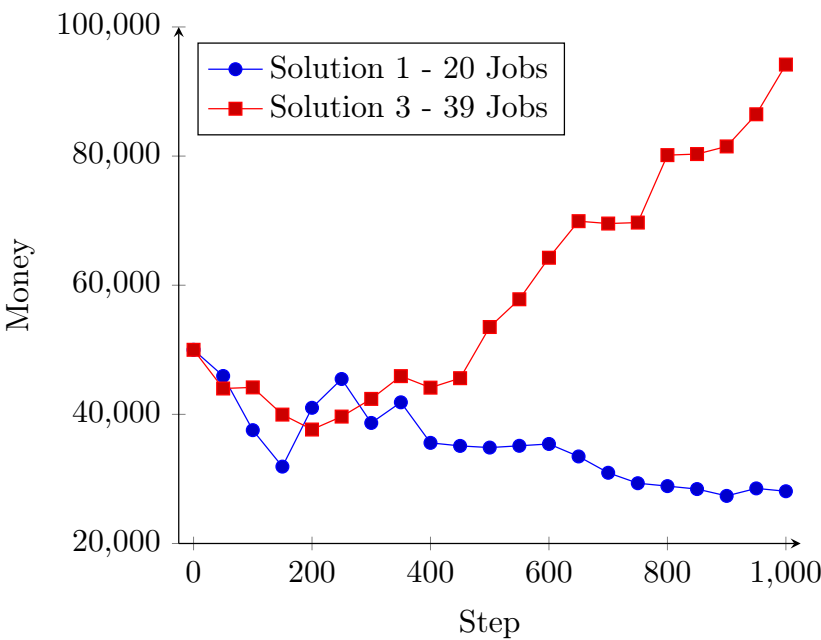


Figure 6.9: The first solution against the third solution in a match.

When agents fail at completing a job, they are not able to deliver their items at the storage facility. By not taking this into account, their inventories will eventually build, reducing their capacities as the simulation goes on. As a consequence, the agents will be less capable of completing jobs in the future. If the items were to be reused, it would not only preserve space, but allow agents to more easily solve subsequent jobs.

Solution 3 vs. Solution 3

The next match compared two instances of the third solution, testing them against each other. This turned out to be rather interesting, as one did much better than the other. It was expected that both would choose the same jobs to begin with, and try to solve them in a rather similar way. While they did choose the same jobs, different agents were chosen to solve them, resulting in one team earning much more money than the other. However, none of them earned nearly as much money as they did on their own.

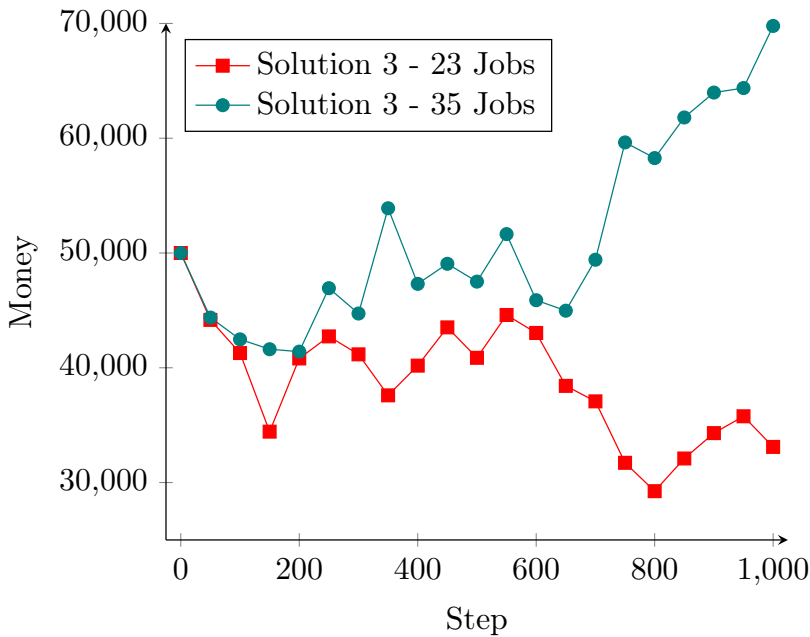


Figure 6.10: Match between two teams, both controlled by the third solution.

Solution 3 vs. Solution 4

All the matches simulated between the third and fourth solution have been very close. One of these matches can be seen in Figure 6.11. In all cases, solution 4 came out on top, but with a relatively small deficit. Solution 3 did manage to make a small profit in some simulation, but did poorly in most.

Solution 4 is developed with additional constraints, but is at the same time

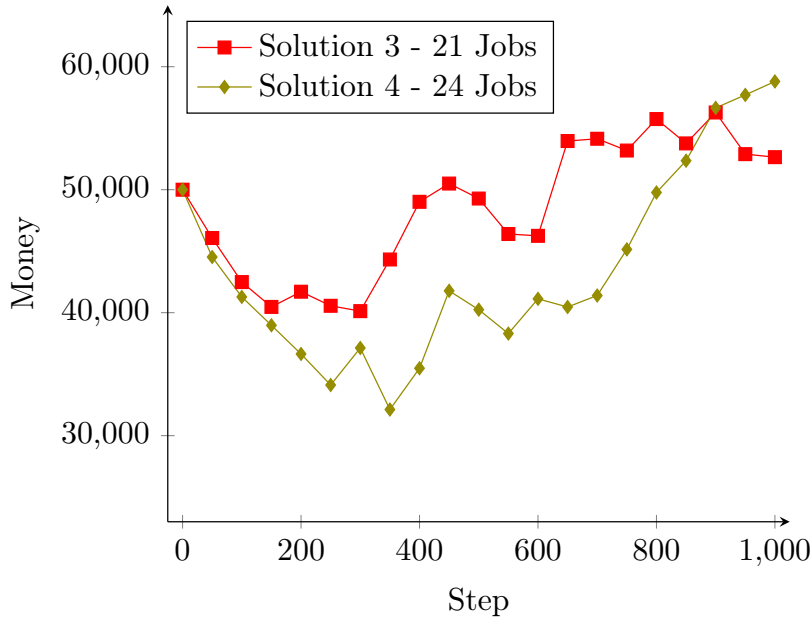


Figure 6.11: Match between solution 3 and 4.

more resistant to failures. When failing a job, it is capable of reusing the items already bought, allowing for faster completion of jobs in the future. The solution also ensures correct behavior by having agents base their execution of plans on beliefs. For instance, an agent will not continue to a given deliver location before it believes that it has all the required items.

Solution 1 vs. Solution 3 vs. Solution 4

The MAPC simulations allows for several teams to compete at the same time, hence solutions 1, 3, and 4 were tested against each other. The results of this simulation is shown in Figure 6.12. The outcome shows that the third and fourth solutions are vastly superior to the first solution, completing only 5 jobs during the entire simulation.

Solution 3 and 4 perform similar to their other matches, completing the exact same number of jobs as in Figure 6.11. Despite solving 21 jobs, solution 3 is unable to earn a profit, possibly due to the agents’ inventories filling up, only being able to complete the simplest of jobs.

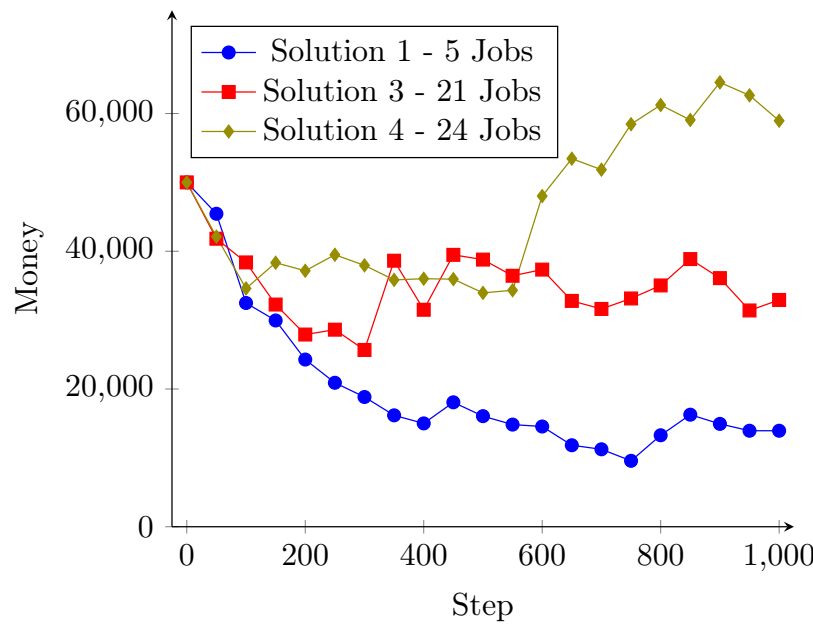


Figure 6.12: Solution 1, 3, and 4 against each other in one match.

The features added in solution 5 focus on a new configuration, where tools are required and random failures a possibility. As so, the other solutions are not capable of competing in the same environment, thus no matches have been simulated with this solution.

Discussion

7.1 Pitfalls of AgentSpeak

AgentSpeak is based on logic programming, allowing for elegant but expressive notation. Given that the language builds on a Belief-Desire-Intention architecture, it is intuitive for agent reasoning. This allows for defining means of how to achieve ones desires, in terms of the beliefs about the environment. However, implementing good reasoning is not always as intuitive, and there some pitfalls to the agent-oriented programming language.

7.1.1 Performance Issues

Having experienced several situations where the Java Virtual Machine (JVM) starts tearing at the CPU, while steadily allocating and consuming more RAM. While the cause is obvious, being an infinite loop, or more likely an infinite recursion, the root of the problem is not obvious or easily detectable. Considering a pure Java application, this would either result in a `OutOfMemoryError` or `StackOverflowError`, with a stack trace leading directly to its origin. In AgentSpeak however, the excessive use of computer resources continues until

the program is terminated, having limited debugging options available. To exemplify, consider the following minimalistic excerpt:

```
+start <- -+start.
```

In this example, when the **start** belief is first added, it will trigger an event removing and adding the belief once more, which will trigger the same event, resulting in an infinite sequence. While the pitfall is obvious, it can be hard to detect when the removing and adding of beliefs are in different plans or wrapped within a larger and more complex piece of code. For instance, by including a **free** belief to denote when an agent is free in terms of taking on new tasks. When solving a task, the **free** belief should be removed, and after completing the task, it should be added once again. Furthermore, once an agent is free, it should try to solve already existing tasks before taking on new ones. If unaware of the possible pitfall, more advance code can easily be boiled down to the previous excerpt. To illustrate:

```
+free : task(Some, Terms) <-
  -free;
  !solveTask(Some, Terms);
  +free.
```

If the **task(Some, Terms)** belief is never removed, the AgentSpeak reasoning will continue solving the task indefinitely, regardless of the task being solved or not. Event triggers are however not the only pitfalls to be aware of, considering simple plan recursion as well. The first plans for buying items were previously defined as follows:

```
+!buyItems([]).
+!buyItems([map(Item, 0)|Items]) <- !buyItems(Items).
+!buyItems([map(Item, Amount)|Items]) : inShop(Shop) <-
  getAvailableAmount(Item, Amount, Shop, AmountAvailable);
  if (AmountAvailable > 0) {
    !doAction(buy(Item, AmountAvailable));
  }
  !buyItems(Items);
  !buyItems([map(Item, Amount - AmountAvailable)]).
```

The only difference is the additional if-statement encapsulating the **doAction** to prevent the agents from attempting to buy 0, or none, of an item. Doing so would in fact fail on the server, responding with a failure message. By including the if-statement, the agent will instead of failing, attempt to buy one of the other items first and retry to buy the item at last, assuming that the shop has been restocked. The results of this inclusion is however undesirable, considering a scenario where there are no other items, or the other items cannot be bought as well. In this case, the plans will enter an infinite recursion, once again,

consuming all computational resources available.

7.1.2 Delayed Percepts

Delayed percepts have also been a difficulty, where agents would use an additional **goto** action to get to a facility they are already in, or **charge** one additional time when they are already fully charged. Having traced the percepts all the way from the server, through the environment interface, entity, artifact, observable property, and finally to the agent, it seems that the observable property would nondeterministically not be received by the agent. This made its beliefs about the environment not synchronized with the actual environment. This asynchronous behavior also occurred at different times for the individual agents, often in combination with solving tasks. Examine the following two very similar methods of executing actions on the environment:

```
+!doAction(Action) : .my_name(Me) <-
    jia.action(Me, Action); .wait(step(_)).
```

```
+!doAction(Action) : .my_name(Me) <-
    jia.action(Me, Action); .wait({+step(_)}).
```

In the first example, after an action is executed, an arbitrary step is waited for, e.g. `step(10)`. In the second example however, an explicitly new step is waited for. The examples are nearly identical, so how big an impact can it possibly have? The first solution waits for an arbitrary step belief, hence it does not wait at all, given that the agents always believes that the step is something, whether it is 0 or 567. What it does on the other hand, is attempting to execute several actions on the environment at once. The environment is set to block the execution of additional actions until an action id becomes available, that is, in the next step when agents are allowed to execute another action.

As a result, the environment blocks the agents, making them unable to receive updates of the observable properties, and the actions they execute in step n are based on the percepts from step $n - 1$, hence using an additional action when going to a facility or charging. For instance, when an agent has just charged up to full battery capacity, its believes about its charge, which is from the previous step, indicates that one more **charge** action is required. By explicitly waiting for a new step, the agent perceives the changes to the environment before continuing its reasoning, and thereby basing the execution of the next action on beliefs that are up to date.

7.2 Side-Effects of CArtAgO

While CArtAgO provides some great functionality for coordinating agents and creating vast distributed networks, it also introduces some unwanted side-effects which are not to be ignored. First of all, CArtAgO provides their own implementation of the environment and the agent architecture, extending the Jason implementation. This is not necessarily a bad thing, but while Jason strongly encourages overriding and customizing their classes, CArtAgO does not. For instance, the `perceive` method in the agent architecture, which is usually overridden to account for the domain-specific percepts, is replaced by an almost 300 lines long method, making it difficult to change its behavior without causing errors. As a result, the developer is forced to work with what CArtAgO provides, instead of using it as an optional extension to the sometimes more intuitive Jason approach.

7.2.1 Percepts and Perceiving

Observable properties would be one of the greatest additions CArtAgO provides, had it not removed the possibility of adding percepts to agents in the environment in other ways. Unless the intention is to create one artifact for each agent, all agent specific percepts have to be defined as observable properties with the name of the associated agent as an argument to distinguish the percepts from one another. Furthermore, the agent specific percepts will be observable by all agents in the environment, which is not very intuitive. Using only Jason, this could easily be achieved by simply adding the percepts to the respective agents in the environment, but since CArtAgO implements their own reasoning cycle, this has not been made possible.

Recall how this project's solutions handle percepts using different (information) artifacts and storing the information in objects (section 5.3). While this approach is more object-oriented than agent-oriented, the alternative is to define every percept as an observable property, given that CArtAgO removes the possibility of adding percepts to agents in the environment. Doing so would not only be less efficient, but less modular and harder to work with. Instead, artifacts in the solutions have been provided with an extensive usage interface, e.g. returning all base items required for an item, or the name of the closest facility of a specific type.

The information artifacts are implemented as static data containers, with operations to retrieve the data and static perceive methods to update them. Instead of relying on static classes, the same could be achieved by using CArtAgO's

artifact approach. This is done by defining the static methods as link operations instead, being able to execute the operations from other artifacts, e.g. the EIArtifact which passes on the percepts. However, linked operations are executed asynchronously, and does therefore not guarantee that the percepts have been perceived once the agents continue their reasoning. This issue arose in an attempt to do so, where none of the information was up to date any longer.

7.2.2 Operation Semantics

CArtAgO operations are very useful for retrieving information from the artifacts. However, they are not very useful for selecting plans given that they do not rely on unification to provide feedback. As a result, operations are not usable in rules or in preconditions for plans. For example, a useful rule could be one which evaluates to true if and only if an agent has all items in a given list. This should easily be achieved by using an operation to retrieve the list of items in the agent's inventory, followed by unifying it with the given list. However, AgentSpeak does not allow for compositions of statements in rules and preconditions, and must therefore be implemented as an internal action instead. The illustration below attempts to clarify the dilemma:

```
// As two separate statements. Valid syntax, but the operation does
// not evaluate to true or false, and is parsed as a rule/belief.
hasItems(Agent, Items) :- getInventory(Agent, Inventory) & Items =
    Inventory.
// As a composition of two statements. The composition evaluates to
// true or false, but this is not valid AgentSpeak syntax.
hasItems(Agent, Items) :- (getInventory(Agent, Inventory); Items =
    Inventory).
// Implementing an internal action which does exactly the same.
// Only valid solution.
hasItems(Agent, Items) :- jia.hasItems(Agent, Items).
```

While this is not entirely on CArtAgO, they could for instance have successfully executed operations evaluate to true, and otherwise to false. More importantly, a prefix should be added to distinguish the operations from rules and beliefs, which is what AgentSpeak recognizes the operation in the first example above as.

7.2.3 Operation Invocation

Aside from not being usable in rules and preconditions, some operations halted the execution of the agent's reasoning cycle. In particular, the operation per-

forming a given action on the server environment. Since the environment changes in every step, the agents should continue their reasoning once they have perceived a new step, which is after all other percepts have been perceived. The appropriate way to do so, would be to wait for the new step to be perceived, once the agent has sent its desired action for the current step to the server. However, the EIArtifact would nondeterministically prevent agents from continuing their reasoning, leading to several missed steps. While the cause seems to be related to synchronous behavior in the CArtAgO kernel, the solution was simply to replace the operation with an internal action.

There has also been some issues disposing an artifact from another artifact. Given prior experiences with the CArtAgO implementation, their source code was inspected and the cause found. Calling the artifact's `dispose(ArtifactId aid)` method with `aid` being the id of the artifact to dispose, subsequently calls the CArtAgO environment's `disposeArtifact` method, but with the wrong id. Instead of passing on the provided `aid`, the method passes on its own field `id`, resulting in the disposal of itself. The bug has been reported and fixed by the CArtAgO development team.

7.3 Future Work

7.3.1 Freeing Resources

There are several situations where the system uses more resources than required, especially considering the number of agents dispatched. This is due to a lack of replanning, for instance when a new job requires the acquisition of items from shops where agents are currently located or on their way to. Assuming they have room for the items and the destination is the same, it would make perfect sense to have the agents shop for multiple jobs simultaneously. By doing so, it would save several agents the extra trip, being able to focus the resources elsewhere.

This is however not as straight forward as it sounds, because some agents, the shoppers, suddenly have the responsibility of solving multiple jobs at once. As a result, the shopping agents have to pass on their additional items, which requires just as many steps as buying the items separately, and will delay the agents' current jobs as well. Alternatively, the shopping agents can simply solve the tasks one at the time, becoming huge bottlenecks and unable to guarantee that the jobs are completed in time.

Replanning can free up a lot of unnecessarily spent resources, but is a complex task when it requires agent coordination. Given the solution of the third iteration (section 5.8), there is an easier way to increase performance. The solution relies on the assembly protocol (subsection 5.9.2) to coordinate the assembling of items. When this protocol involves three or more agents, two or more agents will always be waiting for the last ones. This could however be avoided, if some of the agents were to give all their items to the other agents, hence relieving themselves of the task and freeing the previously unavailable resources. The number of agents that can be freed depends on the volume of the items and the capacity of the remaining agents, and with the introduction of tools, it also depends on which roles are required in terms of role-specific tools.

7.3.2 Competition Application

Having focused on agent coordination rather than competition application, handling `simEnd` percepts have not been a priority, denoting the end of a simulation. After receiving such a percept, all processes should be halted, followed by perceiving initial percepts for the new simulation if and when a `simStart` is received. This has to be done, as the new simulation could take place in a new city with new facilities at new locations. Furthermore, the agents' beliefs about the previous environment should be discarded, allowing for a fresh start in the next round.

While the initialization should happen correctly in the beginning of each continuous simulation, it has not been tested with a complete competition setup. To do so, the simulation should run on a remote server, allowing the system to connect over internet. By using correct configurations, this process should be more or less automatic, but will be tested thoroughly before the actual competition.

Conclusion

This project has shown several successful implementations of a multi-agent system, capable of interacting and competing in the nondeterministic environment from this year's Multi-Agent Programming Competition. The first few solutions have focused on solving as many jobs as possible, while later solutions expand previous ones with additional features. These features include selecting which jobs to complete, handling random failures, and utilizing tools. The best results have shown profits up to 200,000 during a simulation. However, there are still some optimizations to be done regarding choosing jobs, which will likely increase earning potential. This has been done using problem decomposition and hierarchical planning, making the agents are capable of solving goals individually, but also coordinating their efforts to solve more complex problems.

The simulation environment is about earning as much money as possible, which is done by solving jobs. To do so, the solutions utilize many different techniques and technologies, making agents able to complete them efficiently. By using Jason and exploiting its Belief-Desire-Intention agent architecture, the agents are able to maintain beliefs about the environment and themselves, desire goals, and achieve these desires by fulfilling intentions. These concepts have proved to be great tools for developing highly sophisticated agent reasoning, making the agents capable of autonomously deliberate how to most efficiently solve their tasks.

Another important tool is the CArtaGO framework, being used to facilitate agent coordination, to store general information shared by the system, and to implement a Contract Net Protocol. The latter has proven very efficient in terms of task delegation, allowing agents to easily share tasks between each other, and select the agent best suited for the various jobs. In a competition scenario, how well the teams performs boils down to which team is superior in terms of agent coordination.

At this point, the multi-agent system still has room for improvement, especially in terms of using agents more efficiently and possibly how the jobs are selected. However, the focus for this project has been to implement efficient agent coordination, being one the most challenging tasks in terms of multi-agent systems. Having successfully done so, optimizing parts of the system will pose a smaller challenge for future iterations.

APPENDIX A

Benchmark Data

The data from the simulations conducted in this project are included in the following tables. Please note that no data from the second solution has been included, as it was not able to complete an entire simulation without halting.

Step	Sol. 1	Sol. 3	Sol. 3.1	Sol. 4	Sol. 5
0	50000	50000	50000	50000	50000
50	42449	42218	45527	35283	41198
100	32449	44627	45778	56771	39489
150	34827	52156	54018	55965	50263
200	41894	60791	65335	59472	50781
250	61395	64454	86297	65630	75398
300	59796	78063	100469	78502	80863
350	69437	88710	106328	82602	88455
400	76187	97675	116925	89916	89994
450	75048	95610	124192	92363	92249
500	79201	98095	138123	101136	103739
550	76186	113761	156734	100325	109657
600	80620	108490	170158	112910	105083
650	83671	112301	182057	127309	108570
700	91359	119159	174883	125937	126056
750	89749	131028	194515	131901	135322
800	84898	139206	217991	145611	132706
850	81186	148179	225709	143734	138213
900	88612	142635	233380	153297	135537
950	87123	157067	247022	151208	139741
1000	88366	155402	254644	166473	142372

Table A.1: The results from the all the solutions of the multi-agent system.
Solution 3.1 is solution 3 with job evaluation.

Step	0% Failures	1% Failures	10% Failures
0	50000	50000	50000
50	35283	36107	38726
100	56771	55498	41627
150	55965	50931	50464
200	59472	56720	50751
250	65630	65632	60000
300	78502	58325	72751
350	82602	76742	86727
400	89916	89731	86770
450	92363	95444	89098
500	101136	99083	92120
550	100325	106496	100013
600	112910	118526	101241
650	127309	110203	113390
700	125937	133001	119514
750	131901	127097	111838
800	145611	138560	121425
850	143734	144283	122952
900	153297	150061	129939
950	151208	159076	127839
1000	166473	155292	133295

Table A.2: The result data from the three different simulations with solution 4, where random failures are active.

	Match 1		Match 2		Match 3	
Step	Sol. 1	Sol. 3	Sol. 3	Sol. 3	Sol. 3	Sol. 3
0	50000	50000	50000	50000	50000	50000
50	45943	44005	44174	44376	46072	44526
100	37552	44176	41283	42477	42489	41280
150	31906	39963	34437	41616	40469	38974
200	41016	37650	40813	41407	41706	36643
250	45476	39655	42734	46935	40561	34119
300	38671	42395	41171	44727	40126	37131
350	41865	45919	37598	53897	44326	32124
400	35585	44137	40187	47312	49016	35481
450	35114	45601	43520	49060	50500	41778
500	34856	53528	40871	47506	49281	40244
550	35130	57823	44600	51645	46398	38302
600	35415	64260	43034	45887	46249	41125
650	33479	69918	38425	44974	53957	40462
700	30953	69553	37074	49414	54143	41396
750	29334	69703	31721	59634	53184	45147
800	28886	80149	29248	58269	55745	49772
850	28430	80313	32098	61807	53765	52356
900	27371	81495	34307	63976	56277	56652
950	28533	86474	35774	64371	52891	57706
1000	28090	94176	33106	69783	52646	58801

Table A.3: Data from the three matches between two teams.

Step	Solution 1	Solution 2	Solution 3
0	50000	50000	50000
50	45444	41821	42124
100	32487	38381	34598
150	29949	32262	38337
200	24276	27907	37167
250	20905	28614	39491
300	18839	25670	37952
350	16176	38630	35844
400	15003	31507	36011
450	18066	39480	35954
500	16057	38790	33954
550	14833	36443	34335
600	14553	37337	48018
650	11833	32793	53447
700	11245	31625	51847
750	9571	33147	58457
800	13282	35037	61237
850	16269	38865	59065
900	14935	36097	64521
950	13939	31405	62640
1000	13939	32929	58951

Table A.4: Results from match between every solution.

APPENDIX B

Multi-Agent Environment Details

Below we have listed some of the official details of the Multi-agent Competition platform. These have been defined by the creators of the competition, and can be found at <https://github.com/agentcontest/massim/blob/master/docs/scenario.md>.

B.1 Environment Actions

In each step, an agent may execute *exactly one* action. The actions are gathered and executed in random order.

All actions have the same probability to just fail randomly.

Some actions may lead to **conflicts**. For example, two agents might want to buy the same item from a shop (and only one of these items is left). In that case, the agent whose action is executed first gets the item, while the action of the other agent is treated as though no item is available (which is actually true).

Each action has a number of **parameters**. The exact number depends on the

type of action. Also, the position of each parameter determines its meaning. Parameters are always string values.

goto

Moves the agent towards a destination. Consumes **10** charge units if successful. Can be used with 0, 1 or 2 parameters. If 0 parameters are used, the agent needs to have an existing route which can be followed.

No	Parameter	Meaning
0	Facility	The name of a facility the agent wants to move to.

Note: Names of *ResourceNodes* are not allowed, as they are not common knowledge.

No	Parameter	Meaning
0	latitude	The latitude of the agent's desired destination.
1	longitude	The longitude of the agent's desired destination.

Failure Code	Reason
failed_wrong_param	The agent has no route to follow (0 parameters), more than 2 parameters were given or the given coordinates were not valid double values (2 parameters).
failed_unknown_facility	No facility by the given name exists (1 parameter).
failed_no_route	No route to the destination exists or the charge is insufficient to reach the next way-point.

give

Gives a number of items to another agent in the same location.

No	Parameter	Meaning
0	Agent	Name of the agent to receive the items.
1	Item	Name of the item to give.
2	Amount	How many items to give.

Failure Code	Reason
failed_wrong_param	More or less than 3 parameters have been given, no agent by the name is known or an amount < 0 was specified.
failed_unknown_item	No item by the given name is known.
failed_counterpart	The receiving agent did not use the receive action.
failed_location	The agents are not in the same location.
failed_item_amount	The giving agent does not carry enough items to give.
failed_capacity	The receiving agent could not carry all given items.

receive

Receives items from other agents. Can receive items from multiple agents in the same step.

No parameters.

Failure Code	Reason
failed_counterpart	No agent gave items to this agent.

store

Stores a number of items in a storage facility.

No	Parameter	Meaning
0	Item	Name of the item to store.
1	Amount	How many items to store.

Failure Code	Reason
failed_wrong_param	More or less than 2 parameters were given.
failed_location	The agent is not located in a facility.
failed_wrong_facility	The agent is not in a storage facility.
failed_unknown_item	No item by the given name is known.
failed_item_amount	The given amount is not an integer, less than 1 or greater than what the agent is carrying.
failed_capacity	The storage does not have enough free space.
failed	An unforeseen error has occurred.

retrieve and retrieve_delivered

Retrieves a number of items from a storage. The first can be used to retrieve items that have been stored before, while the second is used to retrieve items from the team's 'special' compartment (see Storage).

No	Parameter	Meaning
0	Item	Name of the item to retrieve.
1	Amount	How many items to retrieve.

Failure Code	Reason
failed_wrong_param	More or less than 2 parameters have been given.
failed_location	The agent is not located in a facility.
failed_wrong_facility	The agent is not in a storage facility.
failed_unknown_item	No item by the given name is known.
failed_item_amount	The given amount is not an integer, less than 1 or more than available.
failed_capacity	The agent has not enough free space to carry the items.

assemble

Assembles an item.

No	Parameter	Meaning
0	Item	Name of the item to assemble.

Failure Code	Reason
failed_wrong_param	Not exactly 1 parameter has been given.
failed_location	The agent is not in a facility.
failed_wrong_facility	The agent is not in a workshop.
failed_unknown_item	No item by the given name is known.
failed_item_type	The item cannot be assembled (since it has no requirements).
failed_tools	Some tool is missing.
failed_item_amount	At least one required item is missing.
failed_capacity	The agent does not have enough free space to carry the assembled item (after required items have been removed).

assist_assemble

Marks the agent as an assistant for assembly.

If multiple agents could provide the same item for assembly, it is preferably taken from the agent that used the *assemble* action. If that agent cannot provide the item, the assistants are sorted by name (i.e. first by length and then lexicographically, as the last part of the name is traditionally their number) as provided in the server's team config. Then, the item is taken from the assistants in that order.

Example: Imagine *agentA4*, *agentA3* and *agentA20* want to assemble an item that requires 5 pieces of *item1*. Further, let all agents carry 2 pieces of *item1* and *agentA4* be the "main" assembler (i.e. the one that uses the *assemble* action). Then, the first 2 pieces of *item1* are taken from *agentA4* since it is the initiator. Another 2 pieces are taken from *agentA3* and the last one is taken from *agentA20* (since *agentA3*'s name is shorter).

No	Parameter	Meaning
0	Agent	Name of an agent who uses the assemble action and whom this agent should help.

Failure Code	Reason
failed_wrong_param	Not exactly 1 parameter has been given.
failed_unknown_agent	No agent by the given name is known.
failed_counterpart	The initiator's action has failed or is not assemble .
failed_tools	Some tool is missing.
failed_location	The given agent is too far away.

buy

Buys a number of items in a shop.

No	Parameter	Meaning
0	Item	Name of the item to buy.
1	Amount	How many items to buy.

Failure Code	Reason
failed_wrong_param	More or less than 2 parameters have been given.
failed_location	The agent is not in a facility.
failed_wrong_facility	The agent is not in a shop.
failed_unknown_item	No item by the given name is known.
failed_item_amount	The given amount is not an integer, less than 1, or greater than the shop's available quantity.
failed_capacity	The agent does not have enough free space to carry the items.

deliver_job

Delivers items towards the completion of a job. The agent is automatically drained of all items matching the job's remaining requirements.

No	Parameter	Meaning
0	Job	The name of the job to deliver items for.

Failure Code	Reason
failed_wrong_param	Not exactly 1 parameter has been given.
failed_unknown_job	No job by the given name is known.
failed_job_status	The given job is not active, or the job is an auction and has not been assigned yet or has not been assigned to the agent's team.
failed_location	The agent is not in the storage associated with the job.
successful_partial	Not really a failure. Items have been delivered but the job has not been completed by this action.
useless	The agent does not have any items to contribute to the job.

bid_for_job

Places a bid for an auction job. The bid has to be lower than the current lowest bid.

No	Parameter	Meaning
0	Job	Name of the job to bid on.
1	Bid	The bid to place.

Failure Code	Reason
failed_wrong_param	Not exactly 2 parameters have been given, or the bid is not a positive integer.
failed_unknown_job	No job by the given name is known.
failed_job_type	The job is not an auction.
failed_job_status	The job's auctioning phase is over.

post_job

Posts a job that the other teams may complete. Only regular jobs may be posted.

The number of jobs a team may have posted at any one time is limited. Also, a job cannot be retracted once posted.

A successfully posted job starts being active in the next step.

No	Parameter	Meaning
0	Reward	How much to pay if the job is completed successfully.
1	Duration	After how many steps the job should end.
2	Storage	The target storage for the job.
3, 5, 7, ...	Item	The name of an item required for the job.
4, 6, 8, ...	Amount	How many items to require.

Failure Code	Reason
failed_wrong_param	Less than 5 or an even number of parameters have been given.
	Reward or duration is less than 1.
failed_wrong_facility	No storage by the given name is known.
failed_job_status	The agent's team has reached its post limit.
failed_unkown_item	Some name given is not the name of an item.
failed_item_amount	Some amount given is not a positive integer.

dump

Destroys a number of items at a dump facility.

No	Parameter	Meaning
0	Item	Name of the item to destroy.
1	Amount	How many items to destroy.

Failure Code	Reason
failed_wrong_param	Not exactly 2 parameters have been given.
failed_location	The agent is not in a facility.
failed_wrong_facility	The agent is not at a dump location.
failed_unknown_item	No item by the given name is known.
failed_item_amount	The given amount is not a positive integer or more than the agent is carrying.

charge

Charges the agent's battery at a charging station.

No parameters.

Failure Code	Reason
failed_wrong_param	Parameters have been given.
failed_location	The agent is not in a facility.
failed_wrong_facility	The agent is not in a charging station.
failed_facility_state	The charging station is currently out of order due to a blackout.

recharge

Uses the agent's solar collectors to recharge its battery (slowly).

No parameters.

Failure Code	Reason
failed_wrong_param	Parameters have been given.

continue and skip

Follows an agent's route or does nothing if the agent has no route.

No parameters.

Failure Code	Reason
failed_wrong_param	Parameters have been given.
failed_no_route	The agent's route could not be followed any longer (charge may be too low).

abort

Does nothing and clears the agent's route (if it exists).

unknownAction

This action is substituted if an agent submitted an action of unknown type.

randomFail

This action is substituted if the agent's action randomly failed.

noAction

This action is substituted if the agent did not send an action in time.

gather

Gathers a resource from a resource node.

No parameters.

Failure Code	Reason
failed_wrong_param	Parameters have been given.
failed_location	The agent is not in a facility.
failed_wrong_facility	The agent is not in a resource node.
failed_capacity	The agent does not have enough free space to carry the resource.
partial_success	Not really a failure. The action was executed successfully but there was no resource found.

B.2 Environment Percepts

The parameters for all the percepts can be found at <https://github.com/agentcontest/massim/blob/master/docs/eismassim.md>. Detailed description of the XML send between the server and client can also be found in <https://github.com/agentcontest/massim/blob/master/docs/scenario.md>.

SIM-START percepts

The following percepts might be included in a SIM-START message:

- `id(simId)`
 - `simId` : Identifier - name of the simulation
- `map(mapName)`
 - `mapName` : Identifier - name of the current map
- `seedCapital(sc)`
 - `sc` : Numeral - seed capital of any team in the simulation
- `steps(stepNumber)`
 - `stepNumber` : Numeral - number of steps the simulation will take
- `team(name)`
 - `name` : Identifier - name of the agent's team
- `role(name, speed, load, battery, [tool1, ...])`
 - represents the agent's role
 - `name` : Identifier - name of the role
 - `speed` : Numeral - speed of the role
 - `load` : Numeral - carrying capacity of the role
 - `battery` : Numeral - maximum battery charge of the role
 - `tool1` : Identifier - a tool usable by the role (list might be empty)
- `item(name, volume, tools([tool1, ...]), parts([item1, qty1, ...]))`
 - represents an item type in the simulation
 - `name` : Identifier - name of the item
 - `volume` : Numeral - the item's volume
 - `tools` : Function - all tools required to assemble the item
 - * `tool1` : Identifier - one of the tools required for assembly
 - `parts` : Function - all quantities of items required for assembly
 - * `item1` : Identifier - the first item required for assembly
 - * `qty1` : Numeral - quantity of 'item1' required for assembly
- `min,maxLat,Lon(coordinate)`
 - `coordinate`: Numeral - one of the 4 map bounds

REQUEST-ACTION percepts

The following percepts might be included in a REQUEST-ACTION message. Most of them should be self-explanatory.

- actionID(id)
 - id : Numeral - current action-id to reply with
- timestamp(time)
 - time : Numeral - server time the message was created at
- deadline(time)
 - time : Numeral - timepoint at which the action must be available
- step(number)
 - number : Numeral - the current step
- charge(ch)
 - ch : Numeral - agent's current battery charge
- load(cap)
 - cap : Numeral - agent's currently used capacity
- lat(d)
 - d : Numeral - latitude of the agent's location
- lon(d)
 - d : Numeral - longitude of the agent's location
- routeLength(ln)
 - ln : Numeral - length of the agent's current route
- money(m)
 - m : Numeral - the agent's team's current money
- facility(f)
 - f : Identifier - name of the agent's current facility
- lastAction(type)
 - type : Identifier - name of the last executed action

- `lastActionParams([param1, ...])`
 - `param1` : Identifier - first parameter of the last executed action (list might be empty)
- `lastActionResult(result)`
 - `result` : Identifier - result of the last executed action
- `hasItem(name, qty)`
 - `name` : Identifier - name of a carried item
 - `qty` : Numeral - carried quantity
- `route([wp(index, lat, lon), ...])`
 - represents the agent's current route
 - `wp` : Function - a waypoint in the route
 - * `index` : Numeral - index of the waypoint
 - * `lat` : Numeral - latitude of the waypoint
 - * `lon` : Numeral - longitude of the waypoint
- `entity(name, team, lat, lon, role)`
 - `name` : Identifier - name of an entity/agent in the simulation
 - `team` : Identifier - name of that entity's team
 - `lat` : Numeral - latitude
 - `lon` : Numeral - longitude
 - `role` : Identifier - that entity's role
- `chargingStation(name, lat, lon, rate)`
 - `name` : Identifier
 - `lat` : Numeral
 - `lon` : Numeral
 - `rate` : Numeral - the station's charging rate
- `dump(name, lat, lon)`
 - `name` : Identifier
 - `lat` : Numeral
 - `lon` : Numeral
- `shop(name, lat, lon, restock, [item(name1, price1, qty1), ...])`

- name : Identifier - the shop's name
- lat : Numeral
- lon : Numeral
- restock : Numeral - number of steps between restocks
- item : Function - an item stocked in the shop
 - * name1 - Identifier : that item's name
 - * price1 - Numeral : the item's price in this shop
 - * qty1 - Numeral : the quantity available in this shop
- storage(name, lat, lon, cap, used, [item(name1, stored1, delivered1), ...])
 - name : Identifier - the storage's name
 - lat : Numeral
 - lon : Numeral
 - cap : Numeral - the storage's total capacity
 - used : Numeral - the used capacity of the storage
 - item : Function - an item available in this storage
 - * name1 : Identifier - that item's name
 - * stored1 : Numeral - quantity stored by the agent's team
 - * delivered1 : Numeral - quantity delivered by or for the agent's team (see Storage section)
- workshop(name, lat, lon)
 - name : Identifier
 - lat : Numeral
 - lon : Numeral
- resourceNode(name, lat, lon, resource)
 - name : Identifier
 - lat : Numeral
 - lon : Numeral
 - resource : Identifier - name of the item that can be gathered at the node
- job(id, storage, reward, start, end, [required(name1, qty1), ...])
 - represents a non-auction job (excluding those posted by the agent's team)
 - id : Identifier - the job's ID

- storage : Identifier - name of the storage associated with this job
- reward : Numeral
- start : Numeral
- end : Numeral
- required : Function - an item required to complete the job
 - * name1 : Identifier - name of that item
 - * qty1 : Numeral - required quantity
- posted(id, storage, reward, start, end, [required(name1, qty1), ...])
 - represents a job posted by the agent's team (parameters are the same as for job)
- auction(id, storage, reward, start, end, fine, bid, time, [required(name1, qty1), ...])
 - same parameters as job plus:
 - * fine : Numeral - amount to pay if the auction is assigned but not completed
 - * bid : Numeral - the current lowest bid; might update each step during auction time
 - * time : Numeral - number of steps the auction phase will take
- mission(id, storage, reward, start, end, fine, bid, time, [required(name1, qty1), ...])
 - same parameters as auction
 - * reward and bid are the same

SIM-END percepts

The following percepts might be included in a SIM-END message:

- ranking(r)
 - r : Numeral - the final ranking of the agent's team
- score(s)
 - s : Numeral - the final score of the agent's team

Roles

The roles in the scenario can be configured under the top-level *roles* key in the simulation JSON object (which is one element of the ‘match’ array).

```
"roles" : {
  "car" : {
    "speed" : 3,
    "load" : 550,
    "battery" : 500,
    "roads" : ["road"]
  },
  "drone" : {
    "speed" : 5,
    "load" : 100,
    "battery" : 250,
    "roads" : ["air"]
  },
  "motorcycle" : {
    "speed" : 4,
    "load" : 300,
    "battery" : 350,
    "roads" : ["road"]
  },
  "truck" : {
    "speed" : 2,
    "load" : 3000,
    "battery" : 1000,
    "roads" : ["road"]
  }
}
```

Each role has its name as key and the following parameters:

- **speed**: how many ‘units’ the agent can move in one step
- **load**: how much volume the agent may carry
- **battery**: the agent’s battery size
- **roads**: which roads the agent can navigate (currently ‘road’ for all roads and ‘air’ for travelling linear distances between two points)

APPENDIX C

Source Code

C.1 Jason MAS Configuration File

Below can our configuration file for the system be found. It can be seen, that CArtAgO's environment extension is used to support artifacts, while each agent also uses the CArtAgO agent class. As no agents have special roles, they all uses the same base source file.

```
MAS multiagent_jason {  
  
    environment: c4jason.CartagoEnvironment  
  
    agents:  
        initializer agentArchClass c4jason.CAgentArch;  
        agent      agentArchClass c4jason.CAgentArch #28;  
  
    aslSourcePath: "src/asl";  
}
```

C.2 Jason Source Code

The Jason source code have been divided into different source files for clarity. General plans and rules that all agents use, have been put into its own files, which each of the agents then imports. Protocols for coordination which helps with giving items and assisting assembling, has also been put into its own folder. All Jason files have the .asl extension, meaning *AgentSpeak Language*.

agent.asl

The general code for all agents are included in the *agent.asl*. As it can be seen at the top of this file, many other files containing general plans and rules are included.

```
{ include("connections.asl") }
{ include("rules.asl") }
{ include("plans.asl") }
{ include("protocols.asl") }
{ include("requests.asl") }

// Initial beliefs
free.

// Initial goals
!register.
!focusArtifacts.

+task(JobId, Items, Storage, "mission", CNPId) : free <-
    !doIntention(newTask(JobId, Items, Storage, "mission", CNPId)).
+task(JobId, Items, Storage, Type, CNPId) :
    free & canSolve(Items) & Type \== "auction" <-
    !doIntention(newTask(JobId, Items, Storage, Type, CNPId)).

+!newTask(JobId, Items, Storage, Type, CNPId) : getBid(Items, Bid)
    <-
    bid(Bid)[artifact_id(CNPId)];
    winner(Won)[artifact_id(CNPId)];
    if (Won) {
        jia.getBaseVolume(Items, V);
        ?capacity(C);
        .print(JobId, " ", V, " ", C);
        clear("task", 5, CNPId);
        !deliverJob(JobId, Items, Storage);
    }.

+!doIntention(_) : not free <- .print("Illegal execution").
+!doIntention(Intention) <- -free; !Intention; +free.

+!doAction(Action) : .my_name(Me) <- jia.action(Me, Action); .wait
    ({+step(_)}).
```

```

+step(0) <- !doIntention(acquireTools).
+step(X) : lastAction(A) & A = "deliver_job" & lastActionResult(R)
          & R = "successful"
          & lastActionParam(P) <- .print(R, " ", A, " ", P);
          incJobCompletedCount.
+step(X) : lastActionResult(R) & lastAction(A) & lastActionParam(P)
          & not A = "goto" & not A = "noAction" & not A = "charge" &
          not A = "buy"
          & not A = "assist_assemble" <- .print(R, " ", A, " ", P).
+step(X) : lastActionResult(R) & not lastActionResult("successful")
          & lastAction(A) & lastActionParam(P) <- .print(R, " ", A,
          " ", P).

+reset <- .print("resetting"); .drop_all_desires; .drop_all_events;
          .drop_all_intentions; -reset.

```

rules.asl

Agents can use rules abstract some of the calculation away from the individual plans. For example, agents have a rule to calculate the number of steps it takes to reach its current destination, along with rules for if it has enough charge.

```

// Rules
speed(S)          :- myRole(Role) & role(Role, S, _, _, _).
maxLoad(L)        :- myRole(Role) & role(Role, _, L, _, _).
maxCharge(C)      :- myRole(Role) & role(Role, _, _, C, _).
tools(T)          :- myRole(Role) & role(Role, _, _, _, T).
canUseTool(T)     :- tools(Tools) & .member(T, Tools).
canUseAll(Req)    :- tools(Tools) & .findall(T, .member(T, Req) & .
                    member(T, Tools), Use) &
                    .length(Req, N) & .length(Use, N).
canUseAndCarry(T) :- canUseTool(T) & canCarry([T]).
canUseAndCarry(T, Me) :- canUseTool(T) & canCarry([T]) & .my_name
                    (Me).
// Facility types
isChargingStation(F) :- .substring("chargingStation", F).
isWorkshop(F)        :- .substring("workshop", F).
isStorage(F)         :- .substring("storage", F).
isShop(F)            :- .substring("shop", F).
isDump(F)            :- .substring("dump", F).
// In facility
inChargingStation    :- inFacility(F) & isChargingStation(F).
inWorkshop           :- inFacility(F) & isWorkshop(F).
inStorage            :- inFacility(F) & isStorage(F).
inShop              :- inFacility(F) & isShop(F).
inShop(F)            :- inFacility(F) & inShop.
inDump              :- inFacility(F) & isDump(F).
// Utility
routeDuration(D)     :- routeLength(L) & speed(S) & D = math.ceil(L
                    / S).
capacity(C)          :- maxLoad(M) & load(L) & C = M - L.

```

```

canMove                :- charge(X) & X >= 10.
chargeThreshold(X)     :- maxCharge(C) & X = 0.35 * C.
enoughCharge           :- routeLength(L) & enoughCharge(L).
enoughCharge(L)        :- speed(S) & charge(C) & chargeThreshold(
    Threshold) &
                        Steps = math.ceil(L / S) & Steps <= (C -
                        Threshold) / 10.
// Internal utility actions
canCarry(Items)         :- capacity(C) & jia.getBaseVolume(
    Items, V) & V <= C.
canSolve(Items)         :- capacity(C) & jia.getLoadReq(
    Items, R) & R <= C.
getBid(Items, Bid)      :- capacity(C) & speed(S) & jia.
    getBaseVolume(Items, V) &
                        .min([C, V], Min) & Bid = -S-
                        Min.
// Internal actions
getInventory(Inventory) :- .my_name(Me) & getInventory(Me,
    Inventory).
getInventory(Agent, Inventory) :- jia.getInventory(Agent,
    Inventory).
hasItems(Items)         :- .my_name(Me) & hasItems(Me,
    Items).
hasItems(Agent, Items)  :- jia.hasItems(Agent, Items).
hasBaseItems(Items)     :- .my_name(Me) & hasBaseItems(Me,
    Items).
hasBaseItems(Agent, Items) :- jia.hasBaseItems(Agent, Items).
hasAmount(Item, Amount) :- .my_name(Me) & hasAmount(Me,
    Item, Amount).
hasAmount(Agent, Item, Amount) :- jia.hasAmount(Agent, Item,
    Amount).
hasTools(Tools)         :- .my_name(Me) & hasTools(Me,
    Tools).
hasTools(Agent, Tools)  :- jia.hasTools(Agent, Tools).

contains(map(Item, X), [map(Item, Y) | _]) :- X <= Y.
contains(Item, [_ | Inventory])           :- contains(Item,
    Inventory).

getUsableTools([], [], []).
getUsableTools([H|T], [H|Y], N) :- canUseTool(H) & getUsableTools(T,
    Y, N).
getUsableTools([H|T], Y, [H|N]) :- getUsableTools(T, Y, N).

getInvTools(T, Y, N)      :- getInventory(I) & getInvTools(T,
    Y, N, I).
getInvTools([], [], [], _).
getInvTools([H|T], [H|Y], N, I) :- contains(map(H, 1), I) &
    getInvTools(T, Y, N, I).
getInvTools([H|T], Y, [H|N], I) :- getInvTools(T, Y, N, I).

```


plans.asl

The plans have the code for actually completing goals and thereby jobs. They include plans for buying and retrieving items, assembling, and delivering items.

```

+!deliverJob(Id, Items, F) : hasItems(Items) & inFacility(F) <- !
  doAction(deliVer_job(Id)).
+!deliverJob(Id, Items, F) : hasItems(Items) <- !
  getToFacility(F); !deliverJob(Id, Items, F).
+!deliverJob(Id, Items, F) <- !
  delegateJob(Id, Items, F).

+!delegateJob( _, [], _).
+!delegateJob(Id, Items, F) : canCarry(Items)
  <- !solveJob(Id, Items, F).
+!delegateJob(Id, Items, F) : jia.delegateJob(Id, Items, F, Rest)
  <- !delegateJob(Id, Rest, F).
+!delegateJob(Id, Items, F) : capacity(C)
  <-
    getItemsToCarry(Items, C, ItemsToCarry, Rest);
    !solveJob(Id, ItemsToCarry, F);
    !delegateJob(Id, Rest, F).

// Pre-condition: Items can be carried by agent
//+!solveJob(Id, Items, Storage) : .print("Solving ", Id, " ",
  Items) & false.
+!solveJob(Id, Items, Storage) <-
  getClosestWorkshopToStorage(Storage, Workshop);
  getRequiredTools(Items, Tools);
  getBaseItems(Items, BaseItems);
  getShoppingList(BaseItems, ShoppingList);
  !delegateItems(ShoppingList, Workshop);
  !coordinateAssemble(Items, Tools, Workshop);
  !deliverJob(Id, Items, Storage).

+!delegateItems([ ], _).
+!delegateItems([map( _, [ ]) | ShopList], F) <-
  !delegateItems(ShopList, F).
+!delegateItems([map(Shop, Items) | [ ]], F) <-
  !retrieveItems(Shop, Items).
+!delegateItems([map(Shop, Items) | ShopList], F) : .my_name(Me)
  & jia.delegateItems(Shop, Items, F, Me, Agent, Carry, Rest) <-
  +assistant(Agent, Shop, Carry);
  !delegateItems([map(Shop, Rest) | ShopList], F).
+!delegateItems([map(Shop, Items) | ShopList], F) <-
  !retrieveItems(Shop, Items);
  !delegateItems(ShopList, F).

+!retrieveItems( _, Items) : hasItems(Items).
+!retrieveItems(Shop, Items) : inShop(Shop) <- !buyItems(Items)
  .
//+!retrieveItems(Shop, Items) : .print("Retrieving: ", Shop, " ",
  Items) & false.
+!retrieveItems(Shop, Items) <- !getToFacility(Shop); !

```

```

    retrieveItems(Shop, Items).

// Pre-condition: In workshop and all base items available.
// Post-condition: Items in inventory.
+!assembleItems(Items) : hasItems(Items).
+!assembleItems([map(Name, _) | Items]) : jia.getReqItems(Name,
    []) <-
    !assembleItems(Items).
+!assembleItems([map(Name, Amount) | Items]) : jia.getReqItems(Name,
    ReqItems) <-
    !assembleItem(map(Name, Amount), ReqItems);
    !assembleItems(Items).

// Pre-condition: In workshop and base items available.
// Post-condition: Amount of Item with Name in inventory.
+!assembleItem(Item, _) : hasItems([Item]).
+!assembleItem(map(Name, Amount), ReqItems) : hasItems(ReqItems) <-
    !doAction(assemble(Name));
    !assembleItem(map(Name, Amount), ReqItems).
+!assembleItem(map(Name, Amount), ReqItems) <-
    !assembleItems(ReqItems);
    !doAction(assemble(Name));
    !assembleItem(map(Name, Amount), ReqItems).

+!coordinateAssemble(Items, [], F) <-
    !getToFacility(F);
    !initiateAssembleProtocol(Items).
+!coordinateAssemble(Items, Tools, _) : not assistant(_, _, _) &
    hasTools(Tools).
+!coordinateAssemble(Items, Tools, F) : not assistant(_, _, _) &
    getInventory(Inv) <-
    getMissingTools(Tools, Inv, MissingTools);
    !getToFacility(F);
    !delegateTools(MissingTools, F);
    .wait(inFacility(F));
    !initiateAssembleProtocol(Items).
+!coordinateAssemble(Items, Tools, F) : getInventory(Inv) <-
    .findall(I, assistant(X, _, _) & getInventory(X, I), AllInv);
    collectInventories([Inv | AllInv], Inventory);
    getMissingTools(Tools, Inventory, MissingTools);
    !getToFacility(F);
    !delegateTools(MissingTools, F);
    .wait(inFacility(F));
    !initiateAssembleProtocol(Items).

+!delegateTools([], _).
+!delegateTools(Tools, F) : .my_name(Me) & .print("delegateTools:
    ", Tools)
    & jia.delegateTools(Tools, F, Me, Agent, Carry, Rest) <-
    +assistant(Agent, "tool", Carry);
    !delegateTools(Rest, F).
+!delegateTools(Tools, F) <-
    .wait({+step(_)});
    !delegateTools(Tools, F).

```

```

+!coordinateAssist(Workshop, Agent) <-
  !getToFacility(Workshop);
  !acceptAssembleProtocol(Agent).

+!acquireTools : tools(Tools) <-
  sortByPermissionCount(Tools, SortedTools);
  for (.member(T, SortedTools)) {
    getToolVolume(T, V); ?capacity(C);
    if (C >= V) { !retrieveTool(T); }
  }.
-!acquireTools <- .wait(100); !acquireTools.

+!retrieveTools([]).
+!retrieveTools([Tool|Tools]) <-
  !retrieveTool(Tool);
  !retrieveTools(Tools).

+!retrieveTool(Tool) : hasTools([Tool]).
+!retrieveTool(Tool) <-
  getClosestShopSelling(Tool, Shop);
  !retrieveItems(Shop, [map(Tool, 1)]).

// Post-condition: Empty inventory or -assemble.
+!assistAssemble( _ ) : load(0) | not assemble.
+!assistAssemble(Agent) <-
  !doAction(assist_assemble(Agent));
  .wait(1000); // To allow assembler to remove assemble
  !assistAssemble(Agent).

// Pre-condition: In shop and shop selling the items.
// Post-condition: Items in inventory.
+!buyItems([]) <- !doAction(skip). // To prevent duplicate
  purchases.
+!buyItems(Items) : hasItems(Items).
+!buyItems([Item|Items]) : hasItems([Item]) <- !buyItems(Items).
+!buyItems([map(Item, Amount)|Items]) <-
  ?hasAmount(Item, HasAmount); ?inShop(Shop);
  getAvailableAmount(Item, Amount - HasAmount, Shop,
    AmountAvailable);
  !doAction(buy(Item, AmountAvailable));
  !buyItems(Items);
  !buyItems([map(Item, Amount)]).

// Post-condition: In facility F.
+!getToFacility(F) : inFacility(F).
+!getToFacility(F) : not canMove
  <- !doAction(recharge); !getToFacility(F).
+!getToFacility(F) : not enoughCharge & not isChargingStation(F)
  <- !charge; !getToFacility(F).
+!getToFacility(F)
  <- !doAction(goto(F)); !getToFacility(F).

// Post-condition: Full charge.
+!charge : charge(X) & maxCharge(X).
+!charge : inChargingStation <- !doAction(charge); !charge.

```

```

+!charge <- getClosestFacility("chargingStation", F); !
  getToFacility(F); !charge.

```

requests.asl

The request event handlers for the agent are what distributes the work to the individual agents. These handlers will take new requests that have been announced, bid on them if necessary, and begin to complete the job if the agent has won the task. These handlers will also ensure that the agent begin doing something as soon as they are free, assuming there is something for them to do.

```

+assembleRequest(JobId, Items, Storage, CNPId) : free <-
  !!doIntention(assembleRequest(JobId, Items, Storage, CNPId)).

+retrieveRequest(Shop, Items, Workshop, Agent, CNPId) : free <-
  !!doIntention(retrieveRequest(Shop, Items, Workshop, Agent,
    CNPId)).

+toolRequest(Tools, Workshop, Agent, CNPId) : free <-
  !!doIntention(toolRequest(Tools, Workshop, Agent, CNPId)).

+auction(TaskId, CNPId) : free <-
  -free;
  takeTask(Can)[artifact_id(CNPId)];
  if (Can)
  {
    getBid(TaskId, Bid);
    if (Bid \== 0) { !doAction(bid_for_job(TaskId, Bid)); }
  }
  +free.

+!assembleRequest(JobId, Items, Storage, CNPId)
  : capacity(Capacity) & speed(Speed) <-
  getItemsToCarry(Items, Capacity, ItemsToAssemble, AssembleRest)
  ;
  getBaseVolume(ItemsToAssemble, Volume);

  // Negative volume since lower is better
  Bid = -Speed - Volume;

  if (not ItemsToRetrieve = [])
  {
    bid(Bid, AssembleRest)[artifact_id(CNPId)];
    winner(Won)[artifact_id(CNPId)];

    if (Won)
    {
      .print("Job: ", JobId, "Assembling: ", ItemsToAssemble)
      ;
      !solveJob(JobId, ItemsToAssemble, Storage);
    }
  }
}

```

```

+!retrieveRequest(Shop, Items, Workshop, Agent, CNPID)
: capacity(Capacity) & speed(Speed) <-
  getItemsToCarry(Items, Capacity, ItemsToRetrieve, RetrieveRest)
;
getVolume(ItemsToRetrieve, Volume);
distanceToFacility(Shop, Distance);

// Negative volume since lower is better
Bid = math.ceil(Distance/Speed) * 10 - Volume;

if (not ItemsToRetrieve = [])
{
  bid(Bid, ItemsToRetrieve, RetrieveRest)[artifact_id(CNPId)
    ];
  winner(Won)[artifact_id(CNPId)];

  if (Won)
  {
    .print("Helping ", Agent);
    !retrieveItems(Shop, ItemsToRetrieve);
    !coordinateAssist(Workshop, Agent);
  }
}.

+!toolRequest(Tools, Workshop, Agent, CNPID)
: capacity(Capacity) & speed(Speed) <-
?getUsableTools(Tools, Usable, NotUsable);
?getInvTools(Usable, InInv, NotInInv);

getToolsToCarry(NotInInv, Capacity, ToolsToRetrieve,
  RetrieveRest);
.concat(InInv, ToolsToRetrieve, AllTools);
.concat(NotUsable, RetrieveRest, RestTools);

if (not AllTools = [])
{
  getToolsVolume(AllTools, Volume);
  Bid = -Speed - Volume;

  bid(Bid, AllTools, RestTools)[artifact_id(CNPId)];
  winner(Won)[artifact_id(CNPId)];

  if (Won)
  {
    .print("Helping ", Agent, " with ", AllTools);
    !retrieveTools(ToolsToRetrieve);
    !coordinateAssist(Workshop, Agent);
  }
}.

```

protocols.asl

The *protocols* source file contains the different plans for coordination between agents, such as giving items from one agent to another and helping each other with item assembling.

```
+give(Items, InitStep)[source(Agent)] : not free <-
    !addIntentionFirst(acceptReceiveProtocol(Agent, Items, InitStep
    )).

+give(Items, InitStep)[source(Agent)] <-
    -free; !acceptReceiveProtocol(Agent, Items, InitStep); +free.

+receive(Items, InitStep)[source(Agent)] : not free <-
    !addIntentionFirst(acceptGiveProtocol(Agent, Items, InitStep)).

+receive(Items, InitStep)[source(Agent)] <-
    -free; !acceptGiveProtocol(Agent, Items, InitStep); +free.

+!initiateReceiveProtocol(Agent, Items) : step(MyStep) <-
    .send(Agent, tell, give(Items, MyStep));
    .wait(readyToGive(ReadyStep));
    .print("Waiting for step ", ReadyStep, " to retrieve ", Items);
    .wait(step(ReadyStep));
    !receiveItems(Items).

+!acceptReceiveProtocol(Agent, Items, InitStep) : not hasItems(
    Items) <-
    !addIntentionLast(acceptReceiveProtocol(Agent, Items, InitStep)
    ).

+!acceptReceiveProtocol(Agent, Items, InitStep) : step(MyStep) <-
    .max([InitStep, MyStep], MaxStep);
    ReadyStep = MaxStep + 1;
    .send(Agent, tell, readyToGive(ReadyStep));
    .print("Waiting for step ", ReadyStep, " to give ", Items);
    .wait(step(ReadyStep));
    !giveItems(Agent, Items).

+!initiateGiveProtocol(Agent, Items) : step(MyStep) <-
    .send(Agent, tell, receive(Items, MyStep));
    .wait(readyToReceive(ReadyStep));
    .print("Waiting for step ", ReadyStep, " to give ", Items);
    .wait(step(ReadyStep));
    !giveItems(Agent, Items).

+!acceptGiveProtocol(Agent, Items, InitStep) : step(MyStep) <-
    .max([InitStep, MyStep], MaxStep);
    ReadyStep = MaxStep + 1;
    .send(Agent, tell, readyToReceive(ReadyStep));
    .print("Waiting for step ", ReadyStep);
    .wait(step(ReadyStep));
    !receiveItems(Items).

+!initiateAssembleProtocol(Items) <-
```

```

.wait(.count(assistant(_, _, _), N)
      & .count(assistantReady(_), N));

for (assistant(A, _, _)) {
    .send(A, tell, assemble);
}

!assembleItems(Items);

for (assistant(A, _, _)) {
    -assistant(A, _ , _);
    .send(A, untell, assemble);
}.

+!acceptAssembleProtocol(Agent) : .my_name(Me) <-
    .send(Agent, tell, assistantReady(Me));

    .wait(assemble);

    !assistAssemble(Agent);

    .send(Agent, untell, assistantReady(Me)).

```

initializer.asl

The initializer is a special agent, that does not appear in the environment, but only lives in the beginning of each simulation. Its job is to setup artifacts for other agents, and reset every remove all data from other agents beliefs when the system resets.

```

!init.

+!init : .my_name(Me) <-
    makeArtifact("EIArtifact", "env.EIArtifact", [], Id);
    makeArtifact("ItemArtifact", "info.ItemArtifact", [], _);
    makeArtifact("FacilityArtifact", "info.FacilityArtifact", [], _
    );
    makeArtifact("StaticInfoArtifact", "info.StaticInfoArtifact",
    [], _);
    makeArtifact("DynamicInfoArtifact", "info.DynamicInfoArtifact",
    [], _);
    makeArtifact("JobArtifact", "info.JobArtifact", [], _);
    makeArtifact("TaskArtifact", "cnp.TaskArtifact", [], TaskId);
    focus(Id);
    focus(TaskId).

+reset <-
    for (assembleRequest(_, _, _, _, _, CnpId)) { clear("
        assembleRequest", 6, CnpId); };
    for (retrieveRequest(_, _, _, CnpId))          { clear("
        retrieveRequest", 4, CnpId); };
    -reset.

```

C.3 Java Source Code

Here are the `EIArtifact` class included, which are the central part of the Multi-Agent System. It handles the connection between the agents and the environment, ensuring percepts are passed into the correct artifacts.

```
package env;
// Environment code for project multiagent_jason

import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
import java.util.logging.Level;
import java.util.logging.Logger;

import cartago.Artifact;
import cartago.INTERNAL_OPERATION;
import cartago.OPERATION;
import eis.AgentListener;
import eis.EnvironmentInterfaceStandard;
import eis.iilang.Action;
import eis.iilang.Identifier;
import eis.iilang.Parameter;
import eis.iilang.Percept;
import eis.iilang.PrologVisitor;
import info.AgentArtifact;
import info.DynamicInfoArtifact;
import info.FacilityArtifact;
import info.ItemArtifact;
import info.JobArtifact;
import info.StaticInfoArtifact;
import logging.LoggerFactory;
import massim.eismassim.EnvironmentInterface;
import massim.scenario.city.data.Role;

public class EIArtifact extends Artifact {

    private static final Logger logger = Logger.getLogger(
        EIArtifact.class.getName());

    public static final boolean LOGGING_ENABLED = false;

    private static EnvironmentInterfaceStandard ei;
    private static final String TEAM_A = "conf/eismassimconfig.json";
    private static final String TEAM_B = "conf/
        eismassimconfig_team_B.json";
```



```

private String configFile = TEAM_A;

private static Map<String, String> connections = new HashMap
    <>();
private static Map<String, String> entities    = new HashMap
    <>();

private String team;

/**
 * Instantiates and starts the environment interface.
 */
void init()
{
    logger.setLevel(Level.SEVERE);
    logger.info("init");

    try
    {
        ei = new EnvironmentInterface(configFile);

        // Get the team name from EI. Should be a better way
        this.team = ((String) (ei.getEntities().toArray())[0]).
            substring(10, 11);

        fileLogger = LoggerFactory.createFileLogger(team);

        ei.start();
    }
    catch (Throwable e)
    {
        logger.log(Level.SEVERE, "Failure in init: " + e.
            getMessage(), e);
    }
}

@OPERATION
void register()
{
    String agentName    = getOpUserName();
    String id           = agentName.substring(5);
    String connection   = "connection" + team + id;
    String entity       = "agent" + team + id;

    logger.fine("register " + agentName + " on " + connection);

    try
    {
        ei.registerAgent(agentName);

        ei.associateEntity(agentName, connection);

        connections .put(agentName, connection);
        entities     .put(agentName, entity);
    }
}

```

```

        if (connections.size() == ei.getEntities().size())
        {
            // Attach listener for perceiving the following
            // steps
            ei.attachAgentListener(agentName, new AgentListener
            ()
            {
                @Override
                public void handlePercept(String agentName,
                    Percept percept)
                {
                    if (percept.getName().equals("simStart"))
                    {
                        execInternalOp("perceiveInitial");
                    }
                    else if (percept.getName().equals("simEnd")
                        )
                    {
                        System.out.println("This is the end!");
                        System.out.println(percept);
                    }
                    else if (percept.getName().equals("step"))
                    {
                        execInternalOp("perceiveUpdate");
                    }
                }
            });
        }
    }
}
catch (Throwable e)
{
    logger.log(Level.SEVERE, "Failure in register: " + e.
        getMessage(), e);
}
}

public static void performAction(String agentName, Action
    action)
{
    logger.fine("Step " + DynamicInfoArtifact.getStep() + ": "
        + agentName + " doing " + action);

    try
    {
        if (action.getName().equals("assist_assemble"))
        {
            String name = PrologVisitor.staticVisit(action.
                getParameters().get(0));

            LinkedList<Parameter> params = new LinkedList<>();
            params.add(new Identifier(EIArtifact.getAgentName(
                name)));

            action.setParameters(params);
        }
    }
}

```

```

        ei.performAction(agentName, action);
    }
    catch (Throwable e)
    {
        logger.log(Level.SEVERE, "Failure in performAction: " +
            e.getMessage(), e);
    }
}

@INTERNAL_OPERATION
void perceiveInitial()
{
    logger.finest("perceiveInitial");
    if (DynamicInfoArtifact.getStep() == StaticInfoArtifact.
        getSteps() - 1)
    {
        this.reset();
    }

    try
    {
        Set<Percept> allPercepts = new HashSet<>();

        Map<String, Collection<Percept>> agentPercepts = new
            HashMap<>();

        for (Entry<String, String> entry : connections.entrySet
            ())
        {
            String agentName = entry.getKey();

            Collection<Percept> percepts = ei.getAllPercepts(
                agentName).get(entry.getValue());

            agentPercepts.put(agentName, percepts);

            allPercepts.addAll(percepts);
        }

        // Perceive static info
        ItemArtifact      .perceiveInitial(allPercepts);
        StaticInfoArtifact .perceiveInitial(allPercepts);
        // Perceive dynamic info
        FacilityArtifact   .perceiveUpdate(allPercepts);
        DynamicInfoArtifact .perceiveUpdate(allPercepts);
        JobArtifact        .perceiveUpdate(allPercepts);

        // Define roles
        for (Role role : StaticInfoArtifact.getRoles())
        {
            defineObsProperty("role", role.getName(), role.
                getSpeed(), role.getMaxLoad(),
                role.getMaxBattery(), role.getPermissions()
                    .toArray());
        }
    }
}

```

```

    }

    // Perceive agent info
    for (Entry<String, Collection<Percept>> entry :
        agentPercepts.entrySet())
    {
        String agentName = entry.getKey();

        AgentArtifact.getAgentArtifact(agentName).
            perceiveInitial(entry.getValue());
    }

    // Define step
    defineObsProperty("step", DynamicInfoArtifact.getStep()
        );

    FacilityArtifact.announceShops();
}
catch (Throwable e)
{
    logger.log(Level.SEVERE, "Failure in perceive: " + e.
        getMessage(), e);
}

logger.finest("Perceive initial done");
}

@INTERNAL_OPERATION
void perceiveUpdate()
{
    logger.finest("perceiveUpdate");

    try
    {
        Set<Percept> allPercepts = new HashSet<>();

        for (Entry<String, String> entry : connections.entrySet()
            ())
        {
            Collection<Percept> percepts = ei.getAllPercepts(
                entry.getKey()).get(entry.getValue());

            AgentArtifact.getAgentArtifact(entry.getKey()).
                perceiveUpdate(percepts);

            allPercepts.addAll(percepts);
        }

        FacilityArtifact        .perceiveUpdate(allPercepts);
        DynamicInfoArtifact     .perceiveUpdate(allPercepts);
        JobArtifact              .perceiveUpdate(allPercepts);

        getObsProperty("step").updateValue(DynamicInfoArtifact.
            getStep());
    }
}

```

```

        logData();

        JobArtifact.announceJobs();
    }
    catch (Throwable e)
    {
        logger.log(Level.SEVERE, "Failure in perceive: " + e.
            getMessage(), e);
    }
}

private void reset()
{
    defineObsProperty("reset");

    DynamicInfoArtifact.reset();
    StaticInfoArtifact.reset();
    FacilityArtifact.reset();
    JobArtifact.reset();
    ItemArtifact.reset();

    for (Entry<String, String> entry : connections.entrySet())
    {
        AgentArtifact.getAgentArtifact(entry.getKey()).reset();
    }

    fileLogger = LoggerFactory.createFileLogger(team);

    removeObsProperty("step");
    for (Role role : StaticInfoArtifact.getRoles())
    {
        removeObsPropertyByTemplate("role", role.getName(),
            role.getSpeed(), role.getMaxLoad(),
            role.getMaxBattery(), role.getPermissions().
                toArray());
    }

    removeObsProperty("reset");
}

/**
 * @param entity
 * @return Get the name of the agent associated with the entity
 */
public static String getAgentName(String entity)
{
    return entities.get(entity);
}

private static Logger fileLogger;

private void logData()
{
    fileLogger.info("Step: " + DynamicInfoArtifact.getStep() +
        " - Money: " + DynamicInfoArtifact.getMoney());
}

```

```
        if (DynamicInfoArtifact.getStep() == StaticInfoArtifact.  
            getSteps() - 1)  
        {  
            fileLogger.info("Completed jobs: " +  
                DynamicInfoArtifact.getJobsCompleted());  
        }  
    }  
}
```