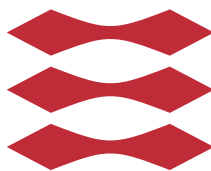


Multi-Agent Systems

Eirik Oterholm Nielsen and Martin Nielsen

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

Preface

This thesis was prepared at DTU Compute in fulfillment of the requirements for acquiring a BEng in IT.

The thesis deals with Multi-Agent Systems programming with AgentSpeak: Jason and the Multi-Agent Programming Contest of 2017.

The thesis has been developed over 12 weeks, from 08/04-2017 to 01/07-2017.

Kongens Lyngby, 01/07-2017

Eirik O. Nielsen
Martin Nielsen

Eirik Oterholm Nielsen and Martin Nielsen

Acknowledgements

We would like to thank Jørgen Villadsen for mentoring us.

Helge Hatteland and Oliver Fleckenstein for good advice.

Rafael H. Bordini and Jomi F. Hübner for clarifications about the Jason language, and for temporary workarounds regarding errors in the Multi-Agent Programming Contest.

Contents

Preface	i
Acknowledgements	iii
1 Introduction	1
2 Multi-Agent Systems	3
2.1 Agents in a Multi-Agent System	3
2.1.1 Belief	5
2.1.2 Desire	5
2.1.3 Intention	5
2.2 The Social Ability of Agents	5
3 The Multi-Agent Programming Contest	7
3.1 Scenario	7
3.2 Roles	8
3.3 Facilities	9
3.4 Jobs	10
3.5 Actions	10
3.6 Percepts	12
4 Analysis	13
4.1 Strategy	13
4.2 Advanced Strategy	14
5 The Jason Programming Language	17
5.1 Eclipse AgentSpeak Plugin	17
5.2 Project File	19
5.3 Agent .asl Files	19

5.4	Concepts of a Jason Agent	19
5.4.1	Actions	19
5.4.2	Beliefs and Rules	20
5.4.3	Plans and Events	20
5.4.4	The Reasoning Cycle	21
6	Setting Up the Project	25
6.1	Setting Up the Project from Scratch	25
6.2	Guide: Importing and Running the Project	26
7	Implementation	29
7.1	Implemented Minimum Viable Product	29
7.1.1	Cooperation	29
7.1.2	Find Job	30
7.1.3	Core Loop	30
7.1.4	Recharge	32
7.2	Improved Version	35
7.2.1	Improved Cooperation	35
7.2.2	Improved Choose Job	35
7.2.3	Improved Core Loop	36
7.2.4	Improved Recharge	37
7.3	Internal Action	40
7.4	Environment Interface	40
8	Discussion and Reflection	41
8.1	Results	41
8.2	Further Work	42
8.2.1	Route Planing	42
8.2.2	Charging Station	43
8.2.3	Missions and Auctions	43
8.2.4	Cooperation Between Agents	43
8.2.5	Prediction and Counter Play	43
8.2.6	Error Handling	44
8.3	Reflections	44
A	Code	45
A.1	Project file (.mas2j)	45
A.2	connectionA (Improved Multi-Agent System.asl)	46
A.3	connectionB (Minimum Viable Mullti-Agent System	57
A.4	Internal Action Code(Java)	65
A.5	Server (Java)	66
A.6	EISAdapter(Java)	66
	Bibliography	73

CHAPTER 1

Introduction

In this chapter we will give a short summary of what the goal of this project is and a short summary of each of the chapters of the thesis.

A multi-agent system is as the name suggests a system using multiple agents. Such a system uses independent agents that work together to fulfill tasks. Agents are often assigned roles, such as organizers, workers and so on.

While a multi-agent system might need at least one agent for each role, the systems are often scalable, functioning with many agents for each role.

Each agent in a multi-agent system is capable of sharing information and intentions, but each agent makes its own decisions. They might even have their own agendas. They are able to communicate, perceive the environment and affect it. This is how they work together or against each other.

We have in this project implemented a multi-agent system for the Multi-Agent Programming Contest [4], constructing and delivering goods while keeping their agents operational.

In chapter 2 we introduce common concepts of multi-agent systems and explain the basics of such a system. We finish by giving an example of two agents communicating together.

In chapter 3 we give an overview of the Multi-Agent Programming Contest simulation.

In chapter 4 we analyze the strategy aspect of the Multi-Agent Programming Contest and identify some key components of a solid strategy.

In chapter 5 we go through some of the concepts of the Jason programming language, which is our language of choice for the Multi-Agent Programming Contest.

In chapter 6 we explain how to set up and run our program and how to edit it in eclipse.

In chapter 7 we go through the finer points of our implementation of a multi-agent system.

In chapter 8 we will discuss the result of our project, give some ideas for further improvement of our multi-agent system and self reflection of the process.

Multi-Agent Systems

In this chapter we will introduce the basic ideas behind multi-agent systems. We discuss what properties an agent should have and how we model their basic decision making logic. For further theory on agents we refer to [9].

2.1 Agents in a Multi-Agent System

An agent perceives its environment and takes logical actions based upon what it has perceived. An agent might use sensors of some kind to perceive its environment. This includes optics, thermometers, GPS and so on. As the contest is entirely simulated on a server, we will be getting our percepts directly from said server.

Other than being in an environment, an agent should be the following: autonomous, proactive, reactive, and social.

Autonomy means that given a set of goals, an agent should be able to independently fulfill said goals, either through delegation or through action.

Being proactive means that an agent acts in accordance with its goals. It means

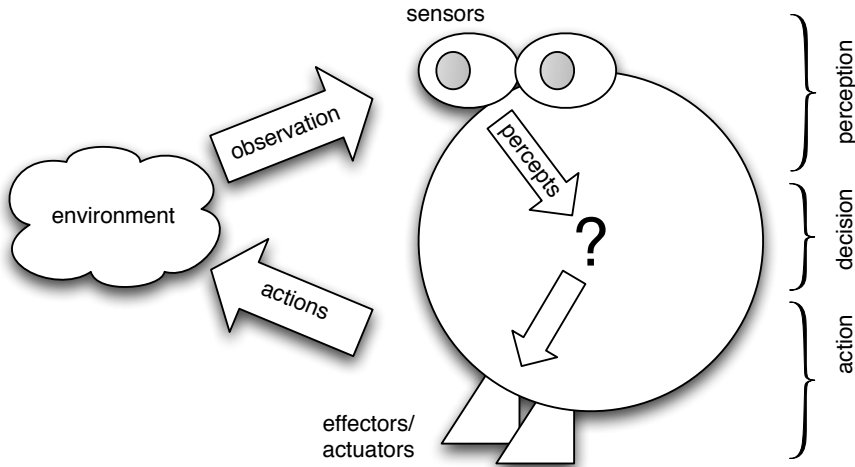


Figure 2.1: Illustration of how an agent perceives and interacts with the environment, figure courtesy of [6]

that once it has a goal, its actions will be to further said goal.

Being reactive means that an agent reacts to changes in its environment. If an agent's path is blocked, the agent either figures a way around the blockage, clears the blockage, or it might even decide to drop the current plan of action, choosing a different goal to fulfill. It means that an agent should not be locked into a bad plan and that it adapts to changes in its environment.

Social ability means that an agent is able to communicate with the other agents and we go more into detail on this subject in section 2.2.

An agent also has its own goals and will act upon them based upon what it believes to be true about the world around it. In Figure 2.1 we can see a simple model of how an agent works. It observes the environment using sensors, the observations become percepts and then beliefs. Based upon the internal logic of the agent it comes to a decision of what to do and decides on an action. It then uses actuators/effectors to affect its environment and the cycle continues.

One of the ways to describe this is with the BDI model. BDI stands for Belief, Desire, and Intention. It is a way to model multi-agent systems.

2.1.1 Belief

A belief is anything the agent believes is true about its environment. It is usually something gotten through an agent senses as seen in figure 2.1, or some logical conclusion of said percepts. It can also be the the result of communication with other agents, which is discussed further in section 2.2.

2.1.2 Desire

A desire is something an agent would like to if the opportunity arises. An agent might want to finish a delivery and recharge their batteries, but if it is stuck in a ditch somewhere its unlikely to ever be able to. A desire is sometimes called an option, it is something an agent would like to do, but it still requires opportunity and reason to act on its desire.

2.1.3 Intention

An intention is the product of belief and desire. It is what the program intends to do to its environment, based upon what it believes to be true about its environment and what it desires to happen.

2.2 The Social Ability of Agents

The key difference between multi-agent system and a regular system is that the agents are able to communicate and cooperate. This could be done through percepts, or some other language specific protocols. By doing this, the agents complete their goals [9].

Common ways of communication includes sending info, perhaps about a spotted resource node, giving an order, e.g. to go deliver an item, or asking for assistance to assemble or fetch items.

One of the more fascinating parts of this is that an agent might be wrong about something, or even lie if it suits its own goals. Depending on the multi-agent system, this might need to be taken into account, in order to have a plan if some piece of information turns out to be bad.



Figure 2.2: The biologist agent (left) tells a lie to the exterminator agent (right)

Lets say that we there are two agents, agent Ed and agent Bob. Agent Ed is an exterminator bot sent to a village to get rid of all bugs inside houses and is paid for each room he cleans. Bob is a biologist bot sent to examine and research houses and is paid for each room he examines. Both have only one day to complete the job. Ed the exterminator sees Bob the biologist coming out of a house. Ed asks Bob if there are any bugs in the house. Since it is on a tight schedule he needs to get through all the houses as quickly as possible and will skip the house if it believes it to be clean. Agent Bob found bugs in the house, but will get paid for inspecting the bugs, but if agent Ed clears the house, Bob will not get paid. Bob tells Ed that the house is clear of bugs. Agent Ed does not know what agent Bob wants, and because of the lie Ed now thinks that the subgoal of clearing the house it completed.

If however agent Ed ever enters this house, it finds out that the house had bugs. Agent Bob has thus been proven unreliable. Agent Ed could give it a 40-60 percentage chance split between agent Bob having broken sensors and being a liar, working with some unknown agenda. Either way, agent Ed now has reason to believe and should have a corresponding belief that agent Bob is unreliable and should verify any communication sent by agent Bob.

CHAPTER 3

The Multi-Agent Programming Contest

In this chapter we will discuss the details of the Multi-Agent Programming Contest. This includes the scenario, what kind of agents we have access to, percepts and actions [2].

3.1 Scenario

This years scenario of the Multi-Agent Programming Contest takes place on a terraformed Mars. The atmosphere is not breathable yet, so unmanned vehicles called *All Terrain Planetary Vehicles* or *ATPV*'s are sent out by entrepreneurs to carry out tasks and make money completing jobs outside. Each team starts with a seed money amount of 50,000.

There are two kinds of jobs. Regular jobs, which pay the first team which completes the job, and auction jobs, which can be completed by the team that offers to do the job for the least amount of money. Jobs may require acquisition, assembly, and transportation of goods.

Two teams compete at a time, with each team having a number of vehicles

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

Figure 3.1: The four roles of agents and their stats

Role\Tool	0	1	2	3	4	5	6	7
Drone		x	x			x		x
Motorcycle							x	
Car				x	x		x	
Truck	x			x				x

Figure 3.2: The tools which each role can use it marked which an 'x'

controlled by the teams multi-agent systems. The goal of the game is to make more money than the other team in the given amount of rounds. The map where the agents compete is from Open Street Map.

Each game runs for a given number of steps. Each step is the minimum unit of time in the game, so a vehicle with speed 10 can move 10 distance units in one step.

Each type of vehicle has a battery charge, which determines how long the vehicle can move without recharging, a capacity, which determines how much the vehicle can carry, and speed, which determines how far a vehicle can move in one step. Each simulation features a set of items, which are used to craft goods for jobs.

3.2 Roles

There are four kinds of autonomous vehicles 3.1. The drone is the most distinct from the others since it is airborne. This means that it does not have to follow the roads like the other vehicles, but can instead go in a direct line to any given

point within its range. Among the ground vehicles, the motorbike is a fast short range vehicle with a low capacity, the car is middling in all aspects and the truck is a slow long range vehicle with a large capacity. Each role is also able to use a different set of tools, used for the assembling of items. An overview of the tools that each role can use is seen in figure 3.2.

3.3 Facilities

There are four kinds of facilities. A number of facilities are randomly places on the map at the start of the simulation, each with a unique name and location. Facilities can be affected by blackouts and these facilities will appear as not working to the agents. The four different kinds of facilities are:

Shops: In shops the agent can buy items. Each shop has a limited quantity of each item, but shops are restocked once in a while. Items bought in shops are used to craft other items out of that might be required for a job.

Charging Stations: Are essential for the agents, since the agents only have a limited battery range. If the battery of an agent should die out while not at a charging station, they can recharge their battery slowly using solar power.

Workshops: In workshops items can be assembled from other items. Most items need certain tools for assembling. Tools can only be used by certain roles, so sometimes the agents need to cooperate in the workshop for assembling an item.

Dumps: Offer a place to destroy items in order to free up capacity of an agent.

Storage Facilities: Allow storage of items up to a set volume. Storage facilities are also where jobs are completed.

Resource Nodes: These nodes are places where items can be found. They take a certain number of gather actions to be found. If one team digs up the item first, the other team will not be able to get an item from the same node. The location of the resource nodes are not common knowledge, but must be found by the teams agents.

3.4 Jobs

In order to earn money, and thereby points, the teams need to complete jobs. There are three kinds of jobs:

Standard Jobs: Reward the first team to complete the job.

Action Jobs: Are assigned to the lowest bidding team for completion of the job.

Missions: Are jobs that both teams can complete without interfering with one another.

Jobs have a start time and an end time. They require a set of items to be delivered to a warehouse, and none of the items can be bought from a shop. They need to be assembled from other items. Jobs also have a reward for the team that fulfills them. A team is fined if they fail to complete missions or an auction jobs which they have won.

3.5 Actions

A full description of the actions can be found on the GitHub page of the Multi-Agent Programming Contest [2].

goto: The command to make an agent go to a desired location. It can be to a location given by coordinates or to a specific facility given by name. If the action is successful it consumes 10 energy charges.

give: Takes three parameters. The first one specifies the recipient, the second argument names the item, and the third argument specifies the amount of items to give.

receive: Puts the agent in a receiving state. If another agent gives something to it, it will be received.

store: If an agent is located at a storage facility, it calling the 'store' command stores items with the name and quantity given in the arguments of the call.

assemble: Assembles a specific item out of other items using tools. The items used for the assembly are consumed and cannot be reused. This must be done

at a workshop.

assist_assemble: Marks the agent as assisting another agent who is using the assemble action in the assembly.

buy: Buys a specified amount of an item. This must be done at a *shop*.

deliver_job: Delivers all items needed for completion of the specified job. The agent must be at the storage facility associated with the specified job.

bid_for_job: Places a bid on a specified action job.

post_job: Posts a job that other teams may complete.

dump: Used to dump a number of a specified item. This can only be done at a *dump*.

charge: Charges the agent's battery given it is at a charging station.

recharge: Uses solar panels on the agent to recharge its battery. This is slower than recharging at a charging station.

continue: Follows the agents route to the end and does nothing if the agent does not have a route.

skip: Same as continue.

abort: Aborts the agent's route.

unknownAction: Is sent if the agent uses an unknown action.

randomFail: Actions have a chance of randomly failing. If this occurs, this action is sent instead of the action that the agent was trying to perform.

noAction: Is sent if the agent did not send an action in time.

gather: Used to gather a resource node that the agent is located at.

3.6 Percepts

A number of percepts are available at simulation start such as the existence and locations of facilities, jobs, items, charge, and the roles of the agents. These are essential for the agents in order to be able to make plans to achieve their goals. A full list of the percepts can be found in the Eismassim description on the GitHub page of the Multi-Agent Programming Contest[1].

Analysis

This chapter gives an overview of the thoughts we have made about the strategy of the Multi-Agent Programming Contest and also which pursuits we found feasible to pursue in the limited time we have for implementing the system.

4.1 Strategy

We have thought about the strategy on two levels. Strategy in a perfect world, where we could achieve anything we wanted i.e. there was no barrier in terms of our knowledge of Jason and multi-agent systems, and then strategy in terms of what we find achievable in a shorter time frame, a minimum viable strategy. The minimum viable strategy is a simple strategy which is enough to complete jobs and make some money, but will not be advanced in terms of cooperation between agents, path finding and auction strategy.

What we want to do is to only use one agent for choosing and completing jobs. This reduces the complexity of the project as we do not need to figure out how to get the agents to cooperate, but makes us a lot less efficient, as the agent needs to do more work in order to complete a job.

We want to reduce the complexity of the jobs chosen, so we only choose jobs that do not require tools in the assembly of items. This makes the jobs easier to complete, as the items required for jobs that do not require tools in their assembly, also only require one or two layers of subitems to assemble. Narrowing down the possible jobs to complete in this way severely reduces the amount of jobs possible to choose, but since we are only using one agent to complete jobs, it is not going to complete that many jobs anyway.

We will completely ignore missions and auctions to reduce the scope of the project. Ignoring auctions will lead to our competitors getting a huge competitive advantage in the competition as they can effectively name their price, but for now it is out of scope. Ignoring missions will mean that we get fined for not completing them, but most missions require tools in the assembly of the required items, so we will just have to eat the cost.

For the minimum implementation, we are going to use one vehicle, the truck, as it has the most space and battery power. The biggest factor is space, since we are only using one agent. It is necessary to assemble and carry all the items required for the mission before the agent delivers them. This serves two purposes, it means that the agent does not spend time going to deliver multiple times for one job and it means that the agent still carries the items if the job ends before it can deliver them. Carrying the items when the job ends means that they can be used for a later job, which saves time on the later job. This only works because we only do jobs that require items which do not require tools to assemble, as it severely restricts the amount of items the agent might buy for any given job.

One easy addition that could make a difference, is an algorithm for finding the closest charging station, since some agents would have very short time on the street, if they would have to recharge in a very remote charging station afterwards. In order to make the implementation simple we would calculate the distances for the vehicles in Manhattan distance and not actual distance of the route, and for drones just the Euclidean distance, which is optimal because they are airborne.

4.2 Advanced Strategy

If we pretend that we could implement anything and had to think of a strategy that could have a good chance of winning the Multi-Agent Programming Contest, then we would have to lay out first a solid base strategy and then also think about small ways to gain an edge over our opponents. Here are a few

points that could show improvements right away.

Path Finding and Measurement of Distances: As the maps are actual street maps from the Earth, using an actual navigation system is possible. This could give more accurate assessments of distances, and help improve path finding and routing in general. Implementing an algorithm to find out which destinations to go to in succession would reduce time spent on travel. This is known as the Traveling Salesman Problem. Unfortunately this is an NP-hard problem, and computation time increases drastically for each destination node. It is feasible to use a traveling salesman algorithm up to around 14 nodes, so for more than that we would have to split facilities into groups based on their location.

Cooperation: We want to split jobs up so that each agent completes different parts of jobs. What matters here is completing jobs quickly, so that the other team does not complete the jobs our agents are working to complete. We want to split the jobs up so that each agent can finish its own part as quickly as possible. The trucks take a long time to travel around, but can carry a heavy load, while drones and bikes are very quick but can carry less. It should be noted that drones are only able to carry the smaller items, and they are not even able to carry all of the tools they can use. This means that drones and also the bikes should be used mostly to help the other agents assemble items, or to buy parts for them. The car and truck could just do their thing only going to the most important way points and then order the bikes and drones to fetch things for them, or to assist in assembly.

Auction Betting: Optimizing auction betting to make our opponents lives as difficult as possible, while maximizing our own benefit would both make us more money and make the opponent less money, as they are going to either have to take jobs at a worse price or get less jobs to complete.

Buying Strategically: Buying items that the opponent team needs, so that they go out of stock, could be a way to freeze the opponent team and secure important items for ourselves. Buying more than we need of certain items can be an advantage, if we know which items are likely to be useful in the future. The drones can transfer items between other agents that need them in order to complete a job.

Selecting Jobs Strategically: It is possible for one team to post a job that the other team can take. As we discussed in section 2.1.3 agents can be deceptive. Thus given a smart opponent, it would be smart to check whether the job was posted by the opposing team. If it is, it might suggest that the job is somehow not a good one to take, since the job could either help the opponent in some way, or be difficult or unprofitable to carry out.

CHAPTER 5

The Jason Programming Language

In this chapter we discuss the Jason programming language and how to use it. This chapter is largely based on the Jason book 'Programming Multi-Agent Systems in AgentSpeak Using Jason' [9]. For further information about the Jason language we refer to the aforementioned book.

5.1 Eclipse AgentSpeak Plugin

In order to use Jason we need an IDE. We have chosen to use the Jason plugin for Eclipse, As seen figure 5.1. The plugin enables us use our preferred IDE Eclipse for developing, running, and debugging our code [10].

With the debugging tool called Mind Inspector, which is shown in figure 5.2, we are able to see the inner workings of our agents. We can see what it currently believes, what events its responding to and so on. We can also run the agent through steps in our simulation.

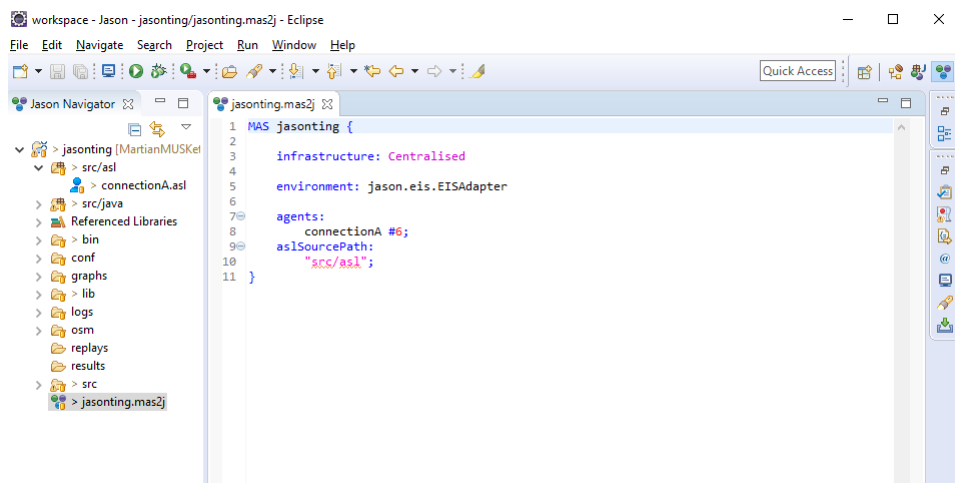


Figure 5.1: Eclipse with Jason plugin, in Jason perspective

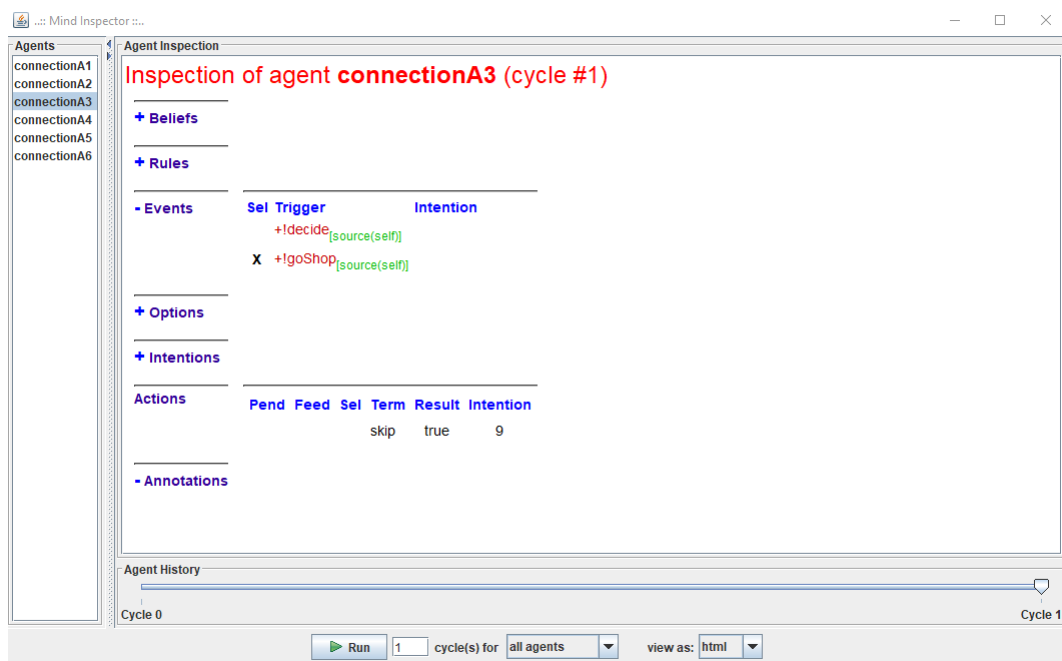


Figure 5.2: Jason Mind Inspector, used for debugging Jason agents

```
1 MAS jasoning {
2
3   infrastructure: Centralised
4
5   environment: jason.eis.EISAdapter
6
7   agents:
8     connectionA #6;
9   aslSourcePath:
10     "src/asl";
11 }
```

Figure 5.3: .mas2j project file

5.2 Project File

At the core of a Jason project is the .mas2j file. It specifies infrastructure, agents, environment and optionally GUI.

The environment specified in the project file is the implementation of an Environment Interface Standard (EIS), through which all communication between agents and environment or server in our project, takes place. Our .mas2j file can be seen in figure 5.3.

5.3 Agent .asl Files

While the .mas2j file is the core of the project, the programming of the agents happens in the agent .asl files. Here we define initial plans, rules, and beliefs. There can be many .asl files in one project defining different agents and .asl files can also be imported into other .asl files using the 'include' keyword.

5.4 Concepts of a Jason Agent

A Jason agent has a lot of stuff on its mind, such as beliefs, rules, plans, and goals. It can also perform actions. In this section we will explain some of the core concepts of a Jason agent.

5.4.1 Actions

Actions can generally be explained as something done by an agent. On the other hand agents are described as *"active, purposeful producers of actions"*[9].

There are two kinds of actions; external actions, which are just called *actions* and *internal actions*. Actions are made by an agent and they influence the environment they are in, in some way. In terms of the Multi-Agent Programming Contest, this could be a *buy* action. It transfers money from the agent and an item from the shop to the agent, thus the effect of the action can be measured. Internal actions on the other hand, do not directly affect the environment they are in. Instead they are more of an intellectual nature. They are used by agents to find, sort, measure, calculate and so on. There is a number of standard library internal actions available in Jason and these can be recognized because they start with a punctuation mark. The developer can also implement their own internal action by creating a Java class, which extends the *DefaultInternalAction*[7] from the Jason library. Internal actions are written in Java and may facilitate the user with helpful methods of processing data, which could be hard to do in Jason. When a user-implemented internal action is called, the package name of the internal action is written before the name of the action. An internal action can take any number of arguments, which are then found in an array of *Terms* which is a parameter of the execute method of the action.

5.4.2 Beliefs and Rules

If say the agent believes that Oslo is a city in Sweden, it might have a belief that says `City(Oslo, Sweden)`. Beliefs can be defined in the agent file, but can also be given to the agent by percepts, logical reasoning within the agent, or communication with other agents. Rules are logical shorthands. If an agent wants to attend a concert, it might have the following beliefs:

```
HasMoney(30) TicketPrice(15)
```

It likely also has the following rule, which returns true if the agent believes that it can afford the ticket:

```
CanAffordTicket(Price, Money):-Price<Money.
```

5.4.3 Plans and Events

In order to be reactive, autonomous, and proactive, as described in chapter 2, an agent needs to be able to have plans. An agent's plan uses the following format:

```
event:condition<-body.
```

- AgentSpeak triggering events:
 - **+b** (belief addition)
 - **-b** (belief deletion)
 - **+!g** (achievement-goal addition)
 - **-!g** (achievement-goal deletion)
 - **+?g** (test-goal addition)
 - **-?g** (test-goal deletion)

Figure 5.4: Full list of events, figure courtesy of [6]

A full list of possible events is shown in figure 5.4.

The beliefs are covered in section 5.4.2. Achievement goals are states an agent wants to achieve and are denoted with a '!'.

One example of an achievement goal is `!enter(car)`, where an agent wants to enter a car. A test goal is denoted by a '?' and is usually only used to check the belief base if a value is there. A Condition is a logical expression, usually a check on what the agent currently believes. Body is a list of actions, subgoals, for loops, and if statements.

5.4.4 The Reasoning Cycle

A Jason agent runs by means of a reasoning cycle. This cycle is the decision making and update cycle of the agents mind. It can be broken down into 10 steps. Steps 1-4 deal with updating the agents beliefs about the environment and about other agents. Steps 5-10 deal with executing and updating an intention. These are the 10 steps:

1. **Perceiving the Environment:** The agent senses its environment and updates its beliefs about the state of the environment.
2. **Updating the Belief Base:** The agent updates its belief base based on the new perception about the environment. This means it discards old beliefs that were not perceived in this cycle and adds new beliefs that were not present in the belief base prior to this cycle. This is done using the

Belief Update Function and the Belief Revision Function. These can be customized in order to better handle updates of beliefs.

3. **Receiving Communication from Other Agents:** During this step the interpreter checks for messages, that other agents may have sent to the agent. This is done by the checkMail method, which can be customized. This method makes the received messages available at the level of the AgentSpeak interpreter. Only one message is processed by the AgentSpeak interpreter per cycle. Unless the selection function is overridden by the developer, Jason will just choose the first message received.
4. **Selecting 'Socially Acceptable' Messages:** The method 'SocAcc', which typically needs to be customized, determines what messages can be accepted. Because agents can send know-how and delegate goals, it is important that the source is a trusted one by the agent.
5. **Selecting an Event:** Events such as addition or removal of beliefs or new goals, can queue up if many events have happened in the span of too few reasoning cycles. In this case a selection function, which can be customized selects the event to be dealt with. The standard implementation will take the first event in the queue. If the set of events is empty the reasoning cycle will proceed directly to step 9.
6. **Retrieving all Relevant plans:** The next step is to retrieve all plans which are relevant to this event. It will look at the arguments and the source or the event to see if they match at and retrieve all the relevant plans.
7. **Determining the Applicable Plans:** The agent now determines all the applicable plans for the event that have a chance of succeeding. This is done on the basis of the know-how of the agent and its current beliefs.
8. **Selecting One Applicable Plan:** The standard implementation of the selection function for the applicable plan called an 'option selection function'. The standard implementation will select the first application plan in the plan library, which in turn is determined by the order of the plans in the source code. So the first applicable plan in the source code will be executed given the standard selection function. This results in an addition to an existing intention for the agent. Added to the intention is that the agent wants to achieve the current goal by means of the selected plan. This is called the intended means for the goal.
9. **Selecting an Intention for Further Execution:** Selecting the right intention can be very important and thus it is advisable to customize the selection function for selecting intentions to act on, in order to do the most important things first. The standard selection function for intention

will use 'round-robin' scheduling for this and carry out one action on each intention on the intention list and then start from the top again and go down the list again. This makes it so each intention gets the same amount of attention. Technically this is implemented by removing the first item on the list and adding it to the end of the list each cycle.

10. **Executing One Step of an Intention:** The agent now performs one step of an intention. Intentions can be suspended if they are waiting for feedback on an execution of an action or they are waiting for message replies from other agents. Before the next reasoning cycle starts, the agent checks for any such feedback and the relevant intentions are updated according to the feedback. The intentions that received their needed feedback are put back in the active list of intentions.

Setting Up the Project

In this chapter we will explain how to set up the project for use and development. The guides are made for the Eclipse IDE which the user will need to have installed as well as a Java Development Kit (JDK) of version 1.7 or newer. The Eclipse version needs to be of version 3.7.0 or newer in order to install the Jason Plugin.

6.1 Setting Up the Project from Scratch

This is a guide to setting up the environment from scratch.

1. Download Jason 2.1 from the Jason web site [8].
2. Install the Jason plugin for eclipse following the guide found on the Jason website [10].
3. Download the compiled version of the Multi-Agent Programming Contest software called "massim-2017-1.2-bin.tar.gz" at the time of writing, found at their official GitHub page [5].
4. Create your own Jason project.

5. Add the following three .jar files to the lib folder of the project: Jason 2.1 jar from your Jason installation, the EISMASSim, and server-2017 jars from the compiled Multi-Agent Programming Contest project.
6. Now add these as local jars to the build path of your project.

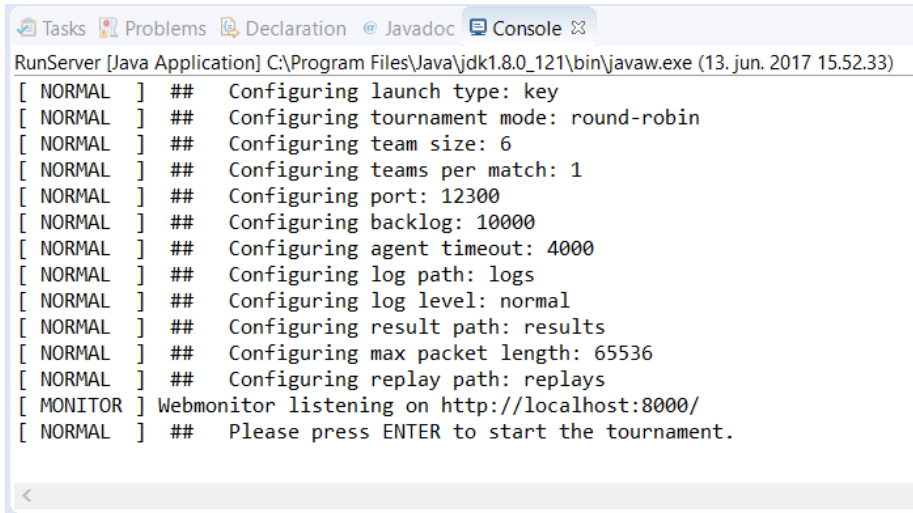
The project is now ready for testing and development. In order to make it easier to run the server, it is a good idea to make a class in your project that takes care of this. See A.4 for an example of how to do this.

6.2 Guide: Importing and Running the Project

Now we will explain how to swiftly get the project up and running.

1. Install the Jason plugin for eclipse following the guide found on the Jason website [10].
2. Add the project to Eclipse.
3. Now run the RunServer.java in the src/java folder as a java application. Now your console should look like in figure 6.1.
4. Start the Jason application by selecting the Jason perspective and hitting the top left run button as seen in figure 6.2.

When the multi-agent system console has popped up, select the Eclipse console and press enter. Now the simulation has started. To watch the agents in action open a web browser and access localhost:8000. Here you will find an interactive graphical representation of the simulation.



```
RunServer [Java Application] C:\Program Files\Java\jdk1.8.0_121\bin\javaw.exe (13. jun. 2017 15.52.33)
[ NORMAL ] ## Configuring launch type: key
[ NORMAL ] ## Configuring tournament mode: round-robin
[ NORMAL ] ## Configuring team size: 6
[ NORMAL ] ## Configuring teams per match: 1
[ NORMAL ] ## Configuring port: 12300
[ NORMAL ] ## Configuring backlog: 10000
[ NORMAL ] ## Configuring agent timeout: 4000
[ NORMAL ] ## Configuring log path: logs
[ NORMAL ] ## Configuring log level: normal
[ NORMAL ] ## Configuring result path: results
[ NORMAL ] ## Configuring max packet length: 65536
[ NORMAL ] ## Configuring replay path: replays
[ MONITOR ] Webmonitor listening on http://localhost:8000/
[ NORMAL ] ## Please press ENTER to start the tournament.
```

Figure 6.1: The Eclipse console after starting the server

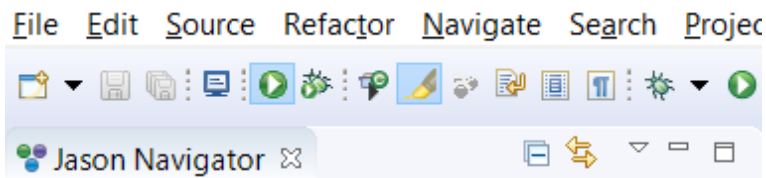


Figure 6.2: The Eclipse console after starting the server

Implementation

In this chapter we discuss how we have implemented our multi-agent system. We have implemented a minimum viable product and an improved version of said minimum viable product. We will go into detail on them both.

7.1 Implemented Minimum Viable Product

In this section we discuss our initial implementation of a multi-agent system, our minimum viable product.

7.1.1 Cooperation

In our first implementation we only let our truck take actions. This is because we have, in this minimum viable strategy, not implemented a way for the agents to divide work between them, so all the agents try to complete the same jobs in the same order. Our truck is the agent most suited for completing jobs on its own and so we have disabled the other agents.

7.1.2 Find Job

When the agent is not currently doing a job, it looks for jobs for it to complete. We have implemented a selector, which chooses a job that requires items the agent is able to construct on its own. The logic checks if any of the items, the job requires, needs tools that the agent does not carry in any way. If it does, we reject the job and find another one. Since we are currently only using the truck, we do not need to check if the agent is able to carry any more items. It needs to be noted that for the purposes of our project build, construct, and assemble are the same things.

7.1.3 Core Loop

Once a job has been selected, we enter the core loop for the agent, illustrated in figure 7.1. The agent checks if it needs to recharge, if it does, it starts a charging plan.

If it cannot, it checks if it can complete the current job, if it does, it starts a delivery job plan.

If it cannot, it checks if it can construct one of the items needed for the job, if it can, it starts a construct item plan.

If it cannot it checks if it can construct a part of a required item, if it can it starts a construct part plan.

If it cannot, it checks if it can buy subparts for constructing a part for a required item, if it can, it starts a plan for that.

If it cannot, it checks if it can buy a part for construction of a required item. If it can, it starts a buys plan.

If it cannot, the job is likely expired, and we check for that before running the core loop again.

The reason for the core loop running the way it does is that it saves space in the truck and the agent does not deliver items if it cannot complete a job. It saves space, since the sum of the space used by the parts is greater than the assembled items. This means that we have a greater chance of having space for the items the agent is constructing.

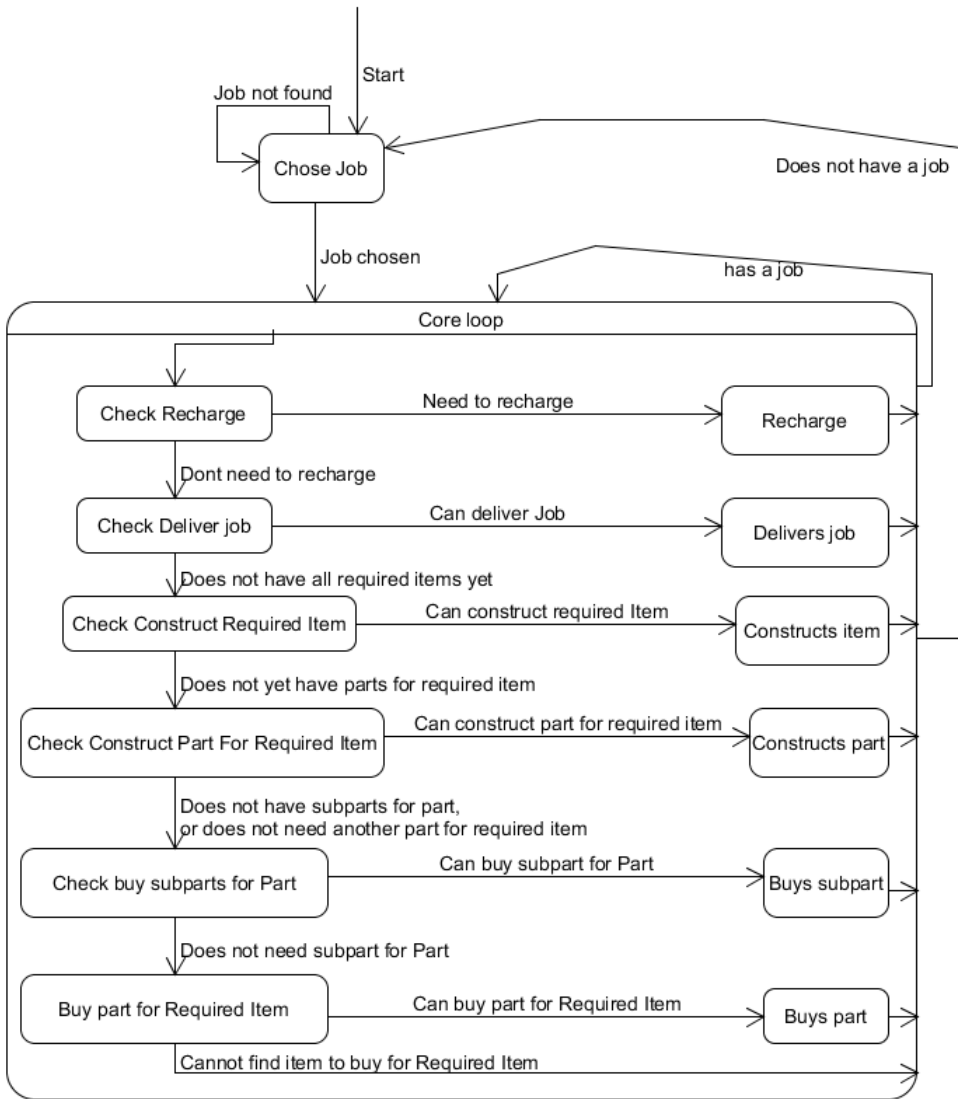


Figure 7.1: Core loop of our agents

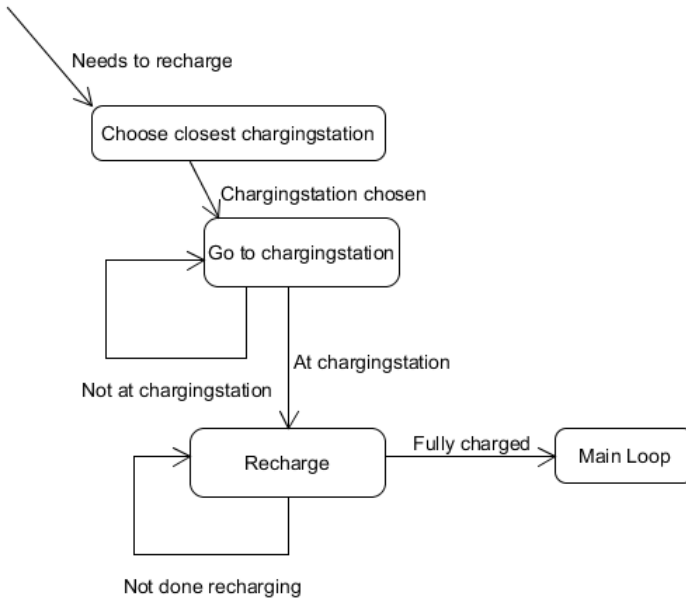


Figure 7.2: Agent recharge logic

7.1.4 Recharge

When an agent checks recharge and finds that its battery level reaches a value based upon its current battery level vs. its maxbattery ($\text{charge} = \langle (150 + 0, 15 * \text{maxbattery}) \rangle$), it checks which charging station is closest, goes to that one and recharges until its battery level is full, then it restarts the core loop, this is illustrated in figure 7.2

Should an agent, in spite of our above check on battery, run completely out of power, it will use the recharge command, which charges the agent on the spot using solar panels. This is quite inefficient, so when the agent reaches the level where it would normally start to look for a charging station it does so, and goes there in order to recharge faster.

7.1.4.1 Deliver

When an agent checks deliver job, it checks whether or not it carries all the items required to finish the active job. If it does, it goes to the facility to deliver the items and delivers them.

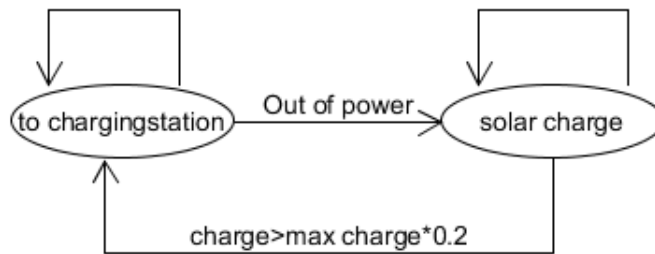


Figure 7.3: Solar charging is used if the battery of the agent is completely depleted

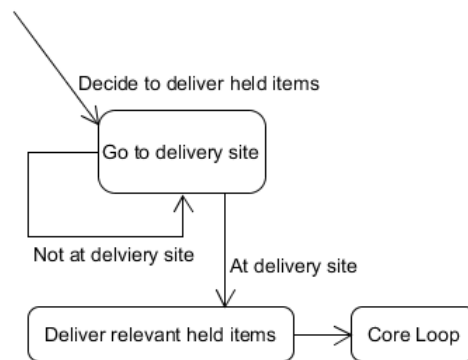


Figure 7.4: Agent delivery logic

This is illustrated in figure 7.4.

7.1.4.2 Construct Item

When an agent checks if it can construct a required item, it checks if it has all the parts needed for any required item it does not currently have enough of to complete the job. If it does, it goes to a workshop and constructs the item. This is illustrated in figure 7.5.

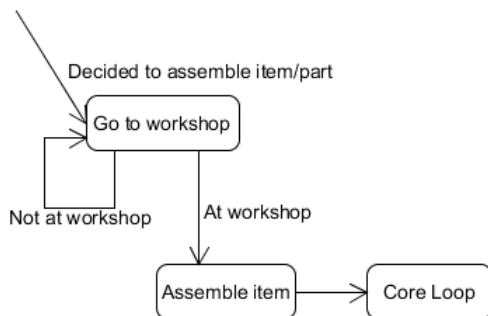


Figure 7.5: Agent assembly logic

7.1.4.3 Construct Part

When an agent checks if it can construct a part for a required item, it checks if there is a required item not yet made that there is a part that requires assembly that it has enough subparts to assemble. If it does, it goes to a workshop and assembles the part. This is illustrated in figure 7.5, and shares logic with Construct item.

7.1.4.4 Buy Subpart

When an agent checks buy subparts for parts, it checks if there is a part we need more of that requires subparts we also need more of. If it does, it goes to buy the subparts. This is illustrated in figure 7.5. For the purposes of buying, parts, items, and subparts are the same, though you cannot buy the items required for missions straight out of a shop.

7.1.4.5 Buy Part

When an agent checks buy parts for required item, it checks if there is a required item we need more of which requires a part we do not have and can buy. If it does, it goes and buys the part. This is illustrated in figure 7.6. Buy part and buy subpart use same logic.

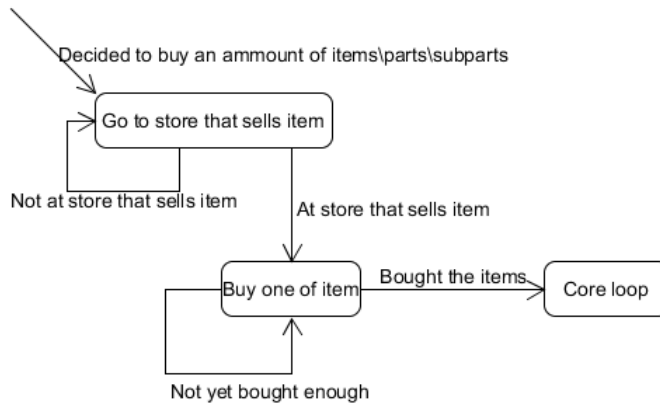


Figure 7.6: Agent buy logic

7.2 Improved Version

This version of our multi-agent system is improved in two ways. There is a dedicated agent which checks what jobs are viable, splits the job into parts and delegates tasks to other agents. We have also made some improvements on the core loop, so that it runs more effectively.

7.2.1 Improved Cooperation

In the improved version of our multi-agent system, we have implemented functionality that lets agents cooperate in the completion of jobs, where each agent constructs and delivers a set of items on its own. We use all of the agents except for the drones, as the drones are not able to carry all of the parts required to construct any of the items that are required to finish jobs.

7.2.2 Improved Choose Job

The new choose job logic lets one drone check what jobs are viable. It checks that none of the items required for the job uses tools and that it at most requires four different kinds of items. It then checks how much space each set of items take. It then tells the other agents to assemble and deliver the items, specifying a set of items each, and making sure that each agent gets a set they have space

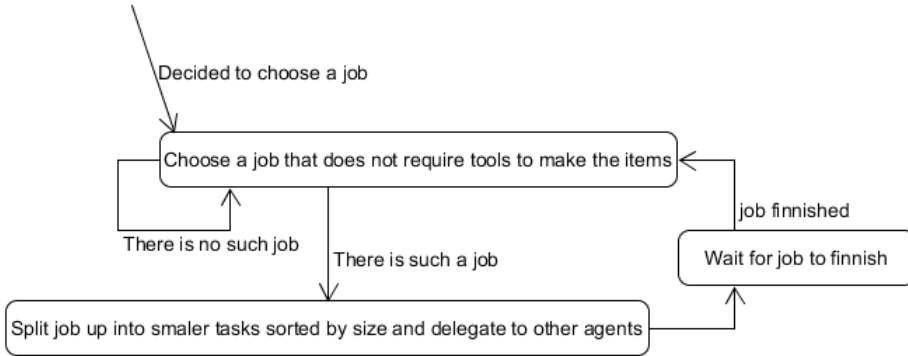


Figure 7.7: Improved Choose Job logic

for. It then waits until the job is either completed or failed, and starts choosing a new job for the other agents to complete. If there is no job for the other agents to complete, that is, there is no jobs that at most require four different items, where none of the items require tools, it simply waits until there is one. This is illustrated in figure 7.7

7.2.3 Improved Core Loop

This is now the only loop most agents use, they no longer check for jobs themselves, instead they wait until they have been told to construct an item for a job.

The agent checks if it needs to recharge, if it can, it starts a charging plan.

If it cannot not, it checks if it can deliver the item(s) its been asked to deliver, if it can, it starts a delivery job plan.

If it cannot not, it checks if it can construct all of the items it has been ordered to construct for the job, if it can, it starts a construct item plan.

If it cannot not it checks if it can construct all the parts, of a single type of part, of the items it has been ordered to construct for the job, if it can it, starts a construct part plan.

If it cannot not, it checks if it can buy all the subparts it needs for constructing all the parts, for a required item, if it can, it starts a plan for that.

If it cannot not, it checks if it can buy a part for construction of a required item. If it can, it starts a buys plan.

If it cannot not, the job is likely expired, and the agent checks for that. If it is, the agent awaits new orders. other wise the agent runs the core loop again.

This is illustrated in figure 7.8.

7.2.4 Improved Recharge

The logic for recharge is unchanged from the minimum viable product.

7.2.4.1 Improved Deliver

When an agent checks deliver job, it checks whether or not it carries all the items it has been assigned to deliver for the active job. If it does, it goes to the facility to deliver the items, and delivers them. This is illustrated in figure 7.4, as it is not different enough from the regular deliver logic to warrant a new figure.

7.2.4.2 Improved Construct Item

When an agent checks if it can construct a required item, it checks if it has all the parts needed to construct the whole set of items it has been assigned to construct and deliver for the current job. If it does, it goes to a workshop and assembles the item(s). This is illustrated in figure 7.9.

7.2.4.3 Improved Construct Part

When an agent checks if it can construct all parts of a type for the required item(s), it checks if there is a required item not yet made that there is a part that requires assembly that it has enough items to assemble. If it does, it goes to a workshop and assembles the part.

This is illustrated in figure 7.9, and shares logic with Construct item.

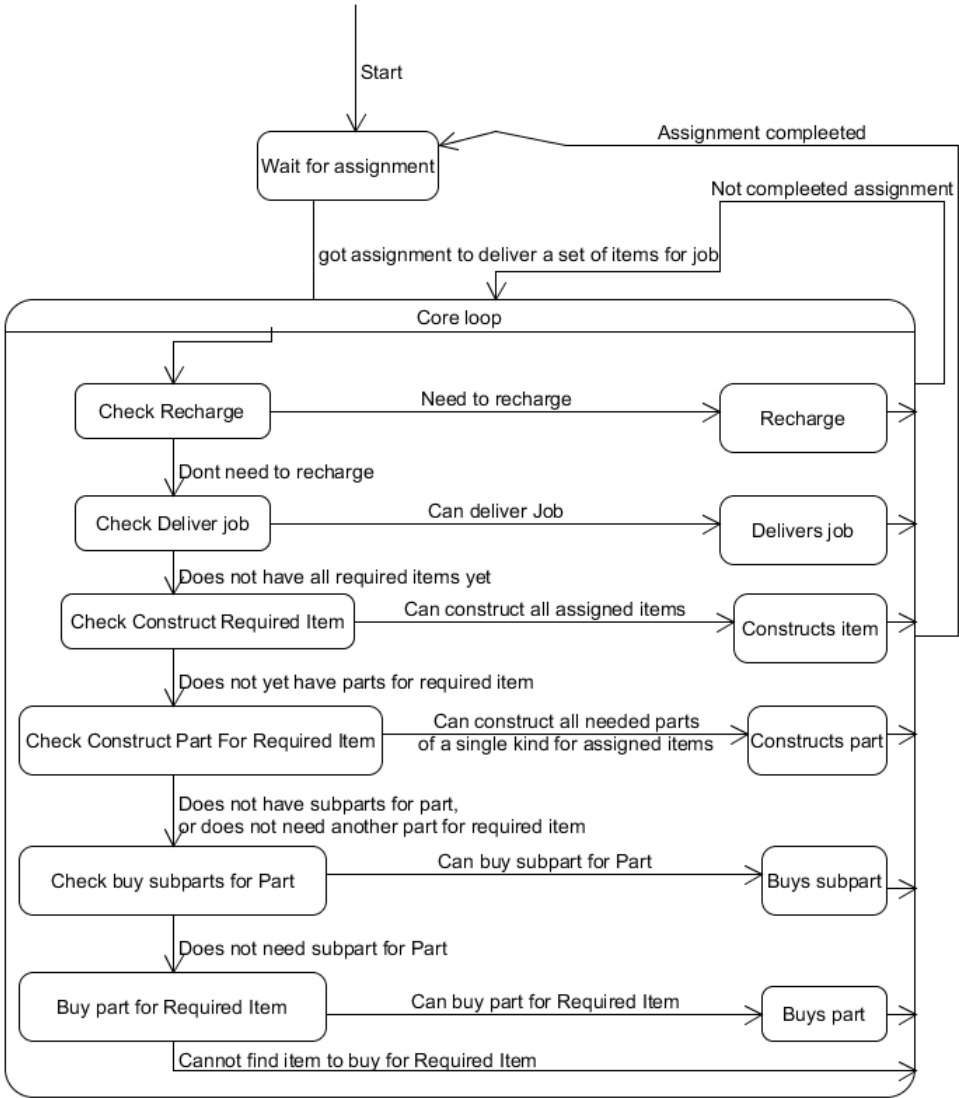


Figure 7.8: Improved Core Loop

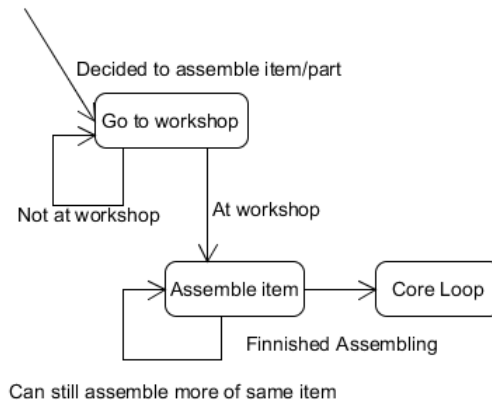


Figure 7.9: Improved Assembly Loop

7.2.4.4 Improved Buy Subpart

When an agent checks buy subparts for parts it checks if there is a part we need more of, which requires subparts we also need more of. If it does, it goes to buy the subparts. The improvement here is that it now buys all the subparts it needs for all the parts, where before it only bought so that it had enough subparts to build a single part of each of the needed parts. This is illustrated in figure 7.6, the flow is not different enough to warrant a new figure.

7.2.4.5 Improved Buy Part

When an agent checks buy parts for required item, it checks if there is a required item we need more of that requires a part we do not have and can buy. If it does, it goes and buys the part. The improvement here is the same as in section 7.2.4.4, it now buys all the subparts it needs for all the parts, where before it only bought so that it had enough subparts to build a single part of each of the needed parts. This is illustrated in figure 7.6, the flow is not different enough to warrant a new figure.

7.3 Internal Action

We have implemented an internal action that calculates the distance between two points on the map. This internal action is used to choose the closest of a given kind of facility. Right now, this is used to select the closest charging station to the agent, so the agent quickly can resume their job, when charge is needed. The measure used is Manhattan distance. This is the same for both implementations.

7.4 Environment Interface

We have elected to save some time on the development of our environment interface and we are using the Jason EISAdapter [3] supplied by the guys behind the contest. This is the same for both implementations.

Discussion and Reflection

In this chapter we discuss our results, possible further improvements upon our program and reflect upon how we have managed this bachelor project.

8.1 Results

We have tested our system by running it on the standard simulation server provided by the Multi-Agent Programming Contest.

In our minimum viable product, the agent complete one or two jobs in the standard 1,000 step simulation, each job having a deadline of approx. 250 steps from start. The seed money given to each team is 50,000 and our agents usually end the simulation with 47,000, most of the money lost is due to fines on uncompleted missions, which we have not implemented a way to handle.

There are of course two reasons for the improvement of the system from the minimum viable product to the improved version. One is that there is actual cooperation between agents, putting the *multi* in multi-agent systems. We now make sure that for every type of item required in a job, an agent assembles and delivers the items of that type. The other reason is that the change to the core

loop means that since the an agent only assembles parts or when it has all the parts or subparts needed, it spends far less time traveling before it assembles and delivers its items.

Since we improved our multi-agent system, we have seen an improvement in how quickly the agents assembles all item of a type by up to 200%. This combined with the fact that every type of item required by a job is assembled by different agents, means that the improved system should out-compete our minimum viable product by a wide margin. There is however one bug which means that the improved version is not yet viable over the old version. Occasionally an agent enters a state, where it completes no actions, even though it says it intends to. We have tried debugging the agent, but we are not yet sure why it enters this state and stays there. This means that the agents cannot fulfill the jobs they are helping to complete and this will cause other jobs, they are assigned to help with, fail in a similar manner.

There are also two other non-critical bugs. One is that agents will buy a couple too many parts for the items they are trying to build and the other is that some agents occasionally fail to start, when we start running the simulation. The first one is likely a logical error, but can be handled by making the agents dump their inventory after they've delivered for a job. We are not sure what causes the second bug, we just need to restart the agents. This can be done in the simulation and needs to be done once per simulation in the first few steps, so it is not that great of a concern.

8.2 Further Work

There are several things we could do to improve our program and make it perform better, apart from fixing the bugs mentioned in the previous section. In the following sections we will discuss some of the changes to the program we believe could improve its performance.

8.2.1 Route Planing

We do not currently use any route planing other than planing the route between the agent and the facility it is currently going to. If we started using route planing for the trips required for entire jobs, we could be able to predict which jobs we would not be able to complete based upon time limit. This would free up resources for other tasks, thereby increasing performance of our program.

8.2.2 Charging Station

In the vein of route planing, we currently use a very rough way of estimating whether an agent needs to recharge its batteries. We simply check whether the agent's battery has less than 150 charge plus 7.5% of the maximum charge. This could be improved greatly if we changed it to be incorporated in the route planing discussed earlier, so that the agent takes as short detours as possible and gets to spend more time completing jobs.

8.2.3 Missions and Auctions

We have not handled Missions or Auction jobs. The jobs themselves are not too different from regular jobs, but we decided we did not have time to handle them properly. The most interesting piece of work left here is estimating the bidding, which ought to be done by checking the prices of every item needed to fulfill the job. It would also be interesting to see how our agents are positioned compared to the opponent's agents and use this information to better adjust the bid.

8.2.4 Cooperation Between Agents

It would be preferable, to our current implementation, if the agents were able to cooperatively assemble items.

8.2.5 Prediction and Counter Play

It would be very neat, if we were able to implement a way to predict how the opponent is trying to complete jobs. This might be used to buy the items the opponent needs in the shops they are trying to buy it in. Game-play wise it would be a great move to disrupt an auction job or a mission in this manner, though we do risk spending more money buying the resources than the opponent loses in fines. As talked about in chapter 4 it would be a good idea, to check whether a job was posted by the server or by the opponent. If it is posted by the opponent there might be a malicious reason behind it, e.g. the jobs have a very bad reward. This could of course also be done by us. If we know that one item is lacking in the stores, or is only found in stores that are inconvenient to go to, we could post a poorly paying job requiring these items, to make the opponent team waste their work force.

8.2.6 Error Handling

We do not currently have much in the way of error handling. The current core loop implementation is actually pretty robust and errors on actions usually mean that the agents will see that the action, which failed earlier, still needs doing, so it will just do it again. This might not hold true as we expand upon the program, so it is something to look out for. We will also need to take blackouts into consideration when they become relevant.

8.3 Reflections

When we started this project we had no experience working with multi-agent systems. It surprised us how long it took for us to set up our programming environments and we frequently got stuck on little things. Bugs, syntax, and a lack of crowd sourced documentation such as StackOverflow all slowed us down.

We did however also get better at checking out source code, syntax documentation and so on. We were also surprised by the quick responses from the people behind the Multi-Agent Programming Contest, when we found errors in the server.

As we have worked with the system, we have come to appreciate working with percepts and the logic involved. We hope to continue working on our project over the summer holiday and participate in the contest come fall.

APPENDIX A

Code

Our source code is split up into a .mas2j file, two .asl files, and internal action code explained in chapter 4. Furthermore the server file made to run the server, as explained in chapter 5.

A.1 Project file (.mas2j)

```
MAS jasanting {  
  
    infrastructure: Centralised  
  
    environment: jason.eis.EISAdapter  
  
    agents:  
        connectionA #6;  
    aslSourcePath:  
        "src/asl";  
}
```

A.2 connectionA (Improved Multi-Agent System.asl)

```

// Agent sample_agent in project jasonting

/* Initial beliefs and rules */
canDoJob(JobID):-job(JobID,_,_,_,RequiredList)&.length(
    ↪ RequiredList,
Length)& Length<5 & not (.member(required(Item,Qty),RequiredList)
    ↪ &
item(Item,_,tools(ToolList),parts(Parts))& not( (.empty(ToolList)
    ↪ ))).
//|.member(Tool,ToolList)&hasItem(Tool,_)
canBuildItem(Item):-item(Item,_,tools(ToolList1),parts(Parts))&
not(.member(required(Item,Qty),RequiredList) & (not .empty(
    ↪ ToolList1))
& not canBuildItem(Item)).//(.member(X,ToolList1)&hasItem(X,_)|
//Not used/finnished because we only build items that dont
    ↪ require tools,
//parts for these items can be bought not built.

needToCharge:-charge(Power)&role(Role,Speed,_,MaxBattery,_)&Power
    ↪ <
(150+MaxBattery/15).

finnishedCharging:-role(Role,Speed,Load,MaxBattery,List)&
charge(Power)&Power==MaxBattery.
atChargingStation(ChargingStationID):- chargingStation(
    ↪ ChargingStationID
,CLAT,CLON,_) & lat(LAT) & lon(LON) & CLAT == LAT & CLON == LON.
atStorage(StorageID):- storage(StorageID,SLAT,SLON,_,_,_) & lat(
    ↪ LAT) &
lon(LON) & SLAT == LAT & SLON == LON.
atShop(ShopID):- shop(ShopID,SLAT,SLON,_,_) & lat(LAT) & lon(LON)
    ↪ & SLAT
== LAT & SLON == LON.
atWorkShop(WorkshopID):- workshop(WorkshopID,SLAT,SLON) & lat(LAT
    ↪ ) &
lon(LON) & SLAT == LAT & SLON == LON.

hasAllItems(JobID):-job(JobID,_,_,_,RequiredList)& not

```

```

( .member(required(Item,Qty),RequiredList) & not hasItem(Item,Qty
  ↪ ) ).
//readyToBuildItem(JobID,Item):-job(JobID,_,_,_,RequiredList) &
.member(required(Item,Qty),RequiredList) & ((not hasItem(Item,_))
  ↪ |(hasItem(Item,Qty1)&Qty>Qty1)) & item(Item,_,_,parts(
  ↪ Parts))&not(.member([Part,Qty2],Parts)&(not(hasItem(Part,_
  ↪ ))|hasItem(Part,Qty3)&Qty2>Qty3)).
readyToBuildItem(Item,Qty):- item(Item,_,_,parts(Parts))& ((not
  ↪ hasItem(Item,_))| (hasItem(Item,Qty1)&Qty>Qty1)) &(not(.
  ↪ member([Part,Qty2],Parts)&((not(hasItem(Part,_))|hasItem(
  ↪ Part,Qty3)&Qty2>Qty3)))| (hasItem(Item,Qty1)&Qty>Qty1)).
//readyToBuildSubItem(JobID,Part):- job(JobID,_,_,_,
  ↪ RequiredList) & .member(required(Item,Qty),RequiredList) &
  ↪ ((not hasItem(Item,_))| (hasItem(Item,Qty1)&Qty>Qty1)) &
  ↪ item(Item,_,_,parts(Parts))&.member([Part,Qty2],Parts)&(
  ↪ not(hasItem(Part,_))|hasItem(Part,Qty3)&Qty2>Qty3)&item(
  ↪ Part,_,_,parts(Parts2))&not(.empty(Parts2))&not(.member([
  ↪ Part2,Qty4],Parts2)&(not(hasItem(Part2,_))|hasItem(Part2,
  ↪ Qty5)&Qty4>Qty5)).

readyToBuildSubItem(Item,Part, Number,Number*Qty2):- item(Item,_,
  ↪ _,parts(Parts)) & .member([Part,Qty2],Parts) & (not(
  ↪ hasItem(Part,_))|hasItem(Part,Qty3)&((Number*Qty2)>Qty3))
  ↪ & item(Part,_,_,parts(Parts2)) & not(.empty(Parts2)) & (
  ↪ not(.member([Part2,Qty4],Parts2)&(not(hasItem(Part2,_))|
  ↪ hasItem(Part2,Qty5)&((Number*Qty2*Qty4)>Qty5))))).

shopForItem(ItemID,Part,ShopID,QtyNeeded,Qty2):- item(ItemID,_,_,
  ↪ parts(Parts))&( .member([Part,Qty2],Parts)&(not(hasItem(
  ↪ Part,_))|hasItem(Part,Qty3)&((Qty2*QtyNeeded)>Qty3)))&shop
  ↪ (ShopID,_,_,Inventory)& .member(item(Part,_,_),Inventory
  ↪ ).

shopForSubItem(ItemID,Part2,ShopID,QtyNeeded,Qty4):- not(hasItem(
  ↪ ItemID,QtyNeeded))&item(ItemID,_,_,parts(Parts))&( .member
  ↪ ([Part,Qty2],Parts)&(not(hasItem(Part,_))|hasItem(Part,
  ↪ Qty3)&((Qty2*QtyNeeded)>Qty3))) & item(Part,_,_,parts(
  ↪ Parts2))&not(.empty(Parts2))&( .member([Part2,Qty4],Parts2)
  ↪ &(not(hasItem(Part2,_))|hasItem(Part2,Qty5)&((Qty4*Qty2*
  ↪ QtyNeeded)>Qty5)))&shop(ShopID,_,_,Inventory)& .member(
  ↪ item(Part2,_,_),Inventory).

/* Initial goals */
!init.

/* Plans */

```

```

+!init:true<-
  if(false&role(truck,Speed,Load,MaxBattery,ToolList)&.
    ↪ member(MissingTool,ToolList)& (not (hasItem(
    ↪ MissingTool,_)))& shop(ShopID, _, _, _, Inventory)
    ↪ & .member(item(MissingTool, _, _),Inventory)){
    .term2string(MissingTool,X);
    !!goBuy(ShopID,X,1,true);
    goto(ShopID);
  }
  else{
    if(charge(_)[entity(connectionA4),source(percept)])
      ↪ {
      +itemList([]);
      +weightList([[item10,176],[item12,135],[
      ↪ item14,195],[item17,460]]);
      if(weightList(List)){
        .print(List);
      }
      !!choseJob;

      //go to plan for chosing jobs for agents
      skip;
    }
    else{
      if(not(role(drone,Speed,Load,MaxBattery,
      ↪ ToolList))){
        if(hasItem(_,_)){
          //!!goDump;
          //goto(dump0);
          !!decide;
          skip;
        }
        else{
          !!decide;
          skip;
        }
      }
      else{
        !!skip;
        skip;
      }
    }
  }
}

```



```

+!skip:true<-
!!skip;
skip;
.

+!choseJob: not(activeJob(_))<-
  if(canDoJob(JobID)&job(JobID,Start,End,Rewawrd,Storage,
    ↪ RequiredList)){
    .print("JobID ",JobID," Start ",Start," End ",End,"
      ↪ Rewawrd ",Rewawrd," Storage ",Storage,"
      ↪ RequiredList ",RequiredList);
    +activeJob(JobID);
    if(weightList(WeightList)){

      for(.member(required(Item, Qty),RequiredList)&.
        ↪ member([Item,Weight],WeightList)){
        if(item(Item,_,_,_)&itemList(List)){
          -+itemList([required(Weight*Qty,Item
            ↪ ,Qty)|List]);
        }
      }
    }
    if(itemList(List)){
      .sort(List,SortedList);
      .print(SortedList);
      -+itemList(SortedList);
    }
    if(itemList(SortedList)&.length(SortedList,Length)
      ↪ &.nth(Length-1,SortedList,required(_,ItemID1
      ↪ ,Qty1)) ){
      .send(connectionA6,tell,dojob(JobID, ItemID1
        ↪ ,Qty1));
      if(Length>1 & .nth(Length-2,SortedList,
        ↪ required(_,ItemID2,Qty2)) ){
        .send(connectionA1,tell,dojob(JobID,
          ↪ ItemID2,Qty2));
        if(Length>2 & .nth(Length-3,
          ↪ SortedList,required(_,ItemID3,
          ↪ Qty3)) ){
          .send(connectionA2,tell,dojob
            ↪ (JobID, ItemID3,Qty3)
            ↪ ;
          if(Length>3 & .nth(Length-4,
            ↪ SortedList,required(_,

```



```

        !!decide;
        skip;
        ///!!goDump;
        //goto(dump0);
    }
    else{
        !!decide;
        skip;
    }
    .

+charge(0):true<-
    .drop_all_intentions;
    .drop_all_desires;
    .drop_all_events;
    !!solarcharge;
    recharge
    .

+!solarcharge: needToCharge<-
    !!solarcharge;
    recharge
    .

+!solarcharge: not needToCharge<-
    !!chooseChargingStation;
    recharge
    .

-job(JobID,_,_,_,_):dojob(JobID,Item,Qty)<-
    -dojob(JobID,Item,Qty)
    .print("removed Dojob");
    skip;
    .

-job(JobID,_,_,_,_):activeJob(JobID)<-
    -activeJob(JobID);
    .print("removed Job id");
    skip;
    .

+!chooseChargingStation: lat(AgentLat) & lon(AgentLon) <- -+
    ↪ minDistBel("", 1000);
    for(chargingStation(Name,Lat,Lon,_)){

```

```

        actions.distance(Lat, Lon, AgentLat, AgentLon,
            ↪ Distance);
        ?minDistBel(N, D);
        if(Distance < D){
            -+minDistBel(Name, Distance);
        }
    }
    ?minDistBel(N, D);
    !!recharge(N);
    if(chargingStation(N,LatC,LonC,_)){
        goto(LatC,LonC);
    }
    else{
        goto(N);
    }
    .

+!recharge(ChargingStation) : needToCharge & not
    ↪ atChargingStation(ChargingStation) <-
    !!recharge(ChargingStation);
    if(chargingStation(ChargingStation,LatC,LonC,_)){
        goto(LatC,LonC);
    }
    else{
        goto(ChargingStation);
    }
    .

+!recharge(ChargingStation) : not needToCharge & (not
    ↪ atChargingStation(ChargingStation) | finishedCharging)<-
    !!decide
    skip.

+!recharge(ChargingStation) : atChargingStation(ChargingStation)
    ↪ & not finishedCharging <-
    !!recharge(ChargingStation);
    charge.

+!goBuild(Item,Qty): not facility(workshop0)<-
    !!goBuild(Item,Qty);
    goto(48.832,2.35192);
    .
+!goBuild(Item,Qty): facility(workshop0)<-

```

```

    if(lastAction(assemble) & not lastActionResult(successful)
        ↪ ){
        !!goBuild(Item,Qty);
        assemble(Item);
    }
    else{
        if(Qty = 0){
            !!decide;
            skip;
        }
        else{
            !!goBuild(Item,Qty-1);
            assemble(Item);
        }
    }
}
.

+!standAndDeliver(JobID):true<-
    if(job(JobID,StorageID,_,_,_,RequiredList)){ // & .member(
        ↪ required(NameRequired,QtyRequired),RequiredList) &
        ↪ hasItem(Name, Qty)& Name==NameRequired){
        !!deliver(StorageID, JobID);
        if(storage(StorageID,LatC,LonC,_,_,_)){
            goto(LatC,LonC);
        }
        else{
            goto(StorageID);
        }
    }
    else{
        !!decide;
        skip;
    }
}
.

+!deliver(StorageID, JobID): not atStorage(StorageID)<-
    !!deliver(StorageID, JobID);
    if(storage(StorageID,LatC,LonC,_,_,_)){
        goto(LatC,LonC);
    }
    else{
        goto(StorageID);
    }
}

```

```

.
+!deliver(StorageID, JobID): atStorage(StorageID)<-
    deliver_job(JobID);
    if(lastAction(randomFail)){
        !!deliver(StorageID, JobID);
        skip;
    }
    else{
        !!decide;
        if(dojob(JobID, ItemID, QtyNeeded)[source(
            ↪ connectionA4)]){
            -dojob(JobID, ItemID, QtyNeeded)[source(
                ↪ connectionA4)];
            .print("removing dojob, by way of dellivery");
        }
        !!decide;
        skip;
    }
.

+!goShop: true <-
    if(job(_,_,_,_,_,RequiredList) & shop(ShopID, LatC, LonC,
        ↪ _, Inventory) & .member(required(NameRequired,
        ↪ QtyRequired),RequiredList) & .member(item(Name1,
        ↪ Price1, Qty1),Inventory) & Name1 == NameRequired){
        .print("We decided to buy something");
        if(QtyRequired > Qty1){
            !!goBuy(ShopID, NameRequired, Qty1,false);
            goto(LatC,LonC);
        }
        else{
            !!goBuy(ShopID, NameRequired, QtyRequired,
                ↪ false);
            goto(LatC,LonC);
        }
    }
    else{
        !!decide;
        skip;
    }
.

+!goBuy(ShopID, NameRequired, Qty1, Init): not facility(ShopID)<-

```

```

    !!goBuy(ShopID, NameRequired, Qty1, Init);
    if(shop(ShopID, LatC, LonC, _, Inventory)){
        goto(LatC,LonC);
    }
    else{
        goto(ShopID)
    }
    .

+!goBuy(ShopID, NameRequired, Qty1, Init): facility(ShopID)<-
    .term2string(Qty1,X);
    buy(NameRequired,X);
    if(lastActionResult(Result)&Result == successful){
        if(Init){
            !!init;
            skip;
        }
        else
        {
            if(Qty1=1){
                !!decide;
                skip;
            }
            else{
                !!goBuy(ShopID, NameRequired,
                    ↪ Qty1-1, Init);
                skip;
            }
        }
    }
    else{
        !!goBuy(ShopID, NameRequired, Qty1, Init);
        skip;
    }
    .

+!closest: true<-
    if(not(First == true)){
        FirstRun = true;
    }
    .

+!choseJob: true<-
    skip;

```



```

.
+!goDump: not facility(dump0)<-
    !!goDump;
    goto(dump0);
.

+!goDump:facility(dump0)<-
    if(hasItem(Item,Qty)){
        .term2string(Qty,X);
        !!goDump;
        dump(Item,X);
    }
    else{
        !!decide;
        skip;
    }
.

```

A.3 connectionB (Minimum Viable Multi-Agent System

```

// Agent sample_agent in project jasonting

/* Initial beliefs and rules */

canDoJob(JobID):-job(JobID,_,_,_,RequiredList)& not (.member(
    ↪ required(Item,Qty),RequiredList)&item(Item,_,tools(
    ↪ ToolList),parts(Parts))& not( (.empty(ToolList))))).
canBuildItem(Item):-item(Item,_,tools(ToolList1),parts(Parts))&
    ↪ not(.member(required(Item,Qty),RequiredList) & (not .empty
    ↪ (ToolList1)) & not canBuildItem(Item)).

needToCharge:-charge(Power)&role(Role,Speed,_,MaxBattery,_)&Power
    ↪ <(100+MaxBattery/20).

finnishedCharging:-role(Role,Speed,Load,MaxBattery,List)&charge(
    ↪ Power)&Power==MaxBattery.
atChargingStation(ChargingStationID):- chargingStation(
    ↪ ChargingStationID,CLAT,CLON,_) & lat(LAT) & lon(LON) &
    ↪ CLAT == LAT & CLON == LON.

```

```

atStorage(StorageID):- storage(StorageID,SLAT,SLON,_,_,_) & lat(
    ↪ LAT) & lon(LON) & SLAT == LAT & SLON == LON.
atShop(ShopID):- shop(ShopID,SLAT,SLON,_,_) & lat(LAT) & lon(LON)
    ↪ & SLAT == LAT & SLON == LON.
atWorkShop(WorkshopID):- workshop(WorkshopID,SLAT,SLON) & lat(LAT
    ↪ ) & lon(LON) & SLAT == LAT & SLON == LON.

hasAllItems(JobID):-job(JobID,_,_,_,_,RequiredList)& not ( .
    ↪ member(required(Item,Qty),RequiredList) & not hasItem(Item
    ↪ ,Qty) ).
readyToBuildItem(JobID,Item):-job(JobID,_,_,_,_,RequiredList) & .
    ↪ member(required(Item,Qty),RequiredList) & ((not hasItem(
    ↪ Item,_) | (hasItem(Item,Qty1)&Qty>Qty1)) & item(Item,_,_,
    ↪ parts(Parts))&not(.member([Part,Qty2],Parts)&(not(hasItem(
    ↪ Part,_) | hasItem(Part,Qty3)&Qty2>Qty3))).
readyToBuildSubItem(JobID,Part):- job(JobID,_,_,_,_,RequiredList)
    ↪ & .member(required(Item,Qty),RequiredList) & ((not
    ↪ hasItem(Item,_) | (hasItem(Item,Qty1)&Qty>Qty1)) & item(
    ↪ Item,_,_,parts(Parts))&.member([Part,Qty2],Parts)&(not(
    ↪ hasItem(Part,_) | hasItem(Part,Qty3)&Qty2>Qty3)&item(Part,
    ↪ ,_,parts(Parts2))&not(.empty(Parts2))&not(.member([Part2,
    ↪ Qty4],Parts2)&(not(hasItem(Part2,_) | hasItem(Part2,Qty5)&
    ↪ Qty4>Qty5))).

shopForItem(JobID,Part,ShopID,Qty2):-job(JobID,_,_,_,_,
    ↪ RequiredList) & .member(required(Item,Qty),RequiredList) &
    ↪ ((not hasItem(Item,_) | (hasItem(Item,Qty1)&Qty>Qty1)) &
    ↪ item(Item,_,_,parts(Parts))&( .member([Part,Qty2],Parts)&(
    ↪ not(hasItem(Part,_) | hasItem(Part,Qty3)&Qty2>Qty3))&shop(
    ↪ ShopID,_,_,_,Inventory)& .member(item(Part,_,_),Inventory)
    ↪ ).
shopForSubItem(JobID,Part2,ShopID,Qty4):-job(JobID,_,_,_,_,
    ↪ RequiredList) & .member(required(Item,Qty),RequiredList) &
    ↪ ((not hasItem(Item,_) | (hasItem(Item,Qty1)&Qty>Qty1)) &
    ↪ item(Item,_,_,parts(Parts))&( .member([Part,Qty2],Parts)&(
    ↪ not(hasItem(Part,_) | hasItem(Part,Qty3)&Qty2>Qty3)) & item
    ↪ (Part,_,_,parts(Parts2))&not(.empty(Parts2))&( .member([
    ↪ Part2,Qty4],Parts2)&(not(hasItem(Part2,_) | hasItem(Part2,
    ↪ Qty5)&Qty4>Qty5))&shop(ShopID,_,_,_,Inventory)& .member(
    ↪ item(Part2,_,_),Inventory) ).
/* Initial goals */
!init.

```

```

/* Plans */
+!init:true<-
    if(false&role(truck,Speed,Load,MaxBattery,ToolList)&.
        ↪ member(MissingTool,ToolList)& (not (hasItem(
        ↪ MissingTool,_))& shop(ShopID, _, _, _, Inventory)
        ↪ & .member(item(MissingTool, _, _),Inventory)){
    .term2string(MissingTool,X);
    !!goBuy(ShopID,X,1,true);
    goto(ShopID);
    }
    else{
        if(role(truck,Speed,Load,MaxBattery,ToolList)){
            !!choseJob;
            skip;
        }
        else{
            !!skip;
            skip;
        }
    }
    .

+!skip:true<-
!!skip;
skip;
.

+!choseJob:true<-
    if(canDoJob(JobID)&job(JobID,Start,End,Rewawrd,Storage,
        ↪ RequiredList)){
        .print("JobID ",JobID," Start ",Start," End ",End,"
            ↪ Rewawrd ",Rewawrd," Storage ",Storage,"
            ↪ RequiredList ",RequiredList);
        +activeJob(JobID);
        !!decide;
        skip;
    }
    else{
        !!choseJob;
        skip
    }
    .

```

```

+!decide:activeJob(JobID)<-
  if(needToCharge){
    !!chooseChargingStation;
    skip;
  }
  else{
    if(hasAllItems(JobID)){
      !!standAndDeliver(JobID);// add logic for
        ↪ delivering for JobID
      skip;
    }
    else{
      if(readyToBuildItem(JobID, Item)){//check if
        ↪ we can construct an item thats
        ↪ required
        !!goBuild(Item);
        skip;
      }else{
        if(readyToBuildSubItem(JobID,Item)){
          .print("Going to build ",
            ↪ Item)
          !!goBuild(Item);
          skip;
        }
        else{//go buy items so that we can
          ↪ construct item
          if(shopForSubItem(JobID,Part,
            ↪ Shop,Qty)){
            !!goBuy(Shop,
              ↪ Part,
              ↪ Qty,
              ↪ false);
            goto(Shop);
          }
          else{
            if(shopForItem(JobID,
              ↪ Part,Shop,Qty))
              ↪ {
                !!goBuy(Shop,
                  ↪ Part,
                  ↪ Qty,
                  ↪ false);
                goto(Shop);
              }
          }
        }
      }
    }
  }
}

```



```

    for(chargingStation(Name,Lat,Lon,_)){
        actions.distance(Lat, Lon, AgentLat, AgentLon,
            ↪ Distance);
        ?minDistBel(N, D);
        if(Distance < D){
            -+minDistBel(Name, Distance);
        }
    }
    ?minDistBel(N, D);
    !!recharge(N);
    goto(N);
    .

+!recharge(ChargingStation) : needToCharge & not
    ↪ atChargingStation(ChargingStation) <-
    !!recharge(ChargingStation);
    goto(ChargingStation).

+!recharge(ChargingStation) : not needToCharge & (not
    ↪ atChargingStation(ChargingStation) | finishedCharging)<-
    !!decide.

+!recharge(ChargingStation) : atChargingStation(ChargingStation)
    ↪ & not finishedCharging <-
    !!recharge(ChargingStation);
    charge.

+!goBuild(Item): not atWorkShop(workshop0)<-
    !!goBuild(Item);
    goto(workshop0)
.
+!goBuild(Item): atWorkShop(workshop0)<-
    !!decide;
    assemble(Item)
.

+!evalJob:true<-
for(job(JobID,StorageID,_,_,,RequiredList)){
    for(.member(required(NameRequired,QtyRequired),
        ↪ RequiredList)){
        }
    }
}

```

```

    }
skip.
+!standAndDeliver(JobID):true<-
    if(job(JobID,StorageID,_,_,_,RequiredList)){
        !!deliver(StorageID, JobID);
        goto(StorageID);
    }
    else{
        !!decide;
    }
    .

+!deliver(StorageID, JobID): not atStorage(StorageID)<-
    !!deliver(StorageID, JobID);
    goto(StorageID)
    .

+!deliver(StorageID, JobID): atStorage(StorageID)<-
    .print("Storing now");
    !!decide;
    deliver_job(JobID)
    .

+!goShop: true <-
    if(job(.,.,.,.,.,RequiredList) & shop(ShopID, ., ., .,
        ↪ Inventory) & .member(required(NameRequired,
        ↪ QtyRequired),RequiredList) & .member(item(Name1,
        ↪ Price1, Qty1),Inventory) & Name1 == NameRequired){
        .print("We decided to buy something");
        if(QtyRequired > Qty1){
            !!goBuy(ShopID, NameRequired, Qty1,false);
            goto(ShopID);
        }
        else{
            !!goBuy(ShopID, NameRequired, QtyRequired,false);
            goto(ShopID);
        }
    }
    else{
        !!decide;
        skip;
    }
    .

```

```

+!goBuy(ShopID, NameRequired, Qty1, Init): not facility(ShopID)<-
    !!goBuy(ShopID, NameRequired, Qty1, Init);
    goto(ShopID).

+!goBuy(ShopID, NameRequired, Qty1, Init): facility(ShopID)<-
    .term2string(Qty1,X);
    buy(NameRequired,X);
    if(lastActionResult(Result)&Result == successful){
        if(Init){
            !!init;
            skip;
        }
        else
        {
            !!decide;
            skip;
        }
    }
    else{
        !!goBuy(ShopID, NameRequired, Qty1, Init);
        skip;
    }
    .

+!closest: true<-
    if(not(First == true)){
        FirstRun = true;
    }
    .

+!choseJob: true<-
    skip;
    .

+!evaluate(JobID):job(JobID,_,_,_,_,RequiredList)<-
    CanDo=true;
    TotalWeight;
    for(.member(required(Item,Qty),RequiredList)){
        if(item(Item,Weight,tools(ToolList1),Parts)){
        }
        for(.member(item(Item,Weight,tools(ToolList2),Parts
        ↪ ),Parts)){

```



```

        if( false==(ToolList2) | hasItem(
            ↪ ToolList2)){
            CanDo=false;
        }
    }
}
.

```

A.4 Internal Action Code(Java)

```

// Internal action code for project jasanting

package actions;

import jason.asSemantics.DefaultInternalAction;
import jason.asSemantics.TransitionSystem;
import jason.asSemantics.Unifier;
import jason.asSyntax.ASSyntax;
import jason.asSyntax.Term;

public class distance extends DefaultInternalAction {

    /**
     *
     */
    private static final long serialVersionUID =
        ↪ 3098238944781352745L;

    @Override
    public Object execute(TransitionSystem ts, Unifier un, Term
        ↪ [] args) {
        // execute the internal action
        ts.getAg().getLogger().info("executing internal action '
            ↪ actions.distance'");

        float distance = 999;

        float x1 = Float.parseFloat(args[0].toString());
        float y1 = Float.parseFloat(args[1].toString());
        float x2 = Float.parseFloat(args[2].toString());
        float y2 = Float.parseFloat(args[3].toString());
    }
}

```

```
        distance = Math.abs(x2 - x1) + Math.abs(y2 - y1);

        boolean returnValue = true;

        returnValue = un.unifies(args[args.length - 1], ASSyntax
            ↪ .parseNumber(String.valueOf(distance)));
        return returnValue;
    }
}
```

A.5 Server (Java)

```
package server;

import massim.Server;

public class RunServer {

    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                Server.main(new String[] { "-conf", "conf/
                    ↪ SampleConfig.json", "--monitor" });
            }
        }).start();
    }
}
```

A.6 EISAdapter(Java)

```
package jason.eis;

import eis.AgentListener;
import eis.EnvironmentInterfaceStandard;
import eis.EnvironmentListener;
```

```
import eis.exceptions.*;
import eis.iilang.*;
import jason.JsonException;
import jason.NoValueException;
import jason.asSyntax.*;
import jason.environment.Environment;
import massim.eismassim.EnvironmentInterface;

import java.util.*;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * This class functions as a Jason environment, using EISMASim
 * ↪ to connect to a MASSim server.
 * (see http://cig.in.tu-clausthal.de/eis)
 * (see also https://multiagentcontest.org)
 *
 * @author Jomi
 * - adapted by ta10
 */
public class EISAdapter extends Environment implements
    ↪ AgentListener {

    private Logger logger = Logger.getLogger("EISAdapter." +
        ↪ EISAdapter.class.getName());

    private EnvironmentInterfaceStandard ei;

    public EISAdapter() {
        super(20);
    }

    @Override
    public void init(String[] args) {

        ei = new EnvironmentInterface("conf/eismassimconfig.json")
            ↪ ;

        try {
            ei.start();
        } catch (ManagementException e) {
            e.printStackTrace();
        }
    }
}
```

```
    }

    ei.attachEnvironmentListener(new EnvironmentListener() {
        public void handleNewEntity(String entity) {}
        public void handleStateChange(EnvironmentState s) {
            logger.info("new state "+s);
        }
        public void handleDeletedEntity(String arg0,
            ↪ Collection<String> arg1) {}
        public void handleFreeEntity(String arg0,
            ↪ Collection<String> arg1) {}
    });

    for(String e: ei.getEntities()) {
        System.out.println("Register agent " + e);

        try {
            ei.registerAgent(e);
        } catch (AgentException e1) {
            e1.printStackTrace();
        }

        ei.attachAgentListener(e, this);

        try {
            ei.associateEntity(e, e);
        } catch (RelationException e1) {
            e1.printStackTrace();
        }
    }
}

@Override
public void handlePercept(String agent, Percept percept) {}

@Override
public List<Literal> getPercepts(String agName) {

    Collection<Literal> ps = super.getPercepts(agName);
    List<Literal> percepts = ps == null? new ArrayList<>() :
        ↪ new ArrayList<>(ps);

    clearPercepts(agName);
}
```

```

    if (ei != null) {
        try {
            Map<String,Collection<Percept>> perMap = ei.
                ↪ getAllPercepts(agName);
            for (String entity: perMap.keySet()) {
                Structure strcEnt = ASSyntax.createStructure("
                    ↪ entity", ASSyntax.createAtom(entity));
                for (Percept p: perMap.get(entity)) {
                    try {
                        percepts.add(perceptToLiteral(p).
                            ↪ addAnnots(strcEnt));
                    } catch (JSONException e) {
                        e.printStackTrace();
                    }
                }
            }
        } catch (PerceiveException e) {
            logger.log(Level.WARNING, "Could not perceive.");
        }
    }
    return percepts;
}

@Override
public boolean executeAction(String agName, Structure action)
    ↪ {
    if (ei == null) {
        logger.warning("There is no environment loaded!
            ↪ Ignoring action " + action);
        return false;
    }

    try {
        ei.performAction(agName, literalToAction(action));
        return true;
    } catch (ActException e) {
        e.printStackTrace();
    }

    return false;
}

/** Called before the end of MAS execution */
@Override

```

```

public void stop() {
    if (ei != null) {
        try {
            if (ei.isKillSupported()) ei.kill();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    super.stop();
}

private static Literal perceptToLiteral(Percept per) throws
    ↪ JasonException {
    Literal l = ASSyntax.createLiteral(per.getName());
    for (Parameter par: per.getParameters())
        l.addTerm(parameterToTerm(par));
    return l;
}

private static Term parameterToTerm(Parameter par) throws
    ↪ JasonException {
    if (par instanceof Numeral) {
        return ASSyntax.createNumber(((Numeral)par).getValue()
            ↪ .doubleValue());
    } else if (par instanceof Identifier) {
        try {
            Identifier i = (Identifier)par;
            String a = i.getValue();
            if (!Character.isUpperCase(a.charAt(0)))
                return ASSyntax.parseTerm(a);
        } catch (Exception ignored) {}
        return ASSyntax.createString(((Identifier)par).
            ↪ getValue());
    } else if (par instanceof ParameterList) {
        ListTerm list = new ListTermImpl();
        ListTerm tail = list;
        for (Parameter p: (ParameterList)par)
            tail = tail.append( parameterToTerm(p) );
        return list;
    } else if (par instanceof Function) {
        Function f = (Function)par;
        Structure l = ASSyntax.createStructure(f.getName());
        for (Parameter p: f.getParameters())
            l.addTerm(parameterToTerm(p));
    }
}

```

```

        return l;
    }
    throw new JasonException("The type of parameter "+par+" is
        ↪ unknown!");
}

private static Action literalToAction(Literal action) {
    Parameter[] pars = new Parameter[action.getArity()];
    for (int i = 0; i < action.getArity(); i++)
        pars[i] = termToParameter(action.getTerm(i));
    return new Action(action.getFunctor(), pars);
}

private static Parameter termToParameter(Term t) {
    if (t.isNumeric()) {
        try {
            return new Numeral(((NumberTerm) t).solve());
        } catch (NoValueException e){
            e.printStackTrace();
        }
        return new Numeral(null);
    } else if (t.isList()) {
        Collection<Parameter> terms = new ArrayList<>();
        for (Term listTerm: (ListTerm)t)
            terms.add(termToParameter(listTerm));
        return new ParameterList( terms );
    } else if (t.isString()) {
        return new Identifier( ((StringTerm)t).getString() );
    } else if (t.isLiteral()) {
        Literal l = (Literal)t;
        if (!l.hasTerm()) {
            return new Identifier(l.getFunctor());
        } else {
            Parameter[] terms = new Parameter[l.getArity()];
            for (int i = 0; i < l.getArity(); i++)
                terms[i] = termToParameter(l.getTerm(i));
            return new Function( l.getFunctor(), terms );
        }
    }
    return new Identifier(t.toString());
}
}
}

```


Bibliography

- [1] Tobias Ahlbrecht et al. Eismassim Documentation. <https://github.com/agentcontest/massim/blob/master/docs/eismassim.md>. Accessed June 2017.
- [2] Tobias Ahlbrecht et al. Massim Scenario Documentation. <https://github.com/agentcontest/massim/blob/master/docs/scenario.md>. Accessed June 2017.
- [3] Tobias Ahlbrecht et al. Massim starter kit for jason 2.1. <https://github.com/agentcontest/massim/tree/master/starterKits/jason>. Accessed June, 2017.
- [4] Tobias Ahlbrecht et al. The Multi-Agent Programming Contest. <https://multiagentcontest.org/>. Accessed June 2017.
- [5] Tobias Ahlbrecht et al. Multi-Agent Programming Contest releases. <https://github.com/agentcontest/massim/releases>. Accessed June 2017.
- [6] Tobias Ahlbrecht et al. Programming Multi-Agent Systems in AgentSpeak Using Jason. <http://jason.sourceforge.net/jBook/SlidesJason.pdf>. Accessed June 2017.
- [7] Rafael H. Bordini and Jomi F. Hübner. DefaultInternalAction. <http://jason.sourceforge.net/api/jason/asSemantics/DefaultInternalAction.html>. Accessed June 2017.
- [8] Rafael H. Bordini and Jomi F. Hübner. Jason. <http://jason.sourceforge.net/>. Accessed June 2017.

- [9] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Wiley, 2007.
- [10] Maicon Rafael Zatelli. Jason Eclipse plugin. <http://jason.sourceforge.net/mini-tutorial/eclipse-plugin/>. Accessed June 2017.