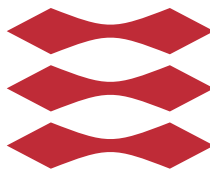


Organizing Heterogeneous Agents

Andreas Frøsig – s093264

Kenneth Balsiger Andersen – s093252

DTU



Kongens Lyngby 2015
DTU Compute-M.Sc.-2015

Technical University of Denmark
DTU Compute
Richard Petersens Plads, building 324,
DK-2800 Kgs. Lyngby
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk DTU Compute-M.Sc.-2015

Summary (English)

The goal of the thesis is to organize heterogeneous agents in a multi-agent system across multiple languages by adding an organizational layer through the existing system, AORTA. In order to make dependency communication possible across multiple languages we will extend AORTA to use RMI to send organizational messages between the agents, hence allowing for multi platform organizing.

We use the environment StarCraft:Brood War which is accessed through the environment interface BWAPI Bridge. This environment interface have been extended to support several multi-agent systems at once, in order to test the possibility of organizing agents across several instances of multi-agent systems.

Through the implementation and the testing we have shown that multi platform organization is possible, but the current state of the used systems is not optimized enough for a continuous environment like StarCraft:Brood War.

Summary (Danish)

Målet for denne afhandling er at organisere heterogene agenter i et multi-agent system på tværs af flere sprog ved at tilføje et organiserende lag ved hjælp af det eksisterende system, AORTA. For at gøre det muligt at have organisationsrelaterede afhængigheder på tværs af sprog vil vi udvide AORTA til at bruge RMI til at sende de organisationsrelaterede beskeder igennem, hvorved man muliggør organisering over flere platforme.

Vi bruger spillet StarCraft:Brood War som miljø for vores agenter. Vi har adgang til StarCraft igennem BWAPI Bridge som vi har udvidet til at understøtte tilkobling af flere multi-agent systemer samtidig. Disse multi-agent systemer kan være lavet i forskellige sprog, hvilket gør det muligt for agenter skrevet i forskellige sprog at arbejde sammen i samme miljø.

Ved at implementere og teste vores system har vi vist at det er muligt at lave organisationer på tværs af sprog og instanser af systemer, men også at den nuværende tilstand af systemet ikke er optimeret tilstrækkeligt til at kunne fungere i et miljø hvor tiden er en faktor.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a M.Sc. in Informatics.

The report deals with the project we have made for our master thesis. The project is about multi-agent systems and in particular the organization of heterogeneous agents in such a system. The goal is to add an organizational layer, to the agents, which spans across several multi-agent systems in the same environment. To achieve this we have extended some existing systems, namely the environment we used, BWAPI Bridge, and the organizational extension to Jason, AORTA.

A big part of the project has been about acquainting oneself with new knowledge about several new subjects.

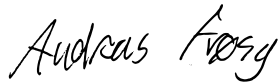
APLs: We had to learn how Agent Programming Languages worked, both Jason and GOAL, so that we could organize agents in those languages and try to extend them with an organizational layer.

BWAPI Bridge: In order to understand APLs we needed to understand how the environment worked, seeing as we tested the APLs in it. We had to understand how to use it and how it worked as we extended it to accept several multi-agent systems at the same time.

AORTA: When we understood how the APLs worked in the system, we needed to understand AORTA, both how to use it, and how to extend it.

The thesis starts by describing the theory behind multi-agent systems, including the logic we have used and the organizational extension we have used, AORTA. Then we describe the implementation of our system, including how we have made distributed multi-agent systems organized and how the organization work. Lastly we test to see how the distribution and organization performed and conclude on the result.

Lyngby, 01-February-2014-2015



Andreas Frøsig – s093264
Kenneth Balsiger Andersen – s093252

Acknowledgements

We would like to thank our supervisors Jørgen Villadsen and Andreas Schmidt Jensen for the feedback they have provided us with throughout the project.

Additionally we would like to thank Andreas Schmidt Jensen for the implementation of AORTA and the quick response when issues occurred.

We would also like to thank Koen Hindriks and Wouter Pasman who helped us compile and understand GOAL and EIS.

Contents

Summary (English)	i
Summary (Danish)	iii
Preface	v
Acknowledgements	vii
1 Introduction	1
1.1 Required Systems	2
2 Theory	5
2.1 Intelligent agent	5
2.1.1 Knowledge representation	6
2.1.2 Reasoning	9
2.1.3 Agent control loop	10
2.1.4 Plan	13
2.2 Environment	15
2.2.1 Properties of an environment	16
2.2.2 StarCraft: Brood War	17
2.2.3 EIS	19
2.3 Multi-Agent Systems	19
2.3.1 Types of MAS	21
2.3.2 Decision making	22
2.3.3 Organizational layer	23
2.4 AORTA	24
2.4.1 Concept	24
2.4.2 Meta Model	24
2.4.3 Reasoning cycle	26

2.4.4	AORTA program	27
2.4.5	Multi-language	27
3	Implementation	29
3.1	BWAPI Bridge	29
3.1.1	Structure	30
3.1.2	Agent interface	30
3.2	StarCraft agents	32
3.2.1	Knowledge	34
3.2.2	Aspects of the simulation	36
3.2.3	Implicit organization	38
3.2.4	Explicit organization	40
3.3	RMI BWAPI Bridge	47
3.3.1	Structure	47
3.3.2	Requirements	48
3.4	Communication between AORTA instances	51
3.4.1	Analysis	51
3.4.2	Result	52
3.5	Implementation discoveries	54
4	Tests and Discussions	57
4.1	Basic agent tests	57
4.1.1	Testing of GOAL and Jason	58
4.2	AORTA test	58
4.2.1	Parse issues	61
4.2.2	Understanding	62
4.2.3	Functionality issues	62
4.2.4	Prolog issues	63
4.3	AORTA functional tests	63
4.4	Remote test	64
4.4.1	GOAL-Jason RMI	72
4.4.2	AORTA Message Server	72
5	Conclusion	75
5.1	BWAPI Bridge: SC:BW as environment	75
5.2	AORTA as organizational layer	76
5.3	GOAL-AORTA Bridge	76
5.4	RMI BWAPI Bridge	77
5.5	AORTA Communication Extension	77
5.6	GOAL and Jason agents	77
5.7	Final Remarks	78

A	Running and compiling the systems	79
A.1	Jason AORTA	80
A.1.1	Single system	80
A.1.2	Local multiple clients system	80
A.2	GOAL	81
A.3	GOAL-AORTA Bridge	81
B	Test Results	83
B.1	Functional test Replay	84
B.2	Remote AORTA test with Jason agents	84
B.2.1	Distributed	85
B.3	Final GOAL agents	88
C	Source code	91
C.1	System	92
C.2	Agents	93
C.2.1	Jason: Aorta organization	93
C.2.2	Jason: Agents for AORTA organization	95
C.2.3	GOAL: Agents for AORTA organization	101
	Acronym List	119
	Bibliography	121

Introduction

In this project we work with a system containing several intelligent heterogeneous agents (see definition 2.15) in an environment (see section 2.2) where these agents can interact and perceive. Since the agents depend on each other they need to cooperate in order to achieve common goals efficiently. Systems like these are known as Multi-Agent Systems (MAS), and it is the organization of such a system that is the main focus in this project.

The environment, we have chosen to use, is the computer game from 1998, StarCraft: Brood War (SC:BW) [5]. It is a classic Real Time Strategy game with three different races: each with many different units where many of them have unique abilities.

The complexity of this environment is sufficient to demonstrate the benefits of organization in a MAS since we have a lot of different units each corresponding to a heterogeneous agent which depends on each other. By using a game as environment, we automatically get a graphical representation of the system and can easily demonstrate and monitor what the agents are doing.

Our goal for this project is to show how to organize a MAS across multiple languages (or even platforms). To do this, we do not need a complete system including all races and units, making it possible to use a prototype like the BWAPI Bridge which only implements parts of the a single race, namely the

Terran. It is an interface for the environment we are using, and it was created by Christian Kaysø-Rørdam for his masters project. It is part of the bridge between SC:BW and the agents; see section 3.1 for more information on how the project works [8].

To illustrate how important organization can be, we will implement some agents in an Agent Centered MAS (ACMAS). Here it will be necessary to implement the organization directly in the agents behaviour, making the implementation more complex than it needs to be. A description of these agents can be seen in section 3.2.3. To accommodate the problem of organizing agents, we will implement an Organizational Centered MAS (OCMAS) using Adding Organizational Reasoning To Agents (AORTA) to make an explicit organizational layer on our agents [2]; since it is still in alpha state, we will be alpha testing it. This will also make it possible to organize agents across multiple languages since the organization should be the same in the different languages making the organization reusable.

1.1 Required Systems

Multiple systems are used in this project; here we will describe what each of them are used for:

EIS: This is an Environment Interface Standard that a lot of Agent Programming Languages (APLs) use in order to connect an environment to the agents. BWAPI Bridge and both APLs that we have been using, namely GOAL and Jason, implements this.

BWAPI Bridge: The BWAPI bridge is a bridge from JNIBWAPI which translates knowledge for the agents from the API to percepts in the APL and gives the APL a possibility to interact with the environment through actions on the agents.

AORTA: Adding Organizational Reasoning To Agents (AORTA) is an addon that adds an organizational layer to agents making it possible to implement an explicit organization. This is the system, we will be using to organize our agents in this project. AORTA is in alpha state, so we will be using it as alpha-testers. It is currently implemented for Jason, and our goal is to implement it for the APL GOAL.

Jason: This is an APL where AORTA is currently working.

GOAL: This is the APL that we plan to use together with Jason. AORTA does not work with GOAL making the GOAL-AORTA bridge one of our goals for this project.

In the project we use a lot of acronyms and a list can be seen in appendix C.2.3.



Figure 1.1: Screenshot of a small base created by our ACMAAS GOAL agents.

In this chapter we will describe the theory used in this project. This includes required knowledge about Intelligent Agents, MAS, and the existing systems we want to use.

2.1 Intelligent agent

When implementing an Artificial Intelligence some sort of intelligent agent is required. In this section we will describe what an Intelligent Agent is and what it is capable of.

Definition 2.1 (Intelligent agent) *An intelligent agent is defined as an autonomous entity with its own mental state for information storage, e.g., knowledge and beliefs. It is able to perceive and interact with its environment (see section 2.2) and thus being able to pursue its goals. The agent will do this by applying actions from its set of capabilities, see definition 2.2.* \square

How the agent chooses what to do can be implemented in a number of ways, here we will describe three of the most commonly used since they are the ones

we use in this project.

Simple Reflex Agent: An agent of this type will choose its actions based on the *current* percepts, i.e., the state of the environment exactly as the agent sees it with no history. Reflexive agents do not plan more than a single step ahead.

Model-based Reflex Agents: Agents of this type will save important knowledge from the environment in an internal state for later usage.

Goal-based agents: Agents might not be able to find an applicable action solely based on knowledge of the environment. They have a set of tasks that they want to pursue on which they base their choices. These are commonly referred to as goals, and how the agent reaches a certain goal is referred to as a plan for that goal.

Definition 2.2 (Capabilities) *An Intelligent Agent has a set of actions that it is able to do in order to interact with its environment or alter its own knowledge base. These actions are called the capabilities of the agent.*

2.1.1 Knowledge representation

Knowledge of an intelligent agent can take shape in various ways. In our system we use modal logic to represent knowledge, more precisely the epistemic logic for the knowledge and belief representation and temporal logic to describe the time aspect of the system.

In this report we only use it to formally describe the knowledge of our agents and not for calculating purposes. For calculation and further understanding of modal logic we refer to [12].

2.1.1.1 Epistemic logic

We distinguish between what an agent *believes*, what it *knows*, and its *knowledge* about the *knowledge* of other agents. Definitions of the different kinds of *knowledge* that we use in this project can be seen in Definition 2.3 - 2.7. One thing to note is that most of these definitions are a way to distinguish *knowledge*, meaning that they are not necessarily represented like this directly in the code.

Definition 2.3 (Knowledge) *Knowledge of an agent is defined as something the agent knows to be true. We denote this:*

$$\mathcal{K}_i\varphi,$$

which means that agent i knows φ is true. \square

Definition 2.4 (Belief) *Belief of an agent is almost the same as knowledge except for the very important distinction that even though an agent believes something to be true, it might not be the case. We denote this:*

$$\mathcal{B}_i\varphi,$$

which means that agent i believes that φ is true.

Definition 2.5 (Group Knowledge) *Group Knowledge means that everybody in a group knows something to be true but have no idea if anybody else in the group knows it as well. We denote this:*

$$E_G\varphi,$$

which means that everybody in group G knows that φ is true, but are not aware if anybody else knows it. \square

Definition 2.6 (Common Knowledge) *Knowledge, that everybody in a given group knows, that everybody knows, that everybody knows recursively 'ad infinitum'.*

$$\mathcal{C}_G = \bigwedge_{n=0}^{\infty} E_G^n\varphi,$$

where $E_G^2\varphi$ is $E_G E_G\varphi$. The importance of common knowledge is shown in example 2.1. \square

Definition 2.7 (Distributed Knowledge) *Knowledge that can be derived from the knowledge of all agents in a given group. We denote this:*

$$\mathcal{D}_G\varphi,$$

meaning that φ can be derived from the knowledge of all agents in group G . \square

Example 2.1 (The Coordinated Attack Problem) *The Coordinated Attack Problem is about two generals that need to coordinate an attack. Unfortunately the communication is unreliable as the only way of communication is by messengers sent through a dangerous valley. The problem arises as general A writes to general B that he want to attack at dawn, but they have to attack at the same time. If A does not know that B has received the message he will not attack, the same goes for B. As B send the acknowledgment of the attack, he needs to know if A has received his acknowledgment as he knows that A will not attack unless he does; hence B will not attack before he knows that A has received the acknowledgment. This depth of knowledge about the acknowledgments of the attacks are infinite, meaning that they can never be sure that the other general will attack. This is why common knowledge can be crucial in coordination and is required in a nondeterministic environment [17].*

2.1.1.2 Temporal logic

Temporal logic is used to logically define the aspect of time. We only use Linear Temporal Logic and in the following definitions we use $M, s \models$ to denote that we are currently in the world M at the state s , where the state s which is the state of the environment at a given time step. Furthermore we use φ and ω as logical formula.

The parts of the temporal logic we use in our system is taken from [18, 11] and can be seen here:

Definition 2.8 (Always operator) *The always operator of the temporal logic means that what comes next will be true in all future states. We denote this:*

$$M, s \models \mathcal{A}\varphi$$

□

Definition 2.9 (Future operator) *The future operator of temporal logic means that what comes next will at some point in the future become true, i.e., it will happen at a later state than the state this operator was concluded. We denote this:*

$$M, s \models \mathcal{F}\varphi$$

The relation between the strong operator \mathcal{A} and the weaker \mathcal{F} are:

$$\mathcal{A}\varphi \equiv \neg\mathcal{F}\neg\varphi$$

□

Definition 2.10 (History operator) *The history operator of the temporal logic means that what comes next has always been true. We denote this:*

$$M, s \models \mathcal{H}\varphi$$

□

Definition 2.11 (Past operator) *The past operator of the temporal logic means that what comes next has at some point in the past been true. We denote this:*

$$M, s \models \mathcal{P}\varphi$$

The relation between the strong operator \mathcal{H} and the weaker \mathcal{P} are:

$$\mathcal{H}\varphi \equiv \neg\mathcal{P}\neg\varphi$$

□

Definition 2.12 (Until operator) *The until operator of the temporal logic is used to describe states that mark the interval in which a certain knowledge hold. The knowledge that φ is consistent from s until ω is the case, given the world M is denoted:*

$$M, s \models \varphi\mathcal{U}\omega$$

Until can be defined as:

$$M, s_x \models \varphi\mathcal{U}\omega \equiv (M, s_y \models \omega) \wedge \forall i (x \leq i < y \Rightarrow M, s_i \models \varphi),$$

where i is the time in a linear time line, and s_i denotes the state at time i . □

2.1.2 Reasoning

An agent needs to have some sort of reasoning in order to be an intelligent agent. In this project we use logical reasoning based on the BDI model (see

section 2.1.3), seeing as this is the reasoning of the languages we have chosen to use.

The subgroup of logical reasoning we use, is the deductive reasoning, in which we make logical conclusions based on beliefs and knowledge according to standard logical rules [21]. An example of deductive reasoning can be seen in the following example 2.2.

Example 2.2 (Deductive reasoning) *An example of deductive reasoning, from StarCraft (see section 2.2.2), would be: if we know that units of type Terran Command Center is a building, and we know that unit A1 is a Terran Command Center; then we can conclude that A1 is a building. Formally written:*

$$\frac{\begin{array}{l} \models \text{isBuilding}(\text{"TerranCommandCenter"}) \\ \models \text{enemy}(\text{A1}, \text{"TerranCommandCenter"}, _, _, _, _) \\ \models \text{enemy}(\text{Name}, \text{Type}, _, _, _, _) \wedge \text{isBuilding}(\text{Type}) \rightarrow \text{isBuilding}(\text{Name}) \end{array}}{\models \text{isBuilding}(\text{A1})}$$

2.1.3 Agent control loop

When designing autonomous agents a common paradigm is the *Belief Desire Intention* (BDI) software model, where *Beliefs* represent both *knowledge* and *beliefs*, *Desire* represent *goals*, and *Intentions* are simply the *actions* that the agent intend to do.

Plans and *goals* are not part of this model, and how they are represented are up to the programmer. The same goes for the distinction between *Beliefs* and *Knowledge*, as defined in Definition 2.4 and 2.3, if the programmer wants to distinguish at all. Distinguishing between *Beliefs* and *Knowledge* can even vary for different forms of *beliefs* and hence it is often reflected in the *plan* on the agent.

The main idea of the BDI software model is to separate the mental state of the agent into *beliefs*, *desires* and *intentions* in order to have more maintainable code as well simulating human reasoning. This happens in cycles where the functionality can be split into three parts, namely the **Belief Revision Function** the **Option Generation** and the **Filtering Function**.

After these three phases we get to the planning part where the agent can choose to interact with the environment. What to do depends on the *goal*, and how the agent determines which *goal* to pursue can be implemented in many ways. Here we will treat it as a black box that, given the *intentions* and the *Beliefs*,

finds a plan for the current cycle, we call it the **Plan Generation**. The last phase is called the **Action Execution** and it allows the agent execute an *action* by taking the first *action* from the generated *plan*. Planning this way lets the agent take new *knowledge* into consideration each cycle, making it possible for the agent to alter plans during runtime. Each step can be seen described more formally below, and an overview can be seen in figure 2.1.

Belief Revision Function: This function will update the *beliefs* with the new *knowledge* perceived from the environment. We denote this:

$$\wp(\text{Beliefs}) \times \text{Percept} \rightarrow \wp(\text{Beliefs})$$

Option Generation: This phase will generate a set of all possible *options* from the *beliefs* which then becomes the *desires* of the agent. We denote this:

$$\wp(\text{Beliefs}) \rightarrow \wp(\text{Desires})$$

Filter Function: This function will, from the set of all *beliefs* and *desires*, find the *intentions* of the agent. We denote this:

$$\wp(\text{Beliefs}) \times \wp(\text{Desires}) \rightarrow \wp(\text{Intentions})$$

Plan Generation: This phase will, given the *beliefs* and the *intentions* of the agent, generate a plan of actions. Note that the plan can even be to do nothing if the agent chooses to. We denote this:

$$\wp(\text{Beliefs}) \times \wp(\text{Intentions}) \rightarrow \wp(\text{Plan}),$$

Note that the **Plan Generation** can be extended taking other aspects into consideration like, e.g., AORTA which adds organizational goals (*intentions*) to the agent. See more on this in section 2.4.

Action Execution: Given a *plan* the agent can then interact with the environment by executing the first *action* in the plan. We denote this:

$$\wp(\text{Plan}) \rightarrow \wp(\text{Action})$$

2.1.3.1 Agent Programming Languages

When programming intelligent agents following the BDI model a specific programming paradigm is often used, namely Agent-Oriented Programming (AOP).

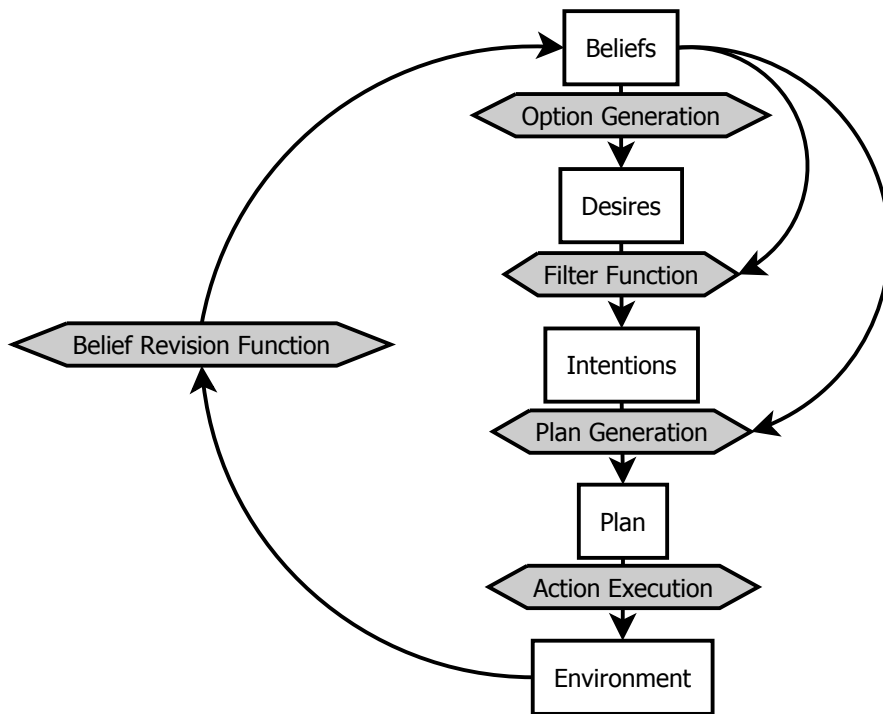


Figure 2.1: BDI control flow.

This paradigm is a subset of the Object-Oriented Paradigm (OOP), where objects freely interact with each other. The differences are that AOP have limited and clearly separated mental components (variables), such as *beliefs*, *capabilities* and *decisions* each of which follows a precise syntax[19], as well as strictly defined interaction options. These mental components are stored in a single set which is the *mental state* of the agent, and the separation and restriction is added to simulate human intelligence.

Programming languages following this paradigm are called Agent Programming Languages (APLs), and in these languages the agents are in focus and each agent is connected to the others through some sort of *messaging capabilities*.

2.1.4 Plan

When deciding what needs to be done, the agent needs to figure out which *goals* to pursue and in which order. *Plans* can be illustrated using a *plan tree* where the root is the final *goal*, for the specific *plan*, and all other nodes are *sub-goals*, possibly required for the final *goal* to be accomplished. To illustrate *plans* we use the *plan tree* structure from [16] which applies three different nodes to show if all or just some of the *sub-goals* should be done, or if there should be a specific order in which they should be solved:

Either: This means that at least one of the children of a *goal* has to be done before the *goal* itself can be considered, shown in figure 2.2a.

All: This means that all the children need to be accomplished before the *goal* can be considered, as shown in figure 2.2b.

Sequence: This means that the children of a *goal* need to be accomplished in the order left to right; i.e., a child is not considered before all the other children to the left of it are accomplished. This is not strictly necessary as making the nodes to the left children of those to the right would have the same effect, but for large *plan trees* this can make the tasks easier to organize. Shown in figure 2.2c.

For an example of a *plan tree* see example 2.4.

Which *goal* to pursue is then usually based on the *gain* of accomplishing the *goal*, which is calculated by subtracting the *cost* of accomplishing the *goal* from the *value* of the *goal*.

$$Gain(g) = Value(g) - Cost(g, a) \quad (2.1)$$

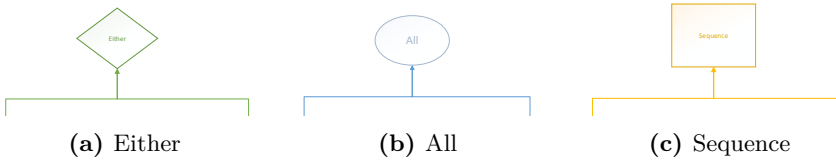


Figure 2.2: Representation of different nodes in the *plan tree*.

where g is a specific *goal* and a is an agent. Note that in a system with only one agent, the a is redundant.

Value of a goal: The *value* of a *goal* is assessed by the agent and is often based on its distance to the root in the *plan tree*, as well as the raw value of the *goal* if one is given by the environment. E.g., in a taxi driver environment (see example 2.3) a *goal* could be to acquire a car. In figure 2.3 we can see that the distance to the root of the *plan tree* for "deliver customer" would be three or four depending on whether the car has a GPS or not, as this would automatically solve the *goal* of "Bring map". As an added value the car could have better fuel economy making that car more valuable for the system since the long term profit would be larger due to cost reductions of some future *goals*.

Cost of a goal: The *cost* of achieving a *goal* for an agent is usually calculated based on the amount of time and resources used to achieve it. In the taxi example, (see example 2.3) the cost of acquiring a car would be the time and/or money spend for obtaining it.

Knowing the *gain* of all *goals*, the agent can determine which *goal* to pursue by taking the one with the largest *gain*. This method could correspond to the **Plan Generation** and **Action Execution** that we dealt with as a black box algorithm when describing the control flow using the BDI model, see section 2.1.3.

Example 2.3 (Taxi environment) *The taxi environment is situated in a city, where a robotic taxi driver is tasked to get a taxi and transport customers from one point in the city to another.* □

Example 2.4 (Plan for taxi) *If we want to deliver a customer in the taxi environment we need to drive a car and pick up the customer. But in order to drive a car we need to get one if we are not already in one. Furthermore we need to make sure we have change for the customer and some kind of map. In*

some cars this goal would be solved implicitly if it had a GPS. The plan tree for this can be seen in figure 2.3. \square

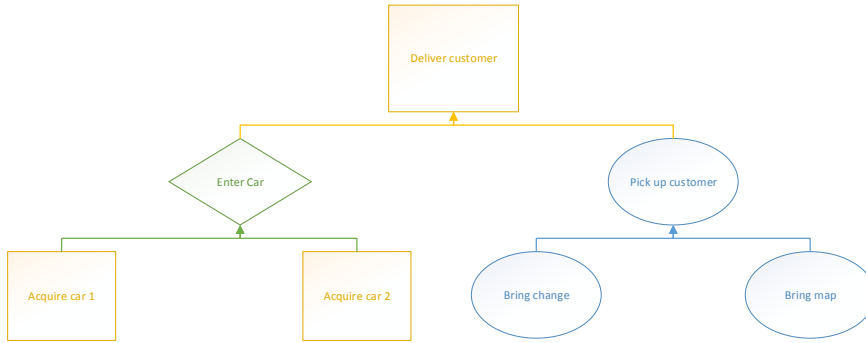


Figure 2.3: An example of a *plan tree* for delivering a customer in the taxi environment.

2.2 Environment

As described in section 2.1 an agent needs an environment to perceive and interact with. In this section we will describe what an environment is and introduce some definitions needed in order to describe how the environment works.

When speaking about AI and especially MASs, one very important element is the environment. This is basically the world in which the agent(s) are going to try to achieve their goals. An environment can be anything from a very simple environment including one button which can be *on* or *off* to a large game like SC:BW or even a real life situation given a robot interacting with the world.

Since an agent is defined to be able to interact and perceive with the environment (see definition 2.1) the environment is required to provide the following:

1. Percepts that can be sent to the agents as a way to make them perceive things about the environment. These can be the state of a button or the amount of minerals in a mineral field in SC:BW.
2. Actions that agents can send to the environment in order to interact with it. These can be press a button or build a building in SC:BW.

The entire required interface between the agents and the environment, namely the percepts and available actions, are defined by the environment.

2.2.1 Properties of an environment

When classifying an environment, there are some properties that needs to be clarified:

Observability: It is important to determine whether the environment is *partially* or *fully observable*. This refers to *knowledge* about the environment, and whether the agents get every relevant information or just some of it. If the agents can access all information about the current state of the system it is *fully observable* otherwise it is only *partially observable*.

Determinism: Whether an environment is *deterministic* refers to the consequence of an action. If the expected result of an action always is the result then the environment is *deterministic* otherwise it is *non-deterministic*. If different outcomes of an action is given by a probability the environment is said to be *stochastic*.

Dynamics: We need to consider whether the environment is *static* or *dynamic*. A *static* environment only changes as a response to the interaction of an agent. *Dynamic* environments changes even when the agents does nothing. Systems which includes entities beyond the control of the agents can be considered *dynamic*, as these other entities might change the state of the environment hence the agents must perceive the environment again for every decision, not taking previously observed information for granted.

Discreteness: Whether an environment is *discrete* or *continuous* refers to the amount of possible *actions*. Per definition if this number is finite the system is *discrete* otherwise it is *continuous*. However systems might be treated as *continuous* if it is simulating a real time scenario, as everything in a computer strictly speaking is discrete.

Episodic: If a system is running for a long time it is important to realize whether the environment is *episodic* or *sequential*. These terms refer to the dependency on previous states. If a system is independent on previous states or all dependencies are removed regularly, the environment is said to be *episodic*. An example of this could be a factory machine which after every item erases its knowledge as the state of the previous item does not effect the next one. In a *sequential* environment the agents need to consider all previous states as they might influence the future; e.g., a taxi

driver spotting some ice on the road which should be remembered so that path is not chosen again [20].

2.2.2 StarCraft: Brood War

In order to understand the environment we first want to explain how it works as a game, and then how it affects its properties as an environment.

The environment we use is the Real Time Strategy (RTS) game from 1998 by Blizzard, StarCraft: Brood War. The basics about SC:BW and how it works as an environment will be described in this sub-section.

2.2.2.1 Game properties

The game is set in a far future where three races fight each other on different planets for ground and resources. It has three very different races where various strategies are required, each race includes several units each with its own special powers. The three races are:

Terran: This is the human race in the game, a technologically advanced version of the human species. They use fire, bullets and explosions to kill their enemy exactly like current war fare. Their war tactics and appearance are much like an advanced version of current military, with earth bound tanks, power suited infantry and airplanes all together for combined assault.

Zerg: This is a hive minded alien race which seeks genetic perfection instead of technological advancements. This race uses different sub races of the Zerg race to do the different tasks, and even their "buildings" are transformed workers. Their appearance varies a lot as each unit is a different creature, and their war tactic is based on cheap and easily produced units which results in quantity over quality.

Protoss: Protoss is a large and physically strong humanoid race, much better connected to the psychic powers than the Terran. They are extremely technologically advanced and are in many ways the diametrical opposites of the Zerg. In the game they are expensive and slowly produced, but strong which makes them prefer quality over quantity. Additionally all buildings and units are protected by an energy shield.

The game is played as a classical RTS game where each player has a base and is required to gather resources in order to expand the base, research new upgrades, and train military units. These units will allow the player to attack and in turn annihilate the opponents units which is how the game is won. As it is a RTS the player can at any point in time order the units to perform any action within the units capabilities.

2.2.2.2 Environment properties

As an environment for a MAS it is very complex, as we have three different races composed of several heterogeneous units, and several implicit and explicit dependencies. The dependencies are due to the order in which the units needs to be created, i.e., some units can only be created if other units have been created. E.g., we need to train Marines, but as the Barracks is the one training Marines this is implicitly dependent, whereas Firebats also need the Academy which is explicitly mentioned in the game.

Aside from the static size of the map, we only get percepts from the visible area of the map making the environment *partially observable*.

It is *non-deterministic* as actions can fail if it is either: illegal, not possible in the current state, or the agent is interrupted.

The environment is in itself *static*, but as we have opponents and neutral units on the map that can change parts of the map, we have to consider it *dynamic* and update percepts even for objects we have already perceived.

The environment is *discrete* as there are a limited number of possible actions. However the number of possible actions is very large as an agent can use any one of their abilities at any point in time at any point on the map. The smallest map is 64×64 build tiles and each of these are 32×32 pixels. This means that even though time in a computer is *discrete* and the number of possible *actions* at every point in time is finite, we have to consider the environment *continuous* as the possibilities are too numerous.

As we only run a single play through of the game, we can perceive the environment as *sequential* which means that we need to save certain information such as last known location of the opponents base.

2.2.2.3 Units

The environment contains a lot of units that we do not use in this project. In Table 2.1 on page 20 a list of the units we have used is shown. The description of the units is based on how they differ from the other units.

The reason we have not been using more units, is a combination of the fact that they are not all implemented in the BWAPI Bridge, and that we do not need all units in order to show how to organize heterogeneous agents using AORTA.

2.2.3 EIS

The Environment Interface Standard (EIS) is an interface making it easier to connect agents to an environment. For normal environments the `EIDefaultImpl` can be extended which includes a lot of the necessary handling of entities and agents [9]. Both APLs that we use, Jason and GOAL, are made for EIS making it very easy to use since BWAPI Bridge also follows EIS. The only problem is that BWAPI Bridge is not a complete system so not all percepts have been implemented, see section 3.1 for more information on this system.

In this project we have extended EIS two make the use RMI between environment and agents possible. The reason we did this is so we can connect multiple clients to the same environment. More on this can be seen in section 3.3.

2.3 Multi-Agent Systems

In this section we will describe what a MAS is, what it can be used for, and how it works. We will describe different kinds of MAS and some important properties a MAS can have.

Definition 2.13 (Multi-Agent System) *A MAS is a system including two or more intelligent agents (see definition 2.1), which are able to interact with an environment and each other in order to achieve individual, mutual, or common goals. In order to do this the individual agents need to have some sort of logical reasoning and communication abilities, as well as knowing what goals they want to achieve and how to do so.* □

Unit name	Description	Cost		
		Minerals	Gas	Supply
SCV	This is the standard worker unit of the Terran race, capable of repairing buildings and machines as well as building buildings and gather resources.	50	0	1
Marine	Standard infantry unit capable of midrange attacks.	50	0	1
Firebat	Heavy armored infantry unit, capable of short range attack.	50	25	1
Medic	Healer infantry unit capable of healing infantry units.	50	25	1
Command Center	Main building of the Terran race capable of training Terran SCVs and for receiving gathered resources from the SCVs. One is owned in the beginning of a match.	400	0	0
Barracks	This is the main infantry unit trainer, capable of training Marines, Firebats and Medics.	150	0	0
Supply Depot	Provides supply for the army, allowing for the creation of more units.	100	0	-8
Refinery	A building for refining the gas from the Vespene Geyser allowing the SCVs to gather it.	100	0	0
Academy	This is a research building, capable of enhancing some of the infantry through upgrades, and allowing the Barracks to train Firebats and Medics.	150	0	0

Table 2.1: A list of the units used in our agents. All units are Terran and all infantry units are capable of simple movement. Not all actions on the buildings and units are implemented.

2.3.1 Types of MAS

We distinguish between MAS consisting of a set of agents all with the same skills, attributes, and reasoning capabilities; and one where the agents have different sets of skills, actions, or reasoning capabilities. We call them homogeneous - and heterogeneous MAS.

Definition 2.14 (Homogeneous MAS) *A MAS (see definition 2.13) where all the intelligent agents have exactly the same skills, attributes, and reasoning skills; i.e., the agents use the exact same implementation code and they are identical.* □

Definition 2.15 (Heterogeneous MAS) *A MAS (see definition 2.13) where the intelligent agents can have different skills, attributes, or reasoning capabilities; e.g., the agents have their individual implementations or differs in other ways. This makes the system much more complex than the homogeneous one.* □

When the type of MAS is determined, we need to understand the structure. To avoid collision between agents and minimize redundant actions, all MASs have the possibility to implement an organization.

Definition 2.16 (Organization in MAS) *An organization in a MAS is a set of rules determining who does what, and when to do so.*

In the classic ACMAS the organization is implicitly defined by the agents interaction with each other, for instance dependencies between agents could be implemented by introducing communication protocols making sure actions are taken in the right order.

One disadvantage of doing it this way is that the system will seem very complex and hard to understand since the same code will describe *what*, *when* and *how* to follow a plan.

By splitting the code into an *organizational layer* and an *agent layer*, it becomes much easier to read and thereby makes it possible for the developer to design even more complex solutions without losing track.

There are a lot of different models used to describe the organizational layer [3], but here we will describe the general principles of such systems.

In Organization Centered MAS (OCMAS) the code is split up such that the agent itself knows *how* to complete goals whereas the organization keeps track of *what* and *when* goals should be done. In general there are three principles an OCMAS should follow[14]:

Principle 1: The organizational layer provides specifications on the behavior of the agents, e.g., what should be done by which roles. It should not describe how an agent achieves this.

Principle 2: Both *reflexive* as well as *goal-based* agents can act in an organization, which means that all mental issues such as *beliefs*, *desires*, *intentions* etc. should be left out and only the expected behavior should be described.

Principle 3: It is possible for the agents in an organization to partition themselves in groups in which they can communicate with each other freely. These groups also work as an extra layer of security since agents in one group do not know anything about the structure of other groups (unless they are also a member of that group).

In our project this organizational layer is implemented using AORTA. See section 2.4 to read more about how AORTA handles the three principles above.

2.3.2 Decision making

When making any decisions in a MAS, it is split into three parts, namely *what* we need to do at the current state, *who* needs to do it, and *how* it is done. How to determine what needs to be done can be seen section 2.1.4. In section 2.3.2.1 we will describe how to use planning to determine who does what. *How* it is done is up to the agent.

2.3.2.1 Coordination

When a MAS needs to figure out who does what it needs a way to ensure that the right agents do each task. This can be done based on the *gain*, as the *gain* or more specifically the cost is based on the current state of each agent, especially any *beliefs* that differs from agent to agent such as position and remaining energy. Then when all gains from all agents are correctly calculated, they need to be gathered so they can be compared. This is often done by sending messages or calculating on each agent.

When all *gains* are gathered they need to be compared, and an optimal solution would be the combination of sub-goals to pursue which accumulate to the largest *gain*. This can be done efficiently with the Hungarian algorithm in $O(A \cdot G^2)$ [1] where A is the number of agents and G is the number of goals available. However, since $O(A \cdot G^2)$ is still fairly heavy, some computational easier solution are often used, even though they are not optimal. These solutions could be algorithms to simulate real world like communication; e.g., an auctioning algorithm where the agents can bid on different *goals* based on their *gain* [4].

2.3.3 Organizational layer

The *organizational layer* is basically an abstract layer specifying the structure and dynamical features of an OCMAS. The purpose of this layer is to make an organization detached from the reasoning of the agents themselves. It is generally designed by introducing some new concepts to the system, e.g., *agents*, *roles* and *groups* as in [14]. Below we have defined these three concepts as well as an extra one for actions, because it is important in order to describe plans from the organizational layer without describing how to accomplish the given plan.

Definition 2.17 (Agent) *An agent is the actual intelligent agent in the MAS. It has a set of goals that it is able to pursue and it will, based on these, play roles and form groups with other agents.* □

Definition 2.18 (Role) *A role is an abstract representation of responsibilities, formed as objectives for the given role, meaning that it is basically a distinction between job positions in the organization. The Roles are then responsible for fulfilling the objectives and an agent can enact multiple roles if so desired.* □

Definition 2.19 (Group) *A group is a set of agents that in some way are following the same goals. It is used to partition the organization to minimize communication. Agents can only communicate with agents in the same group but can be a member of several groups.* □

2.4 AORTA

Now that we understand MASs, this section will explain how we will add an *organizational layer* making both the code and the *plans* more organized. We will use AORTA to do this, e.g., by introducing *dependencies* between different actions.

2.4.1 Concept

The concept of AORTA is to add *organizational reasoning* to already existing intelligent agents by adding an *organizational layer*.

As the *organizational layer* should be clearly distinct from the agents own reasoning AORTA operates only by adding *organizational beliefs* and *options* to the *mental state*, see definition 2.20.

The agents are only affected by the organization when objectives are added to the agents *mental state* as *goals* based on the organization and the *capabilities* of the agent, see section 2.4.3. This is equivalent to injecting additional *intentions* right after the **Filter Function** in the BDI control flow seen in figure 2.1. By adding it this way the agents do not need to change anything when AORTA is added, but will simply get additional *goals* which they can choose to follow for themselves.

Definition 2.20 (Mental state) *The mental state of an agent is the entire set of beliefs, goals, organizational beliefs, capabilities and options that the agent has. In AORTA the different parts of this state is distinguished by a predicate wrapping the rule:*

$$\neg org(a \vee d) \wedge bel(b \wedge c) \rightarrow \neg org(a), \neg org(d), bel(b), bel(c), cap(e)$$

□

2.4.2 Meta Model

The organization of the agents will be simplified by the meta model. In this model we are able to define the **Roles, Objectives, Dependencies, Conditional obligations** and **Rules**.

Roles: The roles in AORTA are following definition 2.18, meaning we are able to assign a set of *objectives* that this *role* can be obligated to achieve. The agent does not need *capabilities* for all the *objectives* assigned, however it must have the *capability* of at least one of the *objectives*. If the agent does not have the *capability* for an *objective* it is responsible for, it can assign it to other agents through *dependencies*. These *roles* are an abstract way of defining and classifying agents. By limiting the enactment of a *role*, the organization can easily define different strategies by allowing or denying different *roles* at different times. *Roles* are denoted:

$$\textit{Role} : \textit{Objectives}.,$$

where *Objectives* is a semicolon separated list of *objectives*. When parsed to the *mental state* the *role* is denoted:

$$\textit{org}(\textit{role}(\textit{Role}, [\textit{Objectives}])),.$$

where *Objectives* are a comma separated list. The message that an agent is enacting a *role* is denoted:

$$\textit{org}(\textit{rea}(\textit{Agent}, \textit{Role})).$$

Objectives: Objectives are the *meta goals* which can be seen as *desires* from the BDI model (see section 2.1.3). If an agent commits to an *objective* it will become a *goal* on the agent. *Plans* are constructed here by making *plan trees* by assigning sub-objectives to *objectives*. How the organization interprets the *plan tree* is up to the reasoning in the AORTA file (see section 2.4.4). *Objectives* are denoted:

$$\textit{Objective} : \textit{SubObjectives}.,$$

where *SubObjectives* are a semicolon separated list of sub-objectives. When parsed to the *mental state* the *objective* is denoted:

$$\textit{org}(\textit{obj}(\textit{Objective}, [\textit{SubObjectives}])),.$$

where *SubObjectives* are a comma separated list.

Dependencies: If an agent has an *objective* that it does not have *capability* to solve by itself it can assign another agent through *dependencies*. We denote this:

$$\textit{Role1} > \textit{Role2} : \textit{Objective},$$

where *Role1* is dependent on *Role2* to do *Objective*. When parsed to the *mental state* the *dependency* is denoted:

$$\textit{org}(\textit{dep}(\textit{Role1}, \textit{Role2}, \textit{Objective})).$$

These *dependencies*, together with sub-objectives, makes it possible to build an advanced *plan tree*.

Conditional obligations: Conditional obligations is a way to obligate an agent to do something based on a certain *condition*. It works by adding an *obligation* to the *mental state* of the agent if the *conditions* are met, and then let the agent do with it what it wants.

$$\text{Role} : \text{Objective} < \text{Deadline} | \text{Condition},,$$

where all agents with the *Role* have an *obligation* to do the *Objective* if the *Condition* is met before the *Deadline*.

If a *Role* violates the *Deadline* a new *option* is generated of the form:

$$\text{viol}(\text{Agent}, \text{Role}, \text{Objective}),,$$

which means that the *Agent* with the *Role* has *violated* the *obligation* to complete *Objective* before the *Deadline*.

If the *condition* is met the *option* will be added to the *mental state* as follows:

$$\text{opt}(\text{obj}(\text{Objective})).$$

Rules: Rules are a way to make prolog functions which can be used in the meta model and AORTA program (see section 2.4.4). They are made for convenience, and to make the meta model and .aorta file more readable. They are written in TuProlog.

2.4.3 Reasoning cycle

On top of the agents normal reasoning cycle, AORTA has its own reasoning cycle which is executed in the beginning of every cycle. This *organizational reasoning* is divided into three parts:

Obligation Check (OC): In this phase the organization checks whether an *obligation* is activated, satisfied, or violated. The *mental state* of the agent is updated accordingly.

Option Generation (OG): This is the phase wherein *options* are generated based on the *mental state* of the agent. The considered organizational aspects are:

Role Enactment/Deactment: An option is generated for *roles* that are possible to enact or deact according to the *mental state* of the agent.

Obligations: If a *condition* in an *obligation* is met, an *option* for the *objective* is generated.

Delegation: An *option* is generated to delegate *objectives* to other agents if it is specified in the *dependency relations* from the meta model.

Information: *Options* are generated to share information with the other agents, e.g., role enactment or knowledge obtained that other roles are dependent on.

Action Execution (AE): Here the agent execute at most one applicable *action* from the **OG** phase. The possible *actions* are to enact/deact a role, commit or drop an *objective*, and send messages to the other agents.

The standard implementation of the reasoning cycle in AORTA is called Linear Strategy and is defined in The AORTA Architecture [3].

2.4.4 AORTA program

This part of the model can be individual for each agent type. This is where the reasoning is implemented and it is here the **AE**-phase of the organization is placed. *Options* that the agent needs to react on can be added as an action rule and the syntax is:

$$option : context \Rightarrow action,$$

where *option* is the generated option the agent should react on if the *context* is true, and the *action* is the action the agent should do.

The order of the action rules defines the priority just like in logic programming in prolog.

2.4.5 Multi-language

A great advantage by using an explicit organization like AORTA is that with an extension the agents can use this plugin across multiple programming languages like, e.g., GOAL and Jason. Our goal is to implement a bridge between GOAL and AORTA and extend the communication in AORTA to use a messaging server distributing messages across different AORTA clients. The bridge to Jason has already been created and more about the process can be read in this article [3].

This extension will make it possible to have groups in AORTA as they are defined in definition 2.19 since each AORTA client would basically form a group. The groups will be formed by a separate MAS in the respective APL, thus making the agents able to communicate directly through its own instance of the APL.

CHAPTER 3

Implementation

In this chapter we will describe and discuss our design choices in the different parts of the system.

We will explain how the implementation of the environment, we will use, works, and how the the agents we have made act in it. We will explain how we extended the environment to support multiple instances of MASs, and the changes needed in the existing systems. Lastly we will explain how we would extend AORTA to allow communication between different AORTA instances.

3.1 BWAPI Bridge

In this section we describe the implementation of the Bridge between APLs and the Brood War API (BWAPI Bridge) implemented as a master project in 2014 on DTU [8].

The project uses the Java Native Interface BWAPI (JNIBWAPI) [15] as the API to SC:BW and is the bridge from JNIBWAPI to any APL based on EIS.

3.1.1 Structure

In order to make the APLs work with SC:BW several systems need to be connected in different ways. For an illustration of the connected system see figure 3.1.

The different parts needed are:

StarCraft: Brood War: This is the standard game as it was made by Blizzard [5].

BWAPI: BWAPI is the Brood War API, which allows for C++ programming of the SC:BW units by *DLL injecting* the game by running it through the BWAPI Injector: Chaoslauncher [7].

JNIBWAPI: This is the java interface for the BWAPI, which allows for java programming to interact with BWAPI [15].

BWAPI Bridge: This is the system that connects to JNIBWAPI and extends EIS allowing APLs to connect to it [8].

3.1.2 Agent interface

In our project we have chosen two APLs, namely GOAL and Jason [13, 6], both of which follow the programming paradigm AOP, and thus follows the BDI model as described in section 2.1.3. This makes BWAPI Bridge responsible for parts of the **Belief Revision Function**, i.e., the knowledge distribution, and the **Action Execution**.

3.1.2.1 Belief Revision Function

The agents in the system receive a limited amount of different percept types which can be seen in Table 3.1 on page 33. Since the agents do not need all *knowledge* they do not receive the same percepts. Which percepts they get is based on which groups they fit into:

All units receive: Idle, IsBeingConstructed, Id, Position, BuildtilePosition, Base, Chokepoint, TotalResources, Unit, Enemy and Friendly percepts.

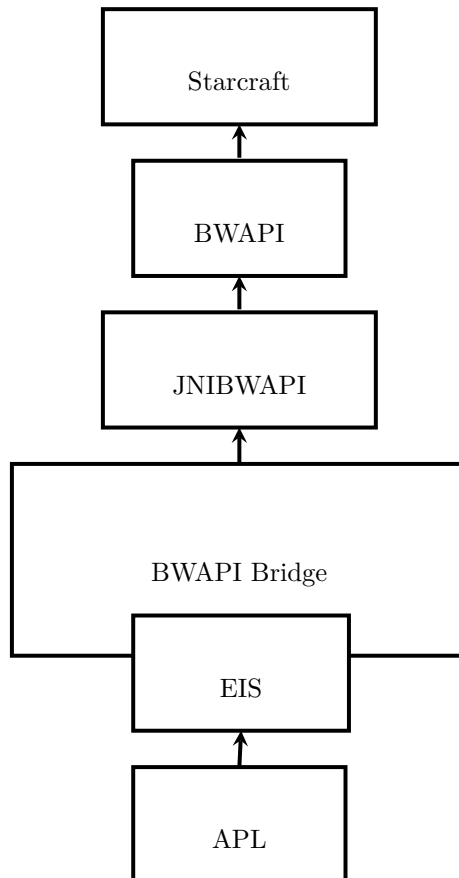


Figure 3.1: An illustration of the systems used in the BWAPI Bridge and how they are connected.

All buildings receive: Minerals, Gas, Supply, QueueSize, BuildUnit percepts.

Attack capable units receive: Attack percepts.

All Worker units receive: Minerals, Gas, Supply, IsConstructing, Gathering, Carrying, VespeNeGeyser, ConstructionSite and WorkerActivity percepts.

Terran Command Center receive: IdleWorker and WorkerActivity percepts.

Terran Marine receive: Stimmed percepts.

All units receive all percepts from all the categories into which they fit; e.g., a Marine would get percepts from the categories: **All Units**, **Attack Capable**, and **Terran Marine**.

3.1.2.2 Action Execution

BWAPI Bridge has a set of *actions* that the agents can use if they have the corresponding *capabilities*. This can be seen in Table 3.2 on page 34. What an agent is *capable* of is determined directly in BWAPI. A description of possible *actions* each unit can use can be seen below:

SCV: Attack, Build, Gather, Move, Stop.

Marine: Attack, Move, Stop, Use(stim).

Firebat: Attack, Move, Stop, Use(stim).

Medic: Move, Stop, Use(heal).

Command Center: Train, Stop.

Barracks: Train, Stop.

3.2 StarCraft agents

Here we will discuss our implementation of the different MAS we have implemented in this project. We will discuss the different aspects of the agents and the different strategies they use as well as the implemented organization.

Percept:	Description:
Attacking	What a certain unit is currently attacking.
Base	This is locations on the map where outposts could be established (near mineral fields).
BuildTilePosition	Own position in Build Tiles.
BuildUnit	Contains the ID of the builder unit which is currently constructing the building.
Carrying	That the unit is currently carrying some sort of resource.
Chokepoint	The location of chokepoints. These are the points on the map where none-flying units needs to pass to get to the other side.
Constructing	That unit is currently building a building.
ConstructionSite	A valid Building Tile for constructing a Terran Command Center.
Enemy	An enemy with ID, type and position.
Friendly	A friendly unit with ID, type and position.
GameStart	That the game is started.
Gas	Remaining vespene gas.
Gathering	What a certain unit is currently gathering.
HitPoint	Remaining hitpoint of the unit.
Id	Own ID.
Idle	That a certain unit is currently idle.
IdleWorker	Contains the ID of an idle worker unit.
IsBeingConstructed	That the unit itself is under construction or training.
Map	This is the width and hight of the map.
MineralField	Mineral field with ID and position.
Minerals	Remaining minerals.
Position	Own position on the map in Walk Tiles.
Queue	What actions are queued on the buildings queue.
QueueSize	The current queue size on the buildings queue.
Stimmed	Whether the unit has used a stimpack.
Supply	Currently used and max supply.
TotalGas	Contains the total amount of vespene gas gathered throughout the game.
TotalMinerals	Contains the total amount of minerals gathered throughout the game.
TotalResources	Contains the current and total gathered of all three resource types.
Unit	The current amount of a unit type.
UnitType	Own unit type.
VespeneGeyser	Vespene geyser with ID and location.
WorkerActivity	This is what the different workers are currently doing.

Table 3.1: Table of the percepts given by BWAPI Bridge.

Action	Description
Attack	Order a unit to attack on either an ID or a position.
Build	Build a certain building type at a certain position.
Gather	Gather either vespene or minerals, determined from the type of the map element based on the specified ID.
Move	Move the unit to a certain position.
Train	Train a certain unit type.
Stop	Stop the unit, no matter what it is currently doing.
Use	Use is used to utilize abilities, can be used either with a position, an ID or nothing, based on the nature of the ability.

Table 3.2: A table of the possible *actions* through BWAPI Bridge.

3.2.1 Knowledge

We have the relation between *knowledge* and *belief* that if we *know* something to be true we cannot *believe* the opposite and vice versa:

$$\begin{aligned}\mathcal{K}_i\varphi &\rightarrow \neg\mathcal{B}_i\neg\varphi \\ \mathcal{B}_i\varphi &\rightarrow \neg\mathcal{K}_i\neg\varphi\end{aligned}$$

How we distinguish between *knowledge* and *belief* in the *mental state* is based on the dynamic of the perceived information.

Belief: Everything that is perceived from the environment is, at the point that it is sent from BWAPI, true, which the agent knows. However at the time our agents process the information, every dynamic part of the perceivable environment might have changed, and hence it must be handled as *beliefs*. This means that we often check if our *beliefs* from the time an action is started still match the ones given by the current percepts. Our understanding of percepts are formally denoted:

$$M, s, i \models \mathcal{K}_i\text{percept}(\varphi) \rightarrow \mathcal{K}_i\mathcal{P}(\varphi)$$

$$M, s, i \models \mathcal{K}_i\text{percept}(\varphi) \rightarrow \mathcal{B}_i(\varphi),$$

where M is the world, s is the state and i is an agent. All entities in the system receive not only what is within their own range of vision, but what is within the entire team's range of vision. This means that all percepts in the system are, at the time the percepts are sent, *distributed knowledge* between all units in SC:BW (see definition 2.7). However, as

the *knowledge* is collected in BWAPI Bridge and is sent with delays and at different points in time for each agent, it is no longer neither *distributed* nor *knowledge* when the agent gets it; it is individual *beliefs*.

The information of other agents' *actions* are often made by conclusions drawn using relations from *knowledge* with the related information from *beliefs*, which means all information about the behavior and mental state of the other agents are *beliefs*.

Knowledge: The only parts of the *mental state* we handle as *knowledge* is methods coded into the agent and the static parts of the perceivable environment. The methods on the agents represent relations which are known to be true; e.g., if an enemy unit has the type "Terran Command Center" then we *know* that it is a building. The static parts of the map can be map width or position of a Vespene Geyser. Formally denoted:

$$M, s, i \models \mathcal{K}_i \text{percept}(\varphi) \rightarrow \mathcal{K}_i A\varphi,$$

where M is the world, s is the state, i is an agent, and φ is a static part of the environment.

All the *knowledge* we have is assumed to be *common knowledge* in the group that needs it; e.g., it is *common knowledge* among SCVs how to calculate distances. This is however only reflected in the system by the fact that our agents never share any conclusions drawn based on their own *knowledge*. They simply assume that the other agents have gotten similar percepts and based on these draw the same conclusions and react accordingly, see example 3.1.

Example 3.1 (Failing to build a building) *SCV1 has the goal of building a Supply Depot. He believes the resources are sufficient, but by measuring all the distances he sees that he is not currently the closest to the construction site, and thus do not act on this goal. It does this since SCV1 knows, that if all other SCVs get percepts similar to the ones it got itself, then one of them must be closest and thus conclude that it should do the task of constructing.*

Meanwhile SCV2 and SCV3 gets similar percepts, both of them are closer to the construction site than SCV1, but as the distance is the same both of them concludes that they are closest and commits to the goal. This results in two SCVs trying to build the same building at the same construction site and thus they get in each others way.

This means that the building was never built, making the agents try again. □

3.2.1.1 Planning

The previously mentioned calculations of *gain* in equation 2.1 are not suited for our *continuous environment* since the agents perceive at different times and since they are required to react fast, they cannot wait for the *gain* from the other agents or do the calculations themselves before deciding what to do. Even if we were to wait for the results for all agents, the result would no longer be correct resulting in even more wasted time without getting better results. To accommodate this we have made a simplification where we filter the *goals* based on the *value* and then the *cost* instead of a combined calculation.

In our MASs the planning part of the control flow as seen in figure 2.1 is done by having a prioritized list of *plan libraries*, meaning that the agent will simply commit to the first plan where the context holds. This corresponds roughly to the *value* of a *plan* as described in section 2.1.4 whereas the *cost* needs to be taken into consideration by the programmer in the context of the *plan*. An example of calculating the *cost* of a *plan* can be seen in example 3.2.

Example 3.2 (Cost calculation) *Consider a plan where an SCV is to build a building at a specific position. By making sure the SCV only commits to the plan iff. it believes it is the closest SCV to the building position, only the SCV with the smallest cost, i.e., believed least distance, will try to follow the plan.*

3.2.2 Aspects of the simulation

The different aspects of the simulation that we have taken into consideration is the small autonomous tasks which the agents need to consider. We have chosen to apply only a few to simplify the system and to show how they interact with AORTA. The aspects we consider are:

Resource Gathering: This deals with the gathering of resources. If an agent decides to gather, it needs to decide *what* and *where* to gather.

Build Building: This concerns whether a building can be built, *where* to build it and whether a certain agent should try to build it or not.

Attack Enemy: This concerns whether an agent should attack an enemy, and which enemy to attack if more are present.

Train Unit: This concerns whether a unit can be trained, and if the agent should try to do so.

Scouting the map: This concerns whether a unit should scout the map to gain more knowledge of the environment.

Heal friendly unit: This concerns whether a unit should try to heal a friendly unit.

Most of the handling of these aspects are done the same way on all the systems we have made, and only the coordination differs from system to system. We have chosen to solve these aspects using a number of goals distributed on the relevant agents as described in Table 3.3 on page 42. Below we will show how the different goals will be achieved if the agent chooses to do so:

gather: This *goal* is used to solve the aspect of Resource Gathering and is done by figuring out whether mineral or gas should be gathered. Gas should be gathered if there is currently less than three other agents doing so and we are the closest agent not currently gathering gas. If this is not true the agent will try to gather mineral from the nearest mineral patch available.

build(Building): This *goal* is used to solve the aspect of building a Building. It is done by first finding the building to build according to the conditions (*objectives*, *obligations* and *dependencies* when using AORTA) implemented on the agent.

When the building has been found, the location is found simply by taking the nearest Building Tile to the Command Center ($\pm\delta$ to make room for agents to move). Lastly the agent determines if it should build the building by seeing if it is the closest to the location.

train(Unit, Amount): This *goal* is used to solve the aspect of Training Units. We train a Unit if we have less than the desired Amount and currently has enough supply and resources to train the unit. Agents that have this *goal* also uses another *goal* called `maintainMentalState` to add this *goal* again if some of the units die.

defend: This *goal* is used in cooperation with `charge` to solve the aspect of Attacking the Enemy. This *goal* will simply make the agent attack enemy units if they are closer than 125 Building Tiles to the Command Center.

charge: This *goal* is used in cooperation with `defend` to solve the aspect of Attacking the Enemy. This *goal* will first try to attack the nearest visible enemy unit, if one is not present it will move to the position of the last spotted enemy base.

spot(X): This *goal* is used in cooperation with `scouting` to solve the aspect of Scouting the map. This *goal* will simply make the agent look for X. We use it to find Vespene Geysers and the Enemy Base.

scouting: This *goal* is used in cooperation with `spot(X)` to solve the aspect of scouting the map by making the agent move around the map in order to try to get new *knowledge* about the environment. The way we did it is a very random scouting mechanism where the agent moves to a random position on the map until the Enemy Base has been found. This could easily be optimized by only moving to unknown areas of the map each time.

For the implementation of the *plans* to achieve these *goals* we refer to the implementations of the agents in appendix C.2.

3.2.3 Implicit organization

As we implemented our agents in GOAL, we realized that some sort of organization (see definition 2.16) would be required in order to prevent all agents of the same type to do the same action at the same time; e.g., all SCVs would try to build a building at the same location at the same time resulting in errors for all of them: ultimately meaning that the building was not built at all. We tried implementing a communication based organization at first, but realized some issues with the communication and therefore we tried an adjusted version applying less communication and more individual reasoning. These are examples of *implicit organizations* as the *organizational reasoning* is distributed across the agents. They are implemented directly in the *conditions* for the individual *actions* since they react on information received through internal messages from other agents. For a description highlighting the key coordinational differences see section 3.2.3.1 and 3.2.3.2.

3.2.3.1 First iteration

Our initial implementation in GOAL was communication based, but was never fully completed due to issues with the communication; it did teach us a lot about GOAL. Below we will describe the three main differences from the previously defined behaviour for the different aspects in section 3.2.2:

Resource management: We tried to make a slightly more advanced resource management than just build or train something if the resources are available in the beginning of an action. We made a new agent which simply received and granted resource requests. If the request was granted, the player reserved the resources for the agent, such that a SCV would never

initiate a building only to find insufficient funds as it arrived at the location for the coming building. However, it turned out that all our resources ended up being allocated to units, who never used it, due to communication errors and the *non-determinism* of the environment.

Mineral Gathering: The SCVs fallback action, in case there was nothing else to do, was to gather minerals. By the simple reasoning on the *belief* of carrying that $justCarried \wedge \neg carrying \rightarrow justDelivered$ we tried to make the agents stop and recalculate closest mineral field after every delivery of minerals to the Command Center. However, they got the percepts quite late, meaning that sometimes they stopped in the middle of a gathering, or not at all, which left this approach doing more harm than good.

Build Building: The Command Center tells one specific SCV to build a certain building and where to build it. Then the SCV pursue this *action* until the building itself says that it is done which in turn makes the SCV tell the Command Center that the building is completed and hereafter continues to do something else. However, as this *action* can fail and the agents do not always listen, the building was not always completed and the agents did not always tell the Command Center about the outcome.

To get assurance that a building would be built at least once, we tried to implement a time based failsafe such that the Command Center would assign a new SCV to the task if the first one never responded. Furthermore, after a given amount of time the SCV would pursue a new goal no matter if the building was built or not. However, the messages were inconsistent in arriving, and often the Command Center never got the message and made another SCV build another.

Very little of this system managed to be coordinated, as messages did not always get to the recipient. This meant that a time based failsafe triggered almost every time when used, and without it the system did not do what it was supposed to. For this reason we tried to make one with less communication, i.e., avoid time sensitive communication.

3.2.3.2 Second iteration

The second iteration had a lot more calculations, as all time-critical information had to be calculated for every agent, instead of only one place and hereafter sent to the others.

Mineral Gathering: The SCVs simply take the mineral field closest to the SCVs location and the beginning of the gathering. We have made recalcu-

lation like the one from the first iteration, but as it does no longer call the `stop` method they use the StarCraft gathering method until they change their goal to something else and back again. This means that they will often keep gathering in one direction away from the Command Center even though closer mineral patches are available.

Vespene Gathering: We tried to make the Refinery responsible for this, however, it was not a valid entity in BWAPI Bridge thus we assigned it to the Command Center instead. The Command Center assigns an additional SCV to the task if less than three is already gathering. It decides which SCV should gather vespene by calculating the closest ones to the Terran Refinery and send a message to them telling them that they should do it. As this is not time sensitive and whether three or four SCVs gather vespene is not important, this solution works even though it is not correct.

Build Building: The SCVs decides which building, if any, should be built and individually calculates the right building spot, and then they calculate whether they are the closest to that spot, and if they are, they try to build it. This often makes more than one SCV attempt to construct the building, causing them to stand in the way of each other and the building is then not build.

Resource Management: As communication is limited and training units and building buildings need to happen as fast as possible, the resource management is simplified. We simply check the current amount of resources directly in the environment and pursue the action if the resources are sufficient. This however often results in insufficient resources; e.g., an SCV initiates the building of a building, but before arriving at the building location the barracks has initiated the training of a marine, hereby spending the resources. This does however only cause the unit, who ends up not building or training anything, to loose a single iteration of reasoning and at most a few seconds in the game.

This method relies a lot on the environment stopping them if: they do something they are not supposed to, do not have enough resources for, or are otherwise not able to do.

3.2.4 Explicit organization

In this sub-section we will explain how we have been implementing our organization according to the rules discussed in section 2.4 in order to show how to organize heterogeneous agents across multiple APLs.

Our original plan was to create an advanced organization including *plans* for building an entire base and complicated attack strategies. After initial testing of both Jason and AORTA we realized that the systems are not optimized enough to be able to accomplish this due to the fairly heavy computations needed and the added delay through all the interfaces, i.e., BWAPI Bridge, JNIBWAPI and BWAPI.

Instead we have made some more simplistic *plans* which are basically a simplification of the *explicit organization* already present in the game, e.g., a Terran Barracks is dependent on a Terran SCV building an Academy in order to be able to train Firebats. Furthermore we limited the number of agents to a single one of each type. The *roles*, *objectives* and *dependencies* can be seen in Table 3.3 on page 42.

The main idea about this is that we can show that the base will be constructed in the correct order while agents only receive goals that makes sense according to the organization. Furthermore we will see that the infantry units, with the *role attacker*, only charges the enemy when the enemy base has been spotted by a *scout*.

3.2.4.1 Reasoning

We have chosen to set a *context* in AORTA such that agents take *roles* if they have *capabilities* of at least one of the *objectives*, the *roles* are responsible for and it is not explicitly ignored on the agent. When an agent has a *role*, we will utilize the *option* to send a message to other agents notifying them of this.

We have chosen to design our *plan tree* as *objectives* and *sub-objectives* using *dependency relations* for communication between dependent roles. By making the agent delegate not yet achieved *sub-objectives* to agents playing the *roles* that can help solve them. We have the functionality of our **all** nodes from section 2.1.4. An agent has the *option* to commit to an *objective* iff. it has the *capabilities* to do so, it is not ignored and all *sub-objectives* has been solved. In order to make it possible for the agent to know this, *dependency relations* are defined for all *sub-objectives* that the *role* can not solve itself as this gives the dependee the option to send a message back to dependent *roles* when an achievement has been solved.

Lastly the agent will react on the *option* to deact a *role* if it already has the *role*.

The implementation of the reasoning described above can be seen in listing C.2.1.

Role	Objectives	Dependencies	Dependency Objective
healer	heal	-	-
gatherer	gather	-	-
scout	scouting	-	-
commander	trainArmy	builder armyTrainer	build("Terran Barracks") train("Terran SCV", 5) train("Terran Marine", 1) train("Terran Firebat", 1) train("Terran Medic", 1)
builder	build(Building) build("Terran Refinery")	- scout	- spot("Vespene geyser")
attacker	defend charge	- scout commander	- spot("Enemy Base") trainArmy
armyTrainer	train(Unit,Count) train("Terran Medic",Count) train("Terran Firebat",Count)	- builder builder	- build("Terran Academy") build("Terran Refinery") build("Terran Academy") build("Terran Refinery")

Table 3.3: Table showing the *objectives* (see section 3.2.4.3) each *role* is responsible for. If the agent cannot achieve the *objective* by itself a *dependency relation* can also be seen. A more detailed description of the *dependencies* will be discussed in section 3.2.4.4.

Unit Name	Capabilities	Roles
SCV	build(Building) build(Building,X,Y) gather	gatherer builder
Marine	spot("Vespene Geyser") spot("EnemyBase") scouting charge defend move	scout attacker
Firebat	charge defend move	attacker
Medic	matchUp heal	healer
Command Center	trainArmy train(Unit,Amount) maintainMentalState	commander armyTrainer
Barracks	train(Unit,Amount) maintainMentalState	armyTrainer
Supply Depot	-	-
Terran Refinery	-	-
Academy	-	-

Table 3.4: This table shows the *capabilities* of our agents for the different units described in Table 2.1 on page 20 and lists their possible *roles*.

3.2.4.2 Roles

Each *role* is associated with one or more *objectives* which they are responsible for gets done. Whether they do it themselves or delegate it will be defined later in the meta model. The *roles* can be seen in listing C.2.1.

3.2.4.3 Objectives

The *objectives* are used in cooperation with *dependencies* and *obligations* to form plan trees. The complete *plan tree* for our organization can be seen in figure 3.2. The *objectives* in Table 3.3 on page 42 are implemented in the Meta Model as shown in listing C.2.1.

Below we will describe when the different *objectives* are used. We refer to section 3.2.2 for descriptions of how these *objectives* are achieved if the agents pursue them as *goals*. We use the syntax of the mental state and AORTA for the different terms in order to make it easier to relate to.

scouting: If an agent has the *role* `scout` it will commit to this *objective* until the enemy base has been spotted. Formally denoted:

$$M, s \models \text{org}(\text{rea}(\text{scout})) \rightarrow \text{commit}(\text{scouting}) \mathcal{U} \text{bel}(\text{spot}(\text{"Enemy Base"}))$$

spot(X): If an agent has the *role* `scout` it will commit to this *objective* until `X` has been spotted; this is basically the way GOAL works with *achievement goals*. Formally denoted:

$$M, s \models \text{org}(\text{rea}(\text{scout})) \rightarrow \text{commit}(\text{spot}(X)) \mathcal{U} \text{bel}(\text{spot}(X))$$

gather: If an agent has the *role* `gatherer` it will commit to this *objective* at all times since we use it as a fallback *action* if there is nothing else to do. Formally denoted:

$$M, s \models \text{org}(\text{rea}(\text{gatherer})) \wedge \neg \text{bel}(\text{busy}) \rightarrow \text{commit}(\text{gather})$$

trainArmy: If an agent has the *role* `commander` it will commit to this *objective* when all *sub-objectives* has been achieved. Formally denoted:

$$M, s \models \text{org}(\text{rea}(\text{commander})) \wedge \text{bel}(\text{SubObjectives}) \rightarrow \text{commit}(\text{trainArmy}),$$

where `bel(SubObjectives)` denotes that the agent *believes* that all *sub-objectives* has been solved.

train(Unit, Amount): If an agent has the *role* `armyTrainer` it will commit to this *objective* if at any point in the game there is less than `Amount Unit` present and all *sub-objectives* has been achieved. Formally denoted:

$$M, s \models \text{org}(\text{rea}(\text{armyTrainer})) \wedge \text{bel}(\text{SubObjectives}) \\ \rightarrow \text{commit}(\text{train}(\text{Unit}, \text{Amount})) \mathcal{U} \text{bel}(\text{unit}(\text{Unit}, \text{Amount})),$$

where `bel(SubObjectives)` denotes that the agent *believes* that all *sub-objectives* has been solved.

build(Building): If an agent has the *role* `builder` it will commit to this *objective* if it is time to build `Building` according to the *plan tree* and if all *sub-objectives* has been solved. Formally denoted:

$$M, s \models \text{org}(\text{rea}(\text{builder})) \wedge \text{bel}(\text{SubObjectives}) \\ \rightarrow \text{commit}(\text{build}(\text{Building})),$$

where `bel(SubObjectives)` denotes that the agent *believes* that all *sub-objectives* has been solved.

defend: If an agent has the *role* `attacker` it will commit to this *objective* until we have trained an army and the `scouts` have found the enemy base. Formally denoted:

$$M, s \models \text{org}(\text{rea}(\text{attacker})) \\ \rightarrow \text{commit}(\text{defend}) \mathcal{U} \text{bel}(\text{trainArmy} \wedge \text{spot}(\text{"Enemy Base"}))$$

charge: If an agent has the *role* `attacker` it will commit to this *objective* when the two *sub-objectives*, i.e., train an army and find the enemy base, has been achieved. This means that we basically switch from a defend strategy to a more aggressive one. Formally denoted:

$$M, s \models \text{org}(\text{rea}(\text{attacker})) \wedge \text{bel}(\text{SubObjectives}) \rightarrow \mathcal{A}\text{commit}(\text{charge}),$$

where `bel(SubObjectives)` denotes that the agent *believes* that all *sub-objectives* has been solved.

heal: If an agent has the *role* `healer` it will commit to this *objective* throughout the whole game since we need to protect our units. Formally denoted:

$$M, s \models \text{org}(\text{rea}(\text{healer})) \rightarrow \mathcal{A}\text{commit}(\text{heal})$$

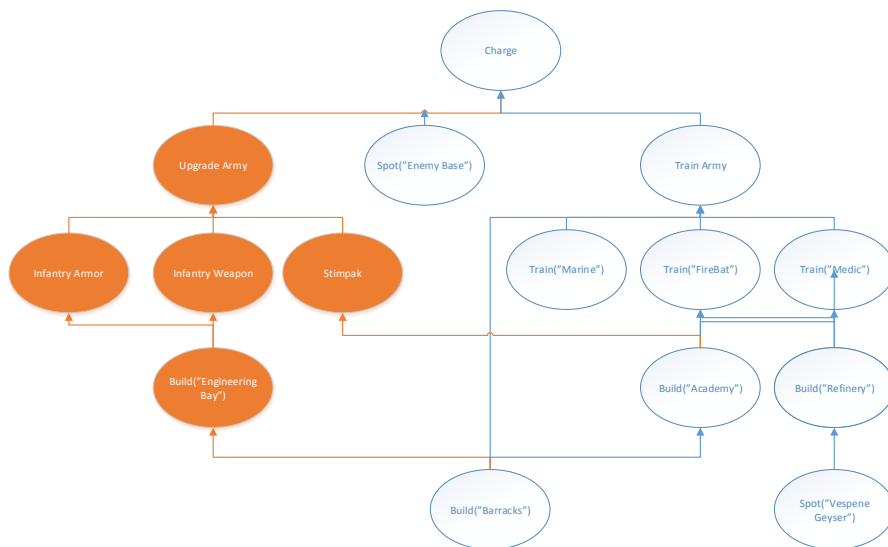


Figure 3.2: Illustration of the *plan tree* we have used without environmental implicit *dependencies*. The blue part indicates the part we implemented and the red indicates the upgrade part of the *plan tree* which was not implemented.

3.2.4.4 Dependencies

All *dependencies* in our system can be derived from the *roles*, *capabilities*, and the *plan tree* formed by *objectives* and *sub-objectives*. To make AORTA generate the right *options* for the agents to communicate and delegate *objectives*, the *dependency relations* still needs to be implemented in the meta model as can be seen in listing C.2.1.

3.2.4.5 Obligations

As discussed in section 2.4 the *obligations* are used to describe who has an *obligation* to follow a *goal* if they meet certain *conditions*. Furthermore a *violation criteria* can be defined. In SC:BW this *deadline* does not really help us that much, since it wont help the system in any way to punish an agent if it does not do something before a *deadline*. Most *objectives* need to be done no matter how late it is, which is why we have chosen not to use the *deadlines* in this project.

Most of the behaviour of the agents are in the *plan tree* formed by the *objectives*, *sub-objectives*, *dependencies*, and *obligations*. The agents will communicate with each other to achieve more complex plans. We use the *obligations* as a way to make sure that a given *role* initiates the execution of a *plan tree*, by *obligating* it to the *root-objective* and let it distribute the remaining *objectives*; e.g., when the `commander` starts the `trainArmy` *objective* he will delegate all *sub-objectives* to the `armyTrainer` who in turn will delegate *sub-objectives* to the `builder` in order to train specific units, who then delegates to the `scout`. See listing C.2.1 for the implementation.

3.3 RMI BWAPI Bridge

To get more than one MAS to connect to the BWAPI Bridge, we have extended it using Remote Method Invocation (RMI) which will make it possible to connect multiple implementations of agents to the same instance of the environment. An example on how we want the information flow to be implemented can be seen in example 3.3. The source code for this system is called EISBW-remote and can be found in appendix C.1.

3.3.1 Structure

A figure of the client/server implementation of the BWAPI Bridge can be seen in figure 3.3. Since we need to connect several implementations to the same instance of the environment we need the server to run a game of SC:BW and let the clients connect through RMI. This means that the environment that each of the implementations use will not be the same, but a *client representation* that can connect to the actual environment on the server.

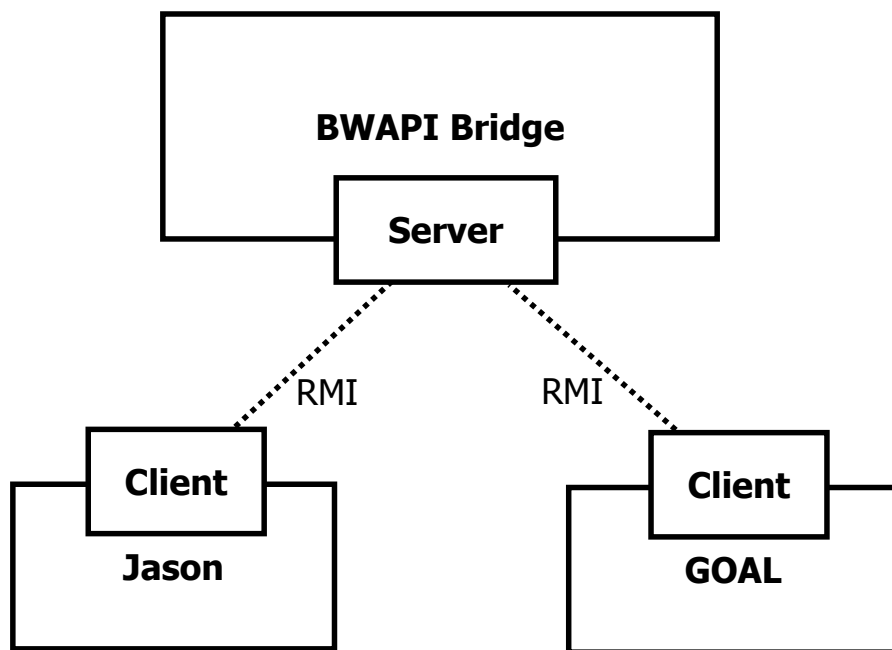


Figure 3.3: Illustration of the design of the client/server implementation of the BWAPI Bridge.

Both the server and the client are based on a remote implementation of EIS called EIS-remote, found on EIS' github[9], as both BWABI Bridge and the APLs are dependent on EIS. The implementation was not finished and we have refactored it to use parts of the implementation from `EIDefaultImpl` by extending it and overwriting methods that we needed to change due to the communication through RMI. The class diagrams of the implementation can be seen in figure 3.4 and 3.5.

3.3.2 Requirements

In order to make the RMI meet the requirements of EIS there are several methods that need to be linked from the `Client` to the `Server`, e.g., *getAllPerceptsFromEntity* and *performEntityAction*. We needed an APL using our modified EIS in order to make debugging statements, to figure out which methods were used directly. We did this in GOAL by compiling the GOAL-core using our EIS,

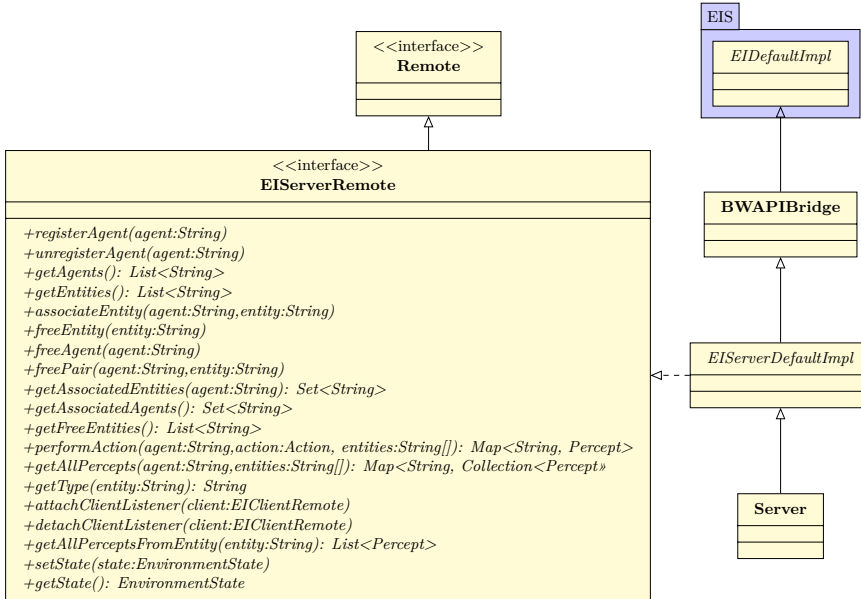


Figure 3.4: Class diagram showing how the server side of the implementation has been designed. All methods specified the **EIServerRemote** interface is the methods that can be activated remotely. The EIS package is the one we have modified.

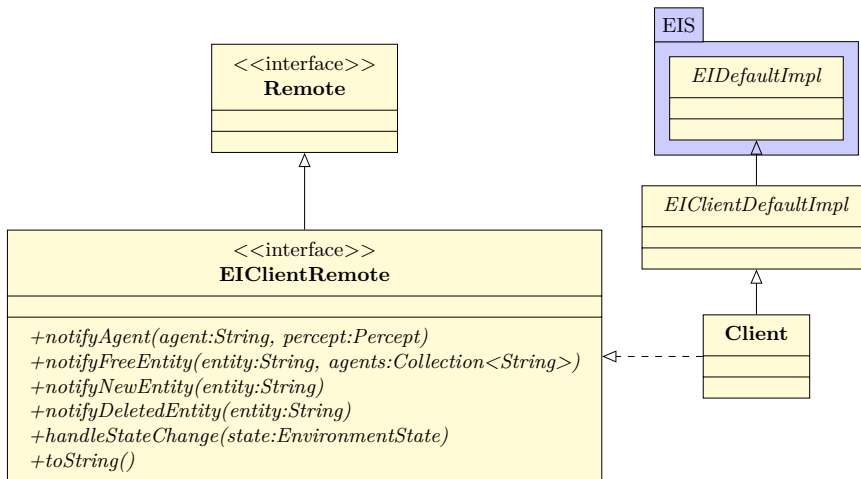


Figure 3.5: Class diagram showing how the client side of the implementation has been designed. All methods specified the **EIClientRemote** interface is the methods that can be activated remotely. The EIS package is the one we have modified.

the process of this can be seen in appendix A.3.

In order to make the required methods remotely accessible we had to change the *visibility* of some of the variables in `EIDefaultImpl` in the EIS project from *private* to *protected*, since this is necessary in order to give the server access to these variables. This modified EIS is used on the server side, meaning that BWAPI Bridge has to be rebuilt using this more open version of EIS. We had to change the *visibility* of the following variables from *private* to *protected* in order to access them from the server:

entities: A list of the entities which currently exists in the environment.

agentsToEntities: A map to see which entities are connected to which agents.

registeredAgents: A list of the agent connected to the environment.

entitiesToType: A map of which type each agent is.

Example 3.3 *As an example we can take a look at how an agent will get his percepts in this new implementation. `getAllPerceptsFromEntity` will be called on the client which will simply pass on the request to the server using RMI. The Server will then fetch the percepts through the BWAPI Bridge and return them all the way back to the Client.* □

3.3.2.1 Bookkeeping

When making the system remote we have to think about which entities connects to which agents and which listeners needs to be where. This means some bookkeeping is required and some decisions regarding the placement of the information needs to be made.

As we need our server to extend the BWAPI Bridge and we need the client to implement EIS; both sides end up implementing EIS meaning that some data structures needs to be kept on both sides; e.g., agents are handled on the client since they are created by the APL, whereas entities are handled on the server since they are created by the environment. The mapping between the agents and the entities are handled on both sides, as the `agentsToEntities` map is included in EIS. However, the actual agent to entity association is initially called at the client side, which means the client needs to call the server to make it associate the agent and entity.

3.3.2.2 Remote objects

Some objects cannot directly be retrieved through a RMI, as they need to be *serializable* and instantiated as a *remote object* or they will be returned as they were instantiated. In EIS the retrieval of stored information happens in the entity management which means we need some *remote object* to handle the entities.

We have chosen to make a *remote object* called `ClientEventHandler` which is a *hash map* linking each client to a *queue* of events which then contains the entity. The client will then regularly access this *queue* and handle the retrieved events.

3.4 Communication between AORTA instances

In this section we will start analyzing different possibilities of connecting multiple MAS through AORTA. Then we will describe the result of our implementation using RMI. The source for this system is in the *framework* for AORTA in package *framework.aorta.remote* and we also refactored parts of the system as we will describe in this section. The source can be found in appendix C.1.

3.4.1 Analysis

When having different MAS connected to the same environment all using AORTA, we need a way to make them acknowledge the agents of the other system and use them in the organization. As we have separate systems with different instances of AORTA, we need some sort of communication between them.

3.4.1.1 Communication method

The communication is possible to do through EIS using percepts as communication, by simply adding the information as a percept on the RMI-BWAPI server we have made and let the server distribute the information as percepts to the agents. However, this would require additional methods to be implemented on every environment the remote AORTA should use, which opposes the concept of AORTA.

Instead we have chosen to implement a new RMI message server for handling of the agents' *organizational communication*, where the server has a mapping between agent names and their individual message queue in a *remote object* as in section 3.3.2.2.

3.4.1.2 Knowledge sharing

The knowledge we need to share for AORTA to be able to complete its reasoning cycle is the agent names and all the *organizational related messages* the agents are currently sending to each other.

All agents need to send and receive through this new messaging server instead of the old method which sent messages through the APLs' internal communication method. This will also make it possible for the agents to use the internal messaging strictly for sharing of *knowledge* between the agents that are part of the group.

3.4.2 Result

We have made the use of the remote methods optional as they are activated by the flag `msgServer` in the `.mas2j` file. This sub-section describes what we did to make AORTA organize through the message server. When implementing this extension we had to implement two new classes and an interface for the `AortaMsgServer`. The classes we had to implement were:

MsgQueueHandlerRemote: This is the interface of our *remote object* which is needed in order to make certain methods *remotely accessible*.

MsgQueueHandler: This is the implementation of the earlier mentioned interface which handles the hash map of agent names to message queues. Furthermore we have added a method (`addAortaAgent(aortaAgentName)`) which adds new AORTA agent names to the system; this method will imitate communication between agents by adding a message to all agents notifying them of the name of the new agent using the message form: `org(aortaAgent(AgentName))`.

AortaMsgServer: This is the actual object that each `AortaAgent` will create. The constructor of this class will then either create a new server and *remote object* (`MsgQueueHandler`) if a server does not already exist, otherwise connect to the existing ones. This means that it will be working as the interface between all the agents and the *remote object* `MsgQueueHandler`.

To use this extended message handling we had to refactor several files in the existing AORTA source code to use `AortaMsgServer`'s added functionality. We start describing all the changes needed in the original source code for AORTA, in order to send messages using the message server:

OrganizationalMessage: In order to send *organizational messages* over the *remote object* it needs to be `serializable`. Furthermore as we no longer use the APL's internal messaging system, we do not need the `IncomingOrganizationalMessage`; however, the field *sender* in this class is needed and we had to add it as a field in the message sent through the `AortaMsgServer`. We have done this by making a new abstract extension of `OrganizationalMessage` called `OrganizationalMessageWithSender` and made `IncomingOrganizationalMessage` as well as the new `AortaMsgServerOrganizationalMessage` extend it, the last one being the one used for the new `AortaMsgServer` as both the *outgoing* and *incoming* message.

Aorta: We use the already existing method for adding new agents (`addAgent(agentName)`) to add a new entry for the *hash map* on the `AortaMsgServer` using the agents name, this is done by calling the method `addAortaAgent` on the *remote object* `MsgQueueHandler`.

AortaAgent: This class now contains an instance of `AortaMsgServer` used for the message handling. Instead of using the internal messaging of the APL to send a message to another agent, it now adds the message to the respective *queue* in the *remote hash map* using the method `sendMessage(msg)`.

SendAction/SendOnceAction: All send functions use the `AortaMsgServerOrganizationalMessage` instead of the `OutgoingOrganizationalMessage`.

In order to receive messages from the message server instead of the APL's internal message system we had to change the following:

AgentState/MessageFunction: As we removed the `IncomingOrganizationalMessage` we had to refactor both `insertMessage` and `process`, from `AgentState` and `MessageFunction` respectively, to use `AortaMsgServerOrganizationalMessage` instead.

Check: Instead of receiving messages through the `ExternalAgent` we use `AortaMsgServer` to get the *message queue* corresponding to the `AgentName`. Furthermore, upon receiving this message we check whether the message declares the existence of a new agent or not, and if it does we add `aortaAgent(Name)` to the *organizational mental state* of the agent.

An illustration of the entire system can be seen in figure 3.6.

3.5 Implementation discoveries

While implementing the system we have come across some noteworthy discoveries which we want to highlight in this section:

Building location: The building locations that are received from BWAPI are not consistent with the way Jason and GOAL handles build actions, i.e., BWAPI returns the *center* of the building but the APLs use the *lower left corner* of the building. This was primarily a problem when trying to build a Terran Refinery since it needs to be exactly on the Vespeene Geyser.

The way we found this error was by moving around a Marine while outputting its position as well as the building location returned by BWAPI. In the end we found that we needed to add $(-2, -1)$ to the position returned from BWAPI for a Terran Refinery, this vector is not the same for all buildings since it is dependent on the building size. We only fixed this on the Refinery since it is not a problem on the other buildings as the position is calculated using a Terran Command Center (the largest building) meaning that it will always be possible to build every building smaller than a Terran Command Center even though this error is there.

Building/walk tiles inconsistency: The percepts in the BWAPI bridge are not very consistent in using walk/building tiles. We have changed some percepts to accommodate this by either sending both positions or adding an extra percept that returns the other position.

Java internal actions: When working with Jason it is possible to use internal actions in Java. Since this is not a possibility in GOAL we added a percept to BWAPI that returns possible construction sites (`constructionSite(BX, BY)`). The generation of this percept is quite heavy since we need to ask the API if it is possible to build on a location for all visible *BuildTiles*.

After working with our addition to BWAPI, namely BWAPI-remote, it is also not possible to use internal actions in Jason since it is not connected to the server directly but has the client in between which means there is no direct contact with BWAPI. This could be solved by some extra handling on BWAPI-remote, but we chose not to do this since we already had a functioning solution for our problem by using `constructionSite(BX, BY)`.

EIS changes: The EIS changes, discussed in section 3.3.2, were found while implementing BWAPI-remote as some of the functions needed to be remotely accessible.

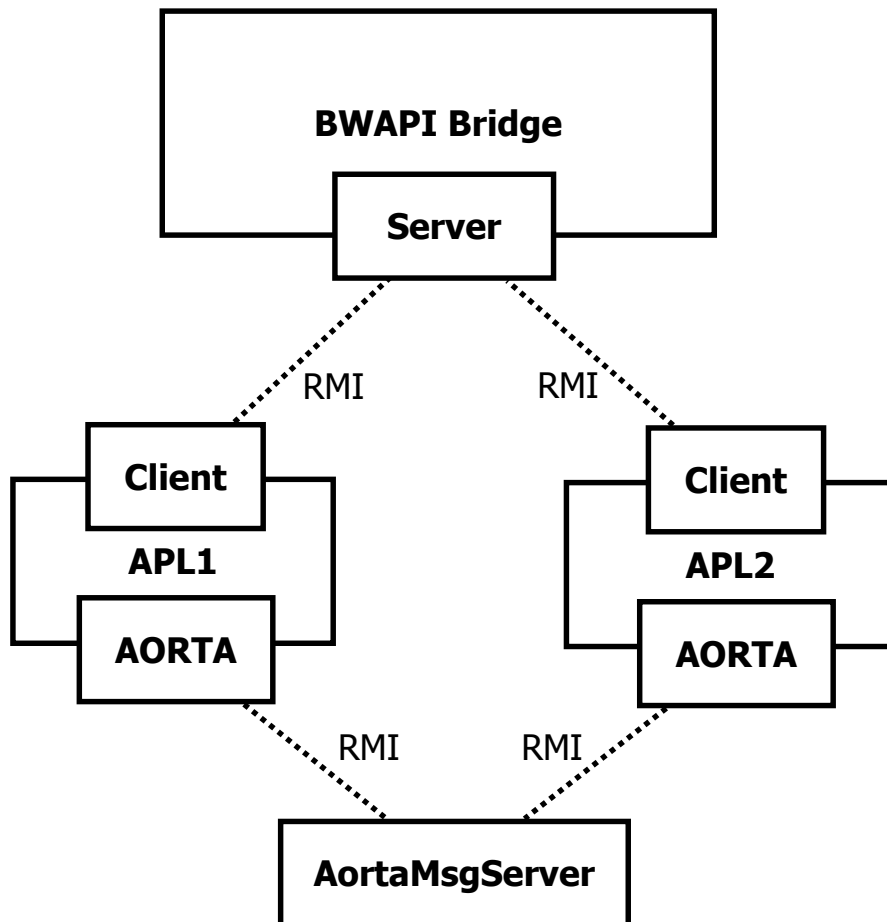


Figure 3.6: Illustration of the entire system. For our system both APLs are Jason.

GOAL: It turned out to be more difficult than anticipated to make a GOAL program, as the two IDEs from the homepage offered different possibilities (see [13]). Their own IDE did not allow for modules, which was quite confusing as the examples we found did not compile. It did however include a *knowledge* inspector much like the one from Jason. The Eclipse plugin did allow for modules, but had next to no implementation hints, which made debugging difficult.

GOAL documentation: The GOAL documentation needed in order to compile: GOAL itself, the plugin for eclipse, and in general the wiki on the GOAL homepage [13]; were not up to date and were missing vital parts, meaning that it was not possible to build without further assistance over email and phone. For the results of this assistance along with discoveries in the GOAL implementation: see appendix A.3.

Tests and Discussions

In this chapter we will describe the tests we have done and discuss the findings we have made. We will start with the general debugging techniques we have used, then the alpha testing of AORTA, and then the functionality of the organization we have implemented. How to run the agents we have implemented can be seen in appendix A.1 and A.2. All time comparisons are not based on an accurate average, but on an observed tendency and the screen shot from a typical run through for each system, and are there simply to show the increasing time usage.

4.1 Basic agent tests

In this section we will describe the basic tests we have been doing for our agents throughout the project. Due to a couple of reasons; i.e., none of us have been working much with APLs before and we are alpha testers on AORTA, we have been implementing the agents in a test driven fashion. Meaning that we have been implementing small parts of the system at a time to see how it works. We have been doing so by inspection of the agents *mental state* through the AORTA inspector interface, in which we could easily see if the agents got the right *beliefs* and/or if they enacted and committed to the right *roles* and *goals*, see figure 4.1.

Furthermore we have been using the oldschool debugging method of printing information to the Jason/GOAL console to see what the agents think belief. This proved to be quite effective as the agents did not always *believe* what we thought they did, see figure 4.2 for a screenshot of this from Jasons console.

4.1.1 Testing of GOAL and Jason

The two APLs we have been working with in this project needed to be tested to see how they work for our environment.

Even though GOAL and Jason both use the same programming paradigm, i.e., AOP, we have found some very distinct differences between the two of them.

- When a lot of agents were created Jason started to crash randomly around 30 agents. GOAL seemed to never crash; instead it just got extremely slow making the agents react very late and in small groups at a time.
- In an attempt to reduce duplication of code we tried to group up *knowledge* and *plans* about specific *goals* in modules. Besides getting slower GOAL started giving a lot of warnings because it thought the agent did not have the right *knowledge* even though it did when running the code. In Jason we found that modules in general seemed to work better but the IDE did not show the modules so you had to open them from outside the IDE.
- In Jason *beliefs* are automatically updated according to the incoming percepts, whereas GOAL adds them to the *beliefs* with an extra predicate `percept()`. This is simply a difference in design and both languages can do the exact same thing with some manual handling.
- Communication in GOAL seemed to be very slow as described in section 3.2.3, but worst after we reached around 45 agents, which makes it difficult to compare with Jason.

4.2 AORTA test

As a part of this project we have been alpha testing the organizational addition to agents, AORTA. In this section we will discuss how we did this, how the flow was and discuss some of the larger issues we found when using AORTA.

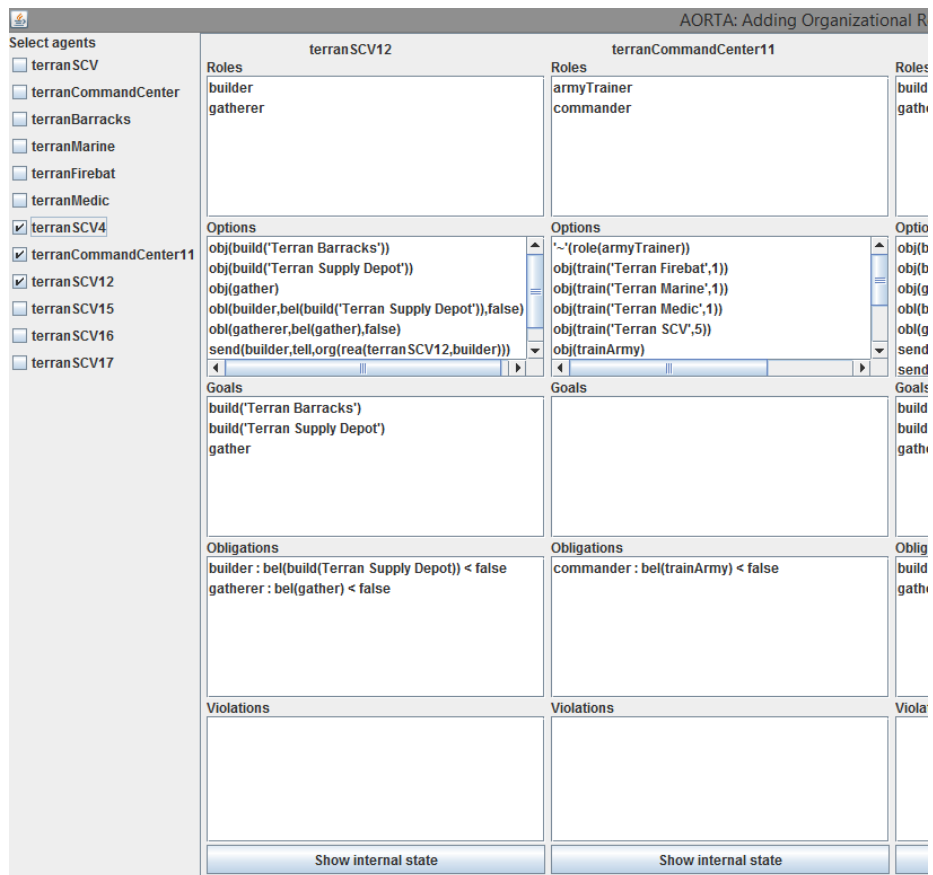
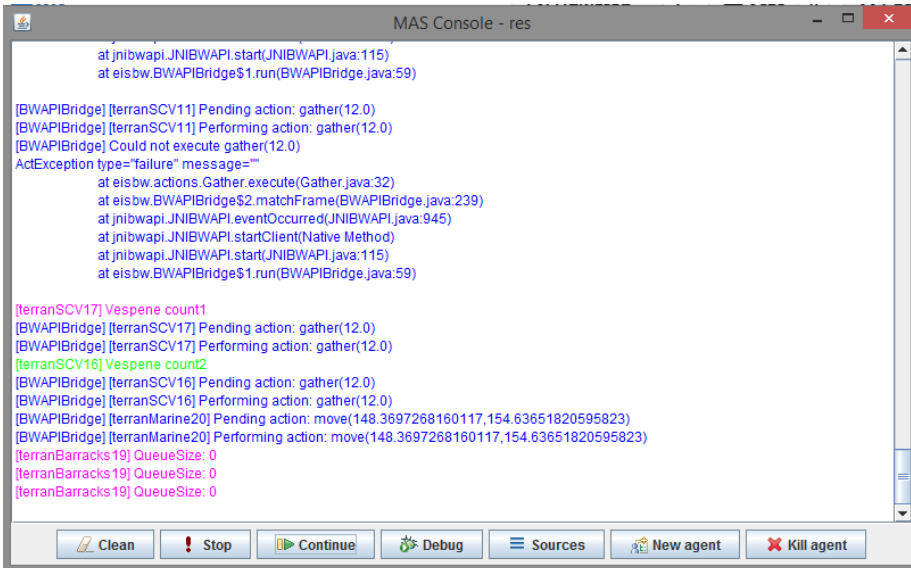


Figure 4.1: The inspector used to inspect the *organizational layer* of the agents. For each agent the complete internal state can be retrieved as a string through the button in the bottom of each chosen agent.



```
at jnibwapi.JNIBWAPI.start(JNIBWAPI.java:115)
at eisbw.BWAPIBridge$1.run(BWAPIBridge.java:59)

[BWAPIBridge] [terrancv11] Pending action: gather(12.0)
[BWAPIBridge] [terrancv11] Performing action: gather(12.0)
[BWAPIBridge] Could not execute gather(12.0)
AcException type="failure" message=""
    at eisbw.actions.Gather.execute(Gather.java:32)
    at eisbw.BWAPIBridge$2.matchFrame(BWAPIBridge.java:239)
    at jnibwapi.JNIBWAPI.eventOccurred(JNIBWAPI.java:945)
    at jnibwapi.JNIBWAPI.startClient(Native Method)
    at jnibwapi.JNIBWAPI.start(JNIBWAPI.java:115)
    at eisbw.BWAPIBridge$1.run(BWAPIBridge.java:59)

[terrancv17] Vespene count1
[BWAPIBridge] [terrancv17] Pending action: gather(12.0)
[BWAPIBridge] [terrancv17] Performing action: gather(12.0)
[terrancv16] Vespene count2
[BWAPIBridge] [terrancv16] Pending action: gather(12.0)
[BWAPIBridge] [terrancv16] Performing action: gather(12.0)
[BWAPIBridge] [terrancmarine20] Pending action: move(148.3697268160117,154.63651820595823)
[BWAPIBridge] [terrancmarine20] Performing action: move(148.3697268160117,154.63651820595823)
[terrancbarracks19] QueueSize: 0
[terrancbarracks19] QueueSize: 0
[terrancbarracks19] QueueSize: 0
```

The screenshot shows a window titled "MAS Console - res" with a scrollable text area containing the above text. At the bottom, there is a toolbar with buttons for "Clean", "Stop", "Continue", "Debug", "Sources", "New agent", and "Kill agent".

Figure 4.2: The MAS Console from Jason which we used to debug agents printing their *beliefs* directly. Here we can also see an example of an SCV trying to gather minerals from a mineral field that is not there anymore, resulting in an exception. This is an example of why percepts are *beliefs* and not *knowledge* as described in section 3.2.1

In the start we only had some general descriptions of AORTA in the form of articles discussing the idea, and some examples in which we could see some of the syntax, which meant that when we started using AORTA we were pretty much using trial and error to figure out how to use it. Some of the examples were for older versions of AORTA where no meta model was present.

Later on in the project we got a description of the language and it helped a lot. Not everything was consistent with the descriptions though, but that is part of alpha testing and we simply sent a 'bug report' and most of the errors got either fixed or explained right away. One example on this is that a role needs to have at least one *objective* assigned, which is not what is explained in the description.

We have listed the larger issues in this section according to their overall type.

4.2.1 Parse issues

In this sub-section we have listed parse related issues about AORTA:

Strings from AORTA to Jason: We found that when using *objectives* like, e.g., `train("Terran Marine", X)` in AORTA, some parse errors appeared such that they were not in accordance with the matching *plan* in the Jason Specification. Until this issue was fixed, we implemented a temporary solution by wrapping *objectives* such that the beforementioned *objective* became: `trainTerranMarineX` which we were able to match in Jason.

Capabilities with identical functors: We found that AORTA did not distinguish between two *plans* with the same functor, but different arguments, as different capabilities, e.g., `spot("Vespene Geyser")` and `spot("Enemy")`. As a temporary solution we made the plan `spot(X)` and specified the value of X inside the plan on the Jason agent.

Integer/float issue on Sub-objectives: When implementing *plan trees* using *sub-objectives*, we found that an *objective* where an integer or float was present resulted in parsing errors, since they were not parsed consistently throughout the *mental state*. An example could be the *objective* `train("Terran Marine", 1)`, when parsed as a *sub-objective* of `trainArmy` we got:

```
org(obj(trainArmy, [train("Terran Marine", 1.0)])),
```

which caused the tuProlog solver not to be able to match the two.

4.2.2 Understanding

In this sub-section we have listed issues related to our understanding of AORTA:

Objectives: When we started we thought the *objectives* were supposed to be *global objectives* meaning that AORTA would make sure only one agent were committed in achieving a *goal*, or at least make agents stop following an *objective* if another agent already achieved it. We assumed the organizational *beliefs* were *global knowledge* between the agents, this was not the case, and the agents themselves had to handle who actually were to complete the *objective*.

Objectives/Sub-objectives misunderstanding. We thought that AORTA would handle a lot of *dependencies* and *obligations* automatically; e.g., if an *objective* was defined with *sub-objectives* the *role* would automatically be dependent on the *role* capable of the *sub-objective*. This is not the case, but AORTA does generate some *organizational beliefs* making it possible to handle this in the .aorta file. E.g., `org(obj(Objective, [SubObjectives]))` and `org(dep(Role1, Role2, Objective))`, describing the *sub-objective* of an *objective* as well as the *dependencies* of a *role*.

4.2.3 Functionality issues

In this sub-section we have listed functionality issues related to AORTA, which, as earlier mentioned, is in alpha state:

Problem with Rules in Metamodel: While implementing our organization in AORTA we have had some problems with the *rules* in the meta model since it seems that not all of tuProlog works.

AORTA agents not generated for new Entities: When we started using AORTA, new agents were not generated for new entities in EIS. This was fixed after a bug-report.

Slow: When using AORTA with a lot of agents in a big environment like this we found that there were some bottlenecks making the system very slow. A voluntary sleep and disabling of the AORTA-inspector was added to address this problem. The sleep does make the agents react slower to changes in the environment, but for our purposes it is not such a big problem.

Options generated for other agents: Since we were using the inspector for debugging we were quite confused at some points seeing that some of the agents had *options* that we had not assigned them. This was an error in AORTA and was partly fixed after a bug-report; the agents still receive *options* to deact roles that they have not enacted. This resulted in agents getting stuck deacting the same *role* over and over instead of committing to new *objectives*, we fixed this in the `.aorta` file by only deacting if we actually have the *role*.

Inconsistency in context of rules in the `.aorta` file: `org(obj(...))` and `org(role(...))` could not be applied in the *context* of `.aorta` file as described in the syntax. This was settled after a bug-report.

Objectives without arguments: In the start *objectives* without arguments, like `scouting`, did not work. We had to add a random argument as a temporary solution like `scouting(X)` until the bug was fixed.

4.2.4 Prolog issues

Lastly we have listed prolog issues related to AORTA:

Rules in metamodel: `checkSubObjectives([])`. does not work even though it is valid prolog [10], instead we used:

```
checkSubObjectives(List) :- List = [].
```

In the end we did not use this either since a rule like:

```
checkSubObjectives(List) :- List = [Head|Tail].
```

did not work neither.

4.3 AORTA functional tests

In this section we will have a description of all the functional tests we have made for our AORTA agents.

Since the organization has such a large communication and calculation overhead, the decision making is so inefficient that we cannot compete with the standard

AI in SC:BW on normal maps. To address this we have made two custom maps that can be seen in figure 4.3 and 4.4.

The reason we have two almost identical maps is that the `scouting` is completely random meaning we might not always `defend` our base in the map in figure 4.4 since we do not find the Medic near the base before the enemy's base is destroyed. On the other map in figure 4.3 everything is visible, making it easy to see that the Marine spots the enemy base early on while it is defending against the medic near the base. We can also see that it does not charge the enemy base before the `commander` announces that the army is ready.

We have created one large test that covers the whole organization, and the complete replay of the game can be seen on the URL in appendix B.1. We have taken some screenshots of the replay to illustrate what happened. It has been split up into the following parts:

Commander and gatherers start up: figure 4.5.

Builders construction of the base: figure 4.6.

Scouts scouting of the map: figure 4.7.

ArmyTrainers training of units: figure 4.8.

Attackers defend and charge against the enemy: figure 4.9.

A time comparison between the MAS using Jason and AORTA and ACMAS we made in GOAL can be seen in figure 4.8c for the AORTA version and appendix B.5 for the GOAL version. Here we can see that the AORTA version uses $9.10min$ where GOAL only used $4.40min$. This is a significant performance difference as they have the same criteria for when to pursue a *goal*, but Jason with an explicit organization and GOAL with an implicit.

4.4 Remote test

We now want to test the remote parts of the system. We have chosen to do so in two steps.

1. GOAL and Jason on each client
2. AORTA Jason on two different channels.

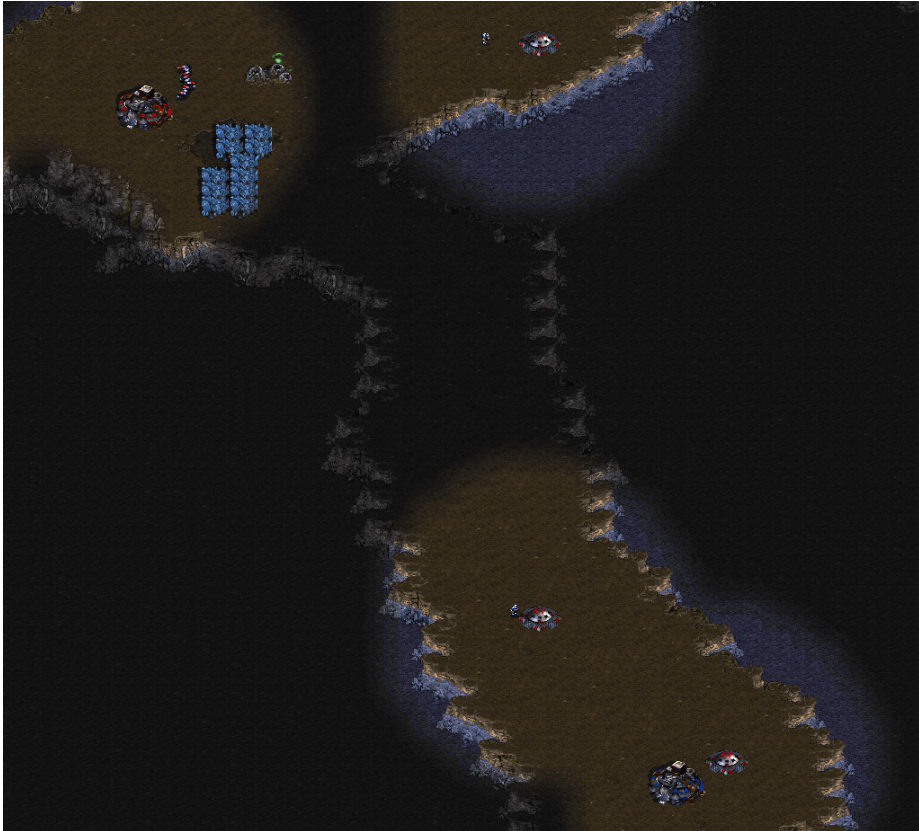


Figure 4.3: This is a custom map where we can see the enemy's units right from the beginning due to pre placed bunkers. The point of this is that we can see by inspecting the Marine that he does not charge the enemy base or the Medic in the middle of the map before the `commander` has done the `trainArmy objective`. The medic in the top right corner will be killed as it is inside a specific radius of the base.

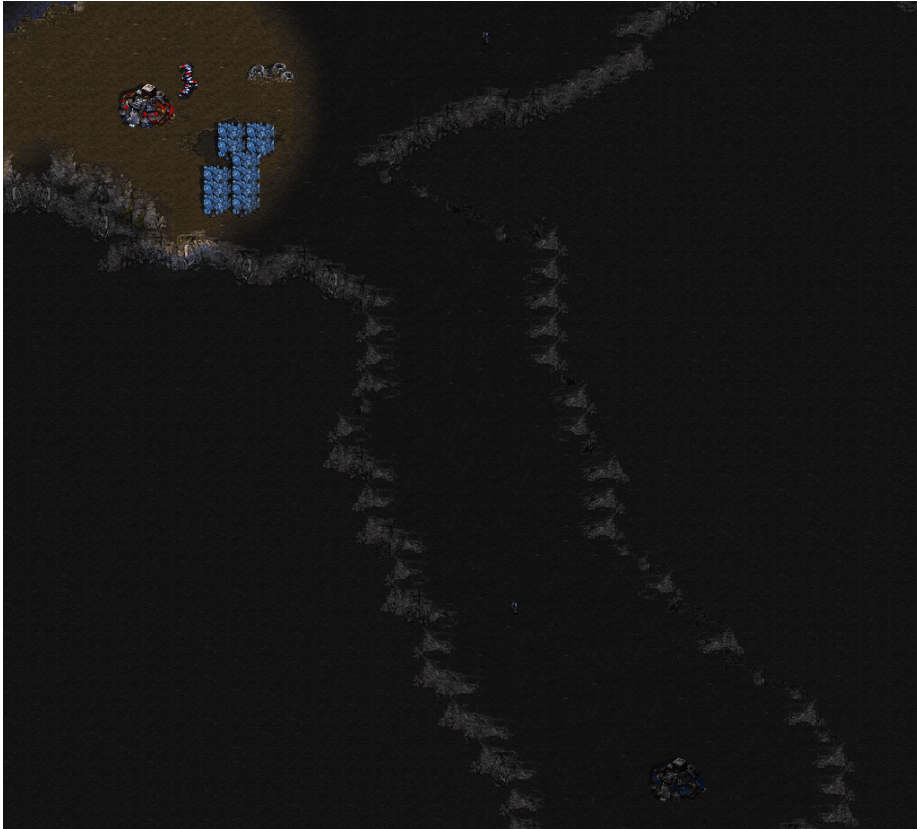


Figure 4.4: This is the same map as the one in figure 4.3 without the bunkers that shows the enemy units. The point here being is that the agents have to scout the map while constructing the base.



(a) We start with a Command Center and three SCVs. The Command Center needs to train 5 SCVs to fulfill one of the *sub-goals* of his *goal* `trainArmy`.



(b) Another SCV is being trained in the Command Center which can be seen in the bottom of the screenshot since it has been selected.



(c) The SCV has been fully trained and emerges from the building. Some of the SCVs decides to gather minerals.



(d) The last of the 5 SCVs needed for the `commander` to train an army, has been trained.

Figure 4.5: Illustration of the completion of the first *sub-objectives* of the *objective* `trainArmy` on the *role* `commander`. In these figures, we can also see that the SCVs have chosen to gather minerals since they have nothing else to do.



(a) The Supply Depot is being built since it is the first building we have resources for and the SCVs are *obligated* to build it from the start.



(b) The Barracks is being built since the commander has sent this *option* to the builders through their *dependencies* on the *objective trainArmy*.



(c) The Academy is being built as it is a *sub-objective* of the *objective* of training Firebats and Medics. Since it has no *sub-objectives*, it is often built before the Refinery.



(d) The Refinery is being built as it is a *sub-objective* of the *objective* of training Firebats and Medics, and the *sub-objective* of finding a Vespene Geyser has been achieved by the scout.



(e) This shows the final base constructed in our example.

Figure 4.6: This illustration shows how and in which order the base is being constructed due to the structure of the organization we have made.



- (a) The Marine spots an enemy Medic far from the base. (b) The Marine continues its scouting routine and spots an enemy Medic close to the base.



- (c) The Marine spots the enemy base which is the last *sub-objective* of the *charge objective* which means that the *attackers* can now charge the enemy base.

Figure 4.7: This illustration shows how the Marine scouts the map. Note that the scout function is quite random and we were just lucky to get this good results. In a MAS designed to be good at SC:BW, this is one of the things that should obviously be changed. We refer to figure 4.9 to show how the agents react to this information.



(a) The first *sub-objective* is to train a Marine, which at the same times gives us a possibility to scout as seen in figure 4.7.



(b) The Firebat is the second unit to be trained and is here seen on the choke point.



(c) The last unit to be trained is the Medic which means that the commander has achieved the goal `trainArmy`. This gives the attackers the possibility to charge the opponent base (if it has been spotted of course).



(d) This figure is actually not about the *sub-objectives* of the `trainArmy` goal, but serves more as an illustration to show that the Medic is paired up with a Firebat and follows it to keep him safe.

Figure 4.8: This illustration shows the Barracks achieve the three *sub-objectives* of training Marines, Medics and Firebats of the commanders goal `trainArmy`.



(a) Just after the Marine discovers the enemy Medic far from our base it starts attacking it. This is not part of our organization but merely a part of the game since it was too close.



(b) Just after the Marine discovers the enemy Medic near our base both the Marine and Firebat attacks it to defend the base.



(c) The Firebat **charges** the enemy base. Note that the Medic is defending the Firebat.



(d) The Marine join in on the action to **charge** the enemy base.



(e) And we end up destroying the enemy base.

Figure 4.9: This illustration shows how the agents react to what the scout discovers. Also see section 3.2.2 for a more detailed description of this.

When we have different systems connected through the RMI BWAPI Bridge, we need to handle which agents are controlled by which of the connected clients. The way we split it between the clients is by agent type; i.e., all SCVs are controlled by the same client and so on.

As the behaviour is basically the same as from the functional test, only findings will be discussed in this section.

4.4.1 GOAL-Jason RMI

This test was to see how two different ACMASs would work together, we chose to let Jason handle the buildings and GOAL handle the rest. This test went according to the plan, but worth noticing was that as Jason had very few agents to handle: it kept going way beyond its normal limit of 30 ingame agents in total, since all the infantry were handled by GOAL. This might indicate that Jason has some memory issues when handling many agents.

4.4.2 AORTA Message Server

The purpose of this test was to see if the organization would work across more than one instance of AORTA using our extended `AortaMsgServer`. We did this by using two versions of the MAS with the organization, where we let one version take the buildings and the other take the rest of the units.

We found that all the communication came through, and the agents did manage to complete their *goals* and *charge* the enemy; however, it turned out to be very slow. For a time comparison see figure 4.8c and appendix B.1 where it can be seen that when all non-random tasks are done the local RMI version takes $10.47min$ and the single system took $9.10min$. This is slower for the remote version, and in a RTS this is critical.

The time for when the enemy base is destroyed in the two examples are however $15.03min$ for the local RMI and $22.51min$ for the single system, see appendix B.2 and figure 4.9e. This last part is however based on the random *scouting* of the *scout* and is not a valid part of the test.

Additionally we tested it using an actual distributed system, having one computer hosting the server and a client, and another hosting the other client. For this we temporarily had to hard code the IP of the server on the client. The results for the non-random tasks were $11.28min$ and for the enemy base to be

destroyed were $15.29min$, see appendix B.3 and appendix B.4. As expected this was slightly slower than the local RMI solution, it did however show that the client and server are completely separated.

As this RMI version of AORTA is very new and have a lot of unoptimized parts it does have some errors and bugs. With some optimization we believe the performance could be boosted a bit, though it will always be slower than the single system version due to the RMI overhead.

This test has shown that the `AortaMsgServer` works with two different instances of AORTA, thus it stands to reason that, if a GOAL-AORTA Bridge was implemented, making GOAL and Jason work together through AORTA, should be possible.

Conclusion

In this project our main goal has been to show how to organize agents using AORTA as the organizational layer. To do this we have been implementing agents in two APLs, i.e., GOAL and Jason. Besides this we aimed to make it possible to organize agents across multiple languages. To do this there were a lot of different systems that needed to be extended.

Here we will discuss our findings throughout the project.

5.1 BWAPI Bridge: SC:BW as environment

BWAPI Bridge distributes percepts to the agents quite well and is easy to extend with new functionalities. Unfortunately the percepts were not always as consistent in the way they were used; e.g., Walk Tiles and Build Tiles were used as X and Y in different percepts resulting in weird errors. With some refactoring of some percepts it did however work quite well.

For implementing an efficient AI for StarCraft: Brood War, this system is not optimal as the *actions* need to go through several different systems in order to reach the game, and so do the percepts in order to get back. This causes the percept get to the agents late, which then in turn means that it is very

difficult to compete with the built in AI of the game. That said it is useful for visualization purposes of MASs.

5.2 AORTA as organizational layer

Seeing as AORTA is only in alpha state we were impressed with how well the separation of code helped organize the agents. This was especially when comparing the logic needed to implement *dependencies*: in GOAL with communication, and our Jason agents using the *explicit organization* through AORTA.

When this has been said, we do believe that there are still a lot of things that could be optimized in future development, when using it as we have. One thing being *common organizational knowledge* since this could potentially automatically infer *dependencies* from the definitions of **Roles**, **Objectives** and **Capabilities** of an agent.

The organization would have been a lot easier to test if we had not chosen a real time simulating environment such as SC:BW but instead had used a *discrete environment* with no time limit on computations. As it is now both AORTA and the other systems, we have been using, are all still not quite finished or optimized, meaning that their are to computationally heavy to run in a real time environment.

5.3 GOAL-AORTA Bridge

In order to make it possible to use AORTA with other languages than Jason: a bridge to the given language is needed. In our case our goal was to implement AORTA for GOAL since it is very similar to Jason as both are based on the BDI model, implements EIS and are implemented in Java. Unfortunately we never got to implement this due to problems compiling the GOAL-core and GOAL-plugin for eclipse.

We did however manage to implement agents in GOAL that are ready for AORTA with minimal changes to the code.

5.4 RMI BWAPI Bridge

In order to connect multiple instances of APLs to BWAPI Bridge we successfully implemented an RMI version of the BWAPI Bridge making it possible to connect multiple MASs to the same environment.

When implementing the system we found that both BWAPI Bridge and Jason were heavily dependent on EIS, meaning that we had to meet the interface on both sides of the RMI. This did however prove to be fairly simple to implement after we made some minor changes to EIS.

We chose to implement this as a server that needs to be started separately instead of making the agents start it automatically in order to avoid having to change the version of EIS used in the languages of all systems connecting to it.

5.5 AORTA Communication Extension

In order to make the two connected systems work together in the case of *dependencies* and *organizational communication*, AORTA needs to communicate through a server instead of the APLs' internal messaging capabilities.

We did this after analyzing the AORTA implementation and implementing the modifications to the existing communication methods. We managed to get it working, for the parts of AORTA that we have been using. Despite the fact that we only used a single language for our multi system test, and the time performance was less than optimal, we have shown the possibility of organizing MASs across multiple systems using AORTA.

5.6 GOAL and Jason agents

We successfully implemented Jason agents that are working with AORTA, we also implemented GOAL agents that work with the same strategies as the Jason agents. Unfortunately we were not able to test the GOAL Agents since we did not manage to implement the GOAL-AORTA Bridge, but the agents should be ready for testing with minimal changes to the implementation though.

Through our work with the two languages, we found that they vary a little but seeing as most of the differences are design choices, deciding which is best is a

matter of personal preferences. A system where multiple languages can work in the same organization, seems to be useful since more people can work together in their favorite languages.

5.7 Final Remarks

The whole idea about splitting the system in multiple MASs working together through the same organization, opens up for a lot of interesting possibilities. However, performance wise it is not preferable since we add yet another layer, adding even more delay to the already inefficient system. By using a less time dependent environment, e.g., a *discrete environment*, this system would make it possible to organize several heterogeneous agents across multiple languages, hereby adding a clean distinction between groups in a MAS while separating the organization completely from the agents' reasoning. When implementing large MASs this could open up for interesting possibilities, as the agents will no longer be limited to a single language.

We believe that the complete system works well as a visualization tool for MASs in research areas, but if the goal is to implement a good AI to play a game of SC:BW there are much better choices out there since all the interfaces and connections simply make the system too slow to make efficient decisions. With more time, optimization could be achieved through all layers possibly making the system fast enough to use for more than visualization purposes in a *continuous environment* like this; until then a less time dependent environment would be preferable.

APPENDIX A

Running and compiling the systems

A.1 Jason AORTA

You need to install Jason from the homepage and the AORTA from the .zip file described in appendix C.1.

We have two ways of running the system, with one system and with several.

A.1.1 Single system

When running Jason - AORTA normally, things you need to be aware of is:

- In the .mas2j file make sure that:
 - All the agents are present.
 - The msgServer flag is not set.
 - That the path for the environment is `"../../EISBW/dist/EISBW-with-deps.jar"`
- In the .aorta file make sure that `agent(X)` is used instead of `aortaAgent(X)`

when running it simply:

- run the .mas2j file in Jason.
- start BWAPI chaos launcher.

A.1.2 Local multiple clients system

When running multiple systems you need to be aware of that:

- In the .mas2j files make sure that:
 - All the agent types are present, but only in one of the systems.
 - The msgServer flag is set.
 - That the path for the environment is `"../../EISBW-remote/dist/EISBW-remote.jar"`

- In the `.aorta` file make sure that `aortaAgent(X)` is used instead of `agent(X)`.

when running the systems you need to:

- run the `server.java` in the EISBW-remote project.
- start all the `.mas2j` files in Jason.
- start BWAPI chaos launcher.

Note: Jason IDE synchronizes so you need to start two instances open one project in one, run it and start the other in the other and the run it there.

A.2 GOAL

To run GOAL do like with Jason-AORTA (see appendix A.1), but use the `.mas2g` file instead of `.mas2j` and ignore everything about AORTA, like the `.aorta` file and the `msgServer` flag.

Important thing to notice is that we use EIS 0.3, and the newest GOAL uses 0.4 which is not compatible, but an older GOAL is. For single system run, BWAPI Bridge can be modified to accept the new GOAL, in the BWAPI Bridge file under required version, but the EISBW-remote is not compatible.

A.3 GOAL-AORTA Bridge

In order to develop GOAL-AORTA Bridge first step is to compile GOAL, which is more complex than the wiki implies. You should follow their wiki with small changes:

1. Make sure that the eclipse you use is 64-bit and the one for plugins.
2. Make sure that you import the 6 project mentioned in the wiki, where the last one to import is the parent folder of the others. Eclipse wont allow it the other way around.

When everything is imported, the way you compile to the eclipse plugin is:

1. In the GOAL project:

- note the version of EIS you used, needs to be 0,4.
- right click on build.xml->run as->build ant.

2. GOALplugin

- Copy the goal.jar just created to the folder GOALplugin is pointing to in its buildpath, redirecting the build path does not work.
- place client-bridge-amd64.dll in the \GOALeclipse\lib\.
- right click on project->run as-> Eclipse application.

3. NewPlugin

- Import the GOAL agents that you want to run.
- Right click on the mas2g file->run as-> GOAL; Note: run in the top does not work.

When this is done, we need to make GOAL compatible with AORTA. In order to do that it is necessary that an AORTA-GOAL Bridge is implemented. What we hereby need to accomplish is to:

- Understand the reasoning cycle of GOAL
- Find the agent generation
- Find the mental state of the agent

As far as we can see all agents in GOAL are created using the same abstract class: `AbstractAgentFactory`, which should be possible to extend, and all agents extend `Agent`, which should make it possible to extend in a similar way to the Jason-AORTA Bridge. Then by keeping the communication in `AortaMsgServer`, this should be possible.

The mental state is probably located in the package named:

"GOAL.src.main.java.goal.core.mentalstate"

However, since the implementation is very large, it is extremely time consuming without a documentation to find the reasoning cycle and specifications of the mental state, which is why we did not have time to implement the bridge.

APPENDIX B

Test Results

B.1 Functional test Replay

The replay of our full functional tests of our agents can be seen in the .zip found in appendix C.1.

B.2 Remote AORTA test with Jason agents



Figure B.1: Screenshot of the Remote AORTA system training the last unit.



Figure B.2: Screenshot of the Remote AORTA system destroying the enemy base.

B.2.1 Distributed

The test where the system is run on two different computers.



Figure B.3: Screenshot of the distributed AORTA system training the last unit.



Figure B.4: Screenshot of the distributed AORTA system destroying the enemy base.

B.3 Final GOAL agents



Figure B.5: Screenshot of the final GOAL system training the last unit.



Figure B.6: Screenshot of the final GOAL system destroying the enemy base.

APPENDIX C

Source code

C.1 System

For source code on the RMI BWAPI Bridge (which is called `EISBW-remote`), `AortaMsgServer` and in general the complete system, we refer to the `.zip` found on:

<https://www.dropbox.com/s/j1s3o86cn19a1gr/result.zip?dl=0>

An explanation of the files in the `.zip` can be seen below:

aorta: This is the source code for AORTA. The original can be found on its GitHub page at: <https://github.com/andreasschmidtjensen/aorta>.

The main part of this folder in which we have made changes are:

```
.\result.zip\result\aorta\framework\src\java\aorta\remote\
```

We did however refactor small parts in the rest of the framework as described in section 3.4.

Custom maps: This is where the two custom maps discussed in section 4.3 is located.

Replays: This is where the replays of our functional tests discussed in section 4.3 and 4.4 are located. The replays can be seen by opening them in `SC:BW`.

- `FunctionalTestAORTA.rep` is the replay of our single Jason system using AORTA.
- `FunctionalTestAORTARMI.rep` is the replay of our Jason agents split into two MAS using AORTA on a single computer.
- `FunctionalTestAORTADistributed.rep` is the replay of our Jason agents split into two MAS using AORTA across multiple platforms.
- `FunctionalTestGOAL.rep` is the replay of our final ACMAS GOAL agents.

scbw-mas: This is the source code for the BWAPI Bridge which is located in the `EISBW` folder. The original as well as our final agents can also be found at its GitHub page at: <https://github.com/andreasschmidtjensen/scbw-mas/>

- `EISBW-remote` is where the source code for RMI Bwapi Bridge is located.

- eis-for-remote is where the source code for our modified eis is located.
- *examples\GOAL\HeterogeneousAgents* is where our final GOAL agents are located.
- *examples\GOAL\StarcraftAgents_communication* is where our GOAL agents using implicit organization in the form of communication protocols are located.
- *examples\Jason\SimpleAgentsAorta* is where our final Jason agents using AORTA is located.
- *examples\Jason\SimpleAgentsAortaRmi1* is where our final Jason agents using AORTA for the infantry part of the system is located.
- *examples\Jason\SimpleAgentsAortaRmi2* is where our final Jason agents using AORTA for the buildings part of the system is located.

C.2 Agents

We have included the source code of all our final agents. For source code on other versions of our agents we refer to the .zip with the complete system found in appendix C.1.

C.2.1 Jason: Aorta organization

```

1 %Enact roles if we have capability for at least one objective and
  we have not ignored the role.
2 role(R) : org(role(R, Os)), bel(member(X,Os)), cap(X), ~bel(ignored
  (R)) => enact(R).
3
4 if bel(me(Me)), org(rea(Me,OwnRole)){
5   % tell others about enacting roles
6   send(_,tell,org(rea(Me,OwnRole))) : bel(agent(A), A \= Me), ~bel(
  sent(A, org(rea(Me,OwnRole)))) => send(A, org(rea(Me,OwnRole)
  )).
7
8   % tell others about dependency roles
9   send(Role,tell,bel(X) : bel(agent(A), A \= Me), org(rea(A,Role))
  , ~bel(sent(A, bel(X))) => send(A, bel(X)).
10  %send(Role,achieve,bel(X)) : bel(agent(A), A \= Me), org(rea(A,
  Role)), ~bel(sent(A, opt(obj(X)))) => send(A, opt(obj(X))).
11  %send SubObjective to dependee.
12  obj(O) : ~goal(O), cap(O), ~bel(ignored(O)), org(obj(O,SOL)), bel
  (member(SO,SOL)),~bel(SO),

```

```

13     org(dep(OwnRole,Role,bel(S0))), org(rea(A,Role)), bel(agent(A
14     )),
15     ~bel(sent(A, opt(obj(S0)))) => send(A, opt(obj(S0))).
16     % commit to goals where all subobjectives are done if we have
17     capability to do so.
18     obj(0) : ~goal(0), cap(0), ~bel(ignored(0)), org(obj(0,SOL)), bel
19     (findall(S0, (member(S0, SOL), \+ bel(S0)),[])=>commit(0).
20
21     ~role(R) : org(rea(Me,R)) => deact(R).
22 }

```

Listing C.1: src/Jason/groundUnit.aorta

```

1  ROLES:
2  gatherer: gather.
3  builder: build(X).
4  scout: scouting.
5  armyTrainer: train(Unit, Amount).
6  attacker: charge; defend.
7  commander: trainArmy.
8  healer: heal.
9
10 OBJECTIVES:
11 scouting.
12 spot("Vespene_Geyser").
13 spot("Enemy_Base").
14 gather.
15 trainArmy:
16     build("Terran_Barracks");
17     train("Terran_SCV", 5);
18     train("Terran_Marine",1);
19     train("Terran_Firebat",1);
20     train("Terran_Medic",1).
21 train("Terran_SCV", 5).
22 train("Terran_Marine",1).
23 train("Terran_Firebat",1):
24     build("Terran_Academy");
25     build("Terran_Refinery").
26 train("Terran_Medic",1):
27     build("Terran_Academy");
28     build("Terran_Refinery").
29 build("Terran_Refinery"):
30     spot("Vespene_Geyser").
31 build("Terran_Barracks").
32 build("Terran_Supply_Depot").
33 build("Terran_Academy").
34 charge:
35     trainArmy;
36     spot("Enemy_Base").
37 defend.
38 heal.
39
40 DEPENDENCIES:
41 attacker > commander: trainArmy.

```

```

42 attacker > scout: spot("Enemy_Base").
43 builder > scout: spot("Vespene_Geyser").
44 armyTrainer > builder: build("Terran_Refinery").
45 armyTrainer > builder: build("Terran_Academy").
46 commander > builder: build("Terran_Barracks").
47 commander > armyTrainer: train("Terran_Firebat",1).
48 commander > armyTrainer: train("Terran_Medic",1).
49 commander > armyTrainer: train("Terran_Marine",1).
50 commander > armyTrainer: train("Terran_SCV",5).
51
52 OBLIGATIONS:
53 gatherer: gather < false | true.
54 builder: build("Terran_Supply_Depot") < false | true.
55 scout: scouting < false | true.
56 commander: trainArmy < false | true.
57 attacker: defend < false | \+ trainArmy, \+ spot("Enemy_Base").
58 attacker: charge < false | true.
59 healer: heal < false | true.

```

Listing C.2: src/Jason/scbw.mm

C.2.2 Jason: Agents for AORTA organization

```

1 MAS res {
2   infrastructure: AORTA(organization("scbw.mm"),sleep(200),
3     inspector)
4   environment: jason.eis.EISAdapter("../.../EISBW/dist/EISBW-with
5     -deps.jar")
6   agents:
7     terranSCV[aorta="groundUnit.aorta"];
8     terranCommandCenter[aorta="groundUnit.aorta"];
9     terranBarracks[aorta="groundUnit.aorta"];
10    terranMarine[aorta="groundUnit.aorta"];
11    terranFirebat[aorta="groundUnit.aorta"];
12    terranMedic[aorta="groundUnit.aorta"];
13
14   classpath: "../.../libs/eis-0.3.jar";"../.../libs/jason-eis.
15     jar";"../.../libs/jnibwapi.jar";"../.../EISBW/dist/EISBW-
16     with-deps.jar";
17 }

```

Listing C.3: src/Jason/broodwar.mas2j

```

1 isBuilding("Terran_Command_Center").
2 isBuilding("Terran_Barracks").
3 isBuilding("Terran_Supply_Depot").
4 isBuilding("Terran_Academy").
5 isBuilding("Terran_Engineering_Bay").
6 isBuilding("Terran_Refinery").
7 isBuilding("Terran_Machine_Shop").
8 isBuilding("Terran_Bunker").

```

```

9  isBuilding("Terran□Armory").
10 isBuilding("Terran□Nuclear□Silo").
11 isBuilding("Terran□Missile□Turret").
12 isBuilding("Terran□Comsat□Station").
13 isBuilding("Terran□Factory").
14
15 distance(MyX,MyY,X,Y,D)
16 :- D = math.sqrt((MyX-X)**2 + (MyY-Y)**2).

```

Listing C.4: src/Jason/generalKnowledge.asl

```

1  +gameStart <- !move.
2
3  closest("Enemy", ClosestId)
4  :- position(MyX,MyY) &
5     .findall([Dist,Id],(enemy(_,Id,WX,WY,_,_)&distance(MyX,MyY,WX,
6     WY,Dist)),L) &
7     .min(L,[ClosestDist,ClosestId]).
8
9  closestToBase("Enemy", ClosestId, ClosestDist)
10 :- friendly(_, "Terran□Command□Center", Id, X, Y, _, _) &
11    .findall([Dist,EId],(enemy(_,EId,WX,WY,_,_)&distance(WX,WY,X,Y,
12    Dist)),L) &
13    .min(L,[ClosestDist,ClosestId]).
14
15 +!charge
16 : closest("Enemy", ClosestId)
17 <- attack(ClosestId); .wait(200); !!charge.
18 +!charge
19 : lastSpottedEnemy(Id,X,Y)
20 <- move(X,Y); .wait(2000); attack(Id).
21 -!charge
22 <- .wait(200); !!charge.
23
24 +!defend
25 : closestToBase("Enemy", ClosestId, ClosestDist) &
26   ClosestDist < 125
27 <- attack(ClosestId).
28 -!defend
29 <- .wait(200); !!defend.
30
31 +!move
32 : position(MyX,MyY) &
33   .findall([D,X,Y],(chokepoint(X,Y) & jia.tileDistance(MyX,MyY,X
34   ,Y,D)), L) &
35   .min(L, [_ ,X,Y])
36 <- move(X,Y).

```

Listing C.5: src/Jason/groundUnit.asl

```

1  +!spot("Vespene□Geyser")

```

```

2   : vespeneGeyser(Id, _, _, BX, BY) &
3     not spottedVespeneGeyser(Id, BX, BY)
4   <- .broadcast(tell, spottedVespeneGeyser(Id, BX, BY));+
      spottedVespeneGeyser(Id, BX, BY);+spot("Vespene_Geyser").
5 +!spot("Vespene_Geyser") <- .wait(200); !!spot("Vespene_Geyser").
6
7 +!spot("Enemy_Base")
8   :   enemy(Type, Id, WX, WY, _, _) &
9       isBuilding(Type) &
10      not lastSpottedEnemy(Id, WX, WY)
11  <- .broadcast(tell, lastSpottedEnemy(Id, WX, WY)); +
      lastSpottedEnemy(Id, WX, WY);+spot("Enemy_Base").
12 +!spot("Enemy_Base") <- .wait(200).
13 -!spot("Enemy_Base") <- .wait(200).
14
15 +!scouting
16   : spot("Enemy_Base")
17   <- +scouting.
18 +!scouting
19   : friendly(Name, "Terran_Command_Center", _, ComX, ComY, _, _) &
20     position(MyX, MyY) &
21     distance(MyX, MyY, ComX, ComY, D) &
22     .findall([Name, OtherX, OtherY, OtherD], (friendly(Name, "Terran_
      Marine", _, OtherX, OtherY, _, _) & distance(OtherX, OtherY,
      ComX, ComY, OtherD) & OtherD > D), []) &
23     map(MapWidth, MapHeight)&
24     .random(Rand1)& X = Rand1 * MapWidth * 4 &
25     .random(Rand2)& Y = Rand2 * MapHeight * 4
26     <- move(X, Y); .wait(2500).
27
28 +!scouting <- .wait(200).
29 -!scouting <- .wait(200).

```

Listing C.6: src/Jason/scouting.asl

```

1 { include("generalKnowledge.asl") }
2 { include("trainer.asl") }
3 ignored(commander).
4
5 cost("Terran_Marine", 50, 0, 2).
6 cost("Terran_Firebat", 50, 25, 2).
7 cost("Terran_Medic", 50, 25, 2).

```

Listing C.7: src/Jason/terranBarracks.asl

```

1 { include("generalKnowledge.asl") }
2 { include("trainer.asl") }
3
4 cost("Terran_SCV", 50, 0, 2).
5
6 +!trainArmy
7   <- +trainArmy.

```

Listing C.8: src/Jason/terranCommandCenter.asl

```

1 { include("generalKnowledge.asl") }
2 { include("groundUnit.asl") }

```

Listing C.9: src/Jason/terranFirebat.asl

```

1 { include("generalKnowledge.asl") }
2 { include("groundUnit.asl") }
3 { include("scouting.asl") }
4 ignored(builder).

```

Listing C.10: src/Jason/terranMarine.asl

```

1 { include("generalKnowledge.asl") }
2 +!matchUp
3   : Type = "Terran□Firebat" &
4     friendly(_, Type, Id, _, _, _, _)
5   <- +match(Id); .print("Matched□with:□", Id).
6
7 +!matchUp
8   <- .wait(1000); !matchUp.
9
10 +!heal
11   : match(Id) &
12     friendly(_, _, Id, WX, WY, _, _) &
13     position(MyX,MyY) &
14     distance(WX,WY,MyX,MyY,Distance) &
15     Distance > 16
16   <- move(WX,WY).
17 +!heal
18   : match(Id) &
19     not friendly(_, _, Id, _, _, _, _)
20   <- -match(Id); !matchUp.
21 +!heal
22   : not match(Id)
23   <- !matchUp.
24 -!heal
25   <- .wait(200); !!heal.

```

Listing C.11: src/Jason/terranMedic.asl

```

1 { include("generalKnowledge.asl") }
2
3 cost("Terran□Supply□Depot", 100, 0).
4 cost("Terran□Barracks", 700, 0).
5 cost("Terran□Academy", 150, 0).
6 cost("Terran□Engineering□Bay", 125, 0).
7 cost("Terran□Bunker", 100, 0).
8 cost("Terran□Refinery", 100, 0).
9
10 findBuildingLocation(Id,Building,ResX,ResY)
11   :- friendly(_, "Terran□Command□Center", _, _, _, CX, CY) &
12     .findall([Dist,X,Y],(constructionSite(X,Y)&distance(CX,CY,X,Y,
13     Dist)), L) &

```



```

13     .min(L, [_ ,ResX,ResY]).
14
15 canBuild(Building, X, Y)
16 :- cost(Building, M, G) &
17    minerals(MQ) & M <= MQ &
18    gas(GQ) & G <= GQ &
19    friendly(_, "Terran□Command□Center", Id, _, _, _, _) &
20    findBuildingLocation(Id, Building, X, Y) &
21    buildTilePosition(MyX,MyY) &
22    distance(MyX,MyY,X,Y,D) &
23    .findall([OtherX,OtherY,OtherD], (friendly(Name, "Terran□SCV",
24        _, _, _, OtherX, OtherY) & distance(OtherX,OtherY,X,Y,
25        OtherD) & OtherD < D), []).
26
27 closest("mineral□Field", ClosestId)
28 :- buildTilePosition(MyX,MyY) &
29    .findall([Dist,Id],(mineralField(Id,_,_,X,Y)&distance(MyX,MyY,X
30        ,Y,Dist)),L) &
31    .min(L,[ClosestDist,ClosestId]).
32
33 +!build(Building)
34 : unit(Building,_)
35 <- +build(Building).
36
37 +!build(Building)
38 : Building = "Terran□Refinery" &
39    spottedVespeneGeyser(Id, X, Y) &
40    buildTilePosition(MyX,MyY) &
41    distance(MyX,MyY,X,Y,D) &
42    .findall([OtherX,OtherY,OtherD], (friendly(Name, "Terran□SCV",
43        _, _, _, OtherX, OtherY) & distance(OtherX,OtherY,X,Y,
44        OtherD) & OtherD < D), [])
45 <- !build(Building, X-2, Y-1).
46
47 +!build(Building)
48 : not Building="Terran□Refinery" &
49    canBuild(Building, X, Y)
50 <- !build(Building, X, Y).
51
52 +!build(Building) <- .wait(200).
53 -!build(Building) <- .wait(200).
54
55 +!build(Building, X, Y)
56 : not(busy) &
57    cost(Building, M, G) &
58    minerals(MQ) & M <= MQ &
59    gas(GQ) & G <= GQ
60 <- +busy; build(Building, X, Y); .wait(2000); -busy.
61
62 +!build(Building, X, Y)
63 <- .wait(200); !build(Building, X, Y).
64
65 +!gather
66 : gathering(vespene) &
67    .findall(_, gathering(_, vespene), L) &
68    .length(L, N) &
69    .print("Stopping□gathering", N) &

```

```

62     N >= 3 //Off by one due to own ID not present in gathering(Id,
        vespene) but in gathering(vespene)
63     <- stop.
64 +!gather
65     : not(busy) &
66       friendly(_, "TerranRefinery", Id, _, _, _, _) &
67       .findall(_, gathering(_, vespene), L) &
68       .length(L, N) &
69       N < 3 &
70       .print("VespeneCount",N)
71     <- +busy; gather(Id); .wait(2000); -busy.
72 +!gather
73     : not(busy) &
74       not(gathering(_)) &
75       id(MyId)&
76       closest("mineralField",ClosestId)
77     <- +busy;gather(ClosestId); .wait(2000); -busy.
78 -!gather <- .wait(200).
79 +!gather <- .wait(200).

```

Listing C.12: src/Jason/terranSCV.asl

```

1  unitType("TerranMarine").
2  unitType("TerranSCV").
3  unitType("TerranFirebat").
4  unitType("TerranMedic").
5
6  canTrain(Unit) :-
7      queueSize(Q) & Q < 3 &
8      cost(Unit, M, G, S) &
9      minerals(MQ) & M <= MQ &
10     gas(GQ) & G <= GQ &
11     supply(C, Max) & C + S <= Max.
12
13 +gameStart <- !!maintainMentalState.
14
15 +!train(Unit, Y)
16     : unit(Unit, Count) &
17       Count >= Y
18     <- +train(Unit, Y).
19 +!train(Unit, Y)
20     : not training &
21       not unit(Unit, _) &
22       queueSize(Q) &
23       Q < Y &
24       canTrain(Unit)&
25       .print("QueueSize: ", Q)
26     <- +training; train(Unit); .wait(2000); -training; !!train(Unit,
27     Y).
28 +!train(Unit, Y)
29     : not training &
30       unit(Unit, Count) &
31       queueSize(Q) &
32       Q+Count < Y &
33       canTrain(Unit)&

```

```

33     .print("QueueSize:␣", Q, "␣Count:␣",Count)
34     <- +training; train(Unit); .wait(2000); -training; !!train(Unit,
        Y).
35
36     -!train(Unit,Y) <- .wait(200); !!train(Unit,Y).
37
38     +!maintainMentalState
39     : unitType(Unit) &
40       not cost(Unit,_,_,_) &
41       not ignored(train(Unit,_)) &
42       .print("Updating␣ignored", ignored(train(Unit,X)))
43     <- +ignored(train(Unit,_)); !!maintainMentalState.
44     +!maintainMentalState
45     : train(Unit,Count) &
46       not (unit(Unit,Count2) & Count2 >= Count) &
47       .print("removing␣train␣belief")
48     <- -train(Unit,Count); !!maintainMentalState.
49     +!maintainMentalState
50     <- .wait(200); !!maintainMentalState.
51     -!maintainMentalState
52     <- .wait(200); !!maintainMentalState.

```

Listing C.13: src/Jason/trainer.asl

C.2.3 GOAL: Agents for AORTA organization

```

1  environment {
2    env = "../..../EISBW/dist/EISBW-with-deps.jar".
3  }
4
5  agentfiles{
6    "TerranSCV.goal" [name = terranSCV].
7    "TerranBarracks.goal" [name = terranBarracks].
8    "TerranMarine.goal" [name = terranMarine].
9    "TerranFirebat.goal" [name = terranFirebat].
10   "TerranMedic.goal" [name = terranMedic].
11   "TerranSupplyDepot.goal" [name = terranSupplyDepot].
12   "TerranAcademy.goal" [name = terranAcademy].
13   "TerranEngineeringBay.goal" [name = terranEngineeringBay].
14   "TerranRefinery.goal" [name = terranRefinery].
15   "TerranCommandCenter.goal" [name = terranCommandCenter].
16 }
17
18 launchpolicy{
19   when [type = terranSCV]@env do launch terranSCV:terrnanSCV.
20   when [type = terranBarracks]@env do launch terranBarracks:
        terranBarracks.
21   when [type = terranMarine]@env do launch terranMarine:
        terranMarine.
22   when [type = terranFirebat]@env do launch terranFirebat:
        terranFirebat.

```

```

23  when [type = terranCommandCenter]@env do launch
      terranCommandCenter:terranCommandCenter.
24  when [type = terranMedic]@env do launch terranMedic:terranMedic.
25  when [type = terranSupplyDepot]@env do launch terranSupplyDepot:
      terranSupplyDepot.
26  when [type = terranAcademy]@env do launch terranAcademy:
      terranAcademy.
27  when [type = terranEngineeringBay]@env do launch
      terranEngineeringBay:terranEngineeringBay.
28  when [type = terranRefinery]@env do launch terranRefinery:
      terranRefinery.
29  }

```

Listing C.14: src/GOAL/HeterogeneousAgents.mas2g

```

1  main module{
2    knowledge{
3      #import "GeneralKnowledge.pl".
4      shouldUpgrade('U-238□Shells').
5      shouldUpgrade('Caduceus□Reactor').
6      %shouldUpgrade('StimPack').
7      %shouldUpgrade('Restoration').
8      %shouldUpgrade('Optic Flare').
9    }
10   beliefs {
11   }
12   goals{
13   }
14   }
15   program[order=linearall]{
16     if true then handlePercepts+upgrade.
17   }
18 }
19 }
20
21 #import "UpgradeHandler.mod2g".
22 #import "Communication.mod2g".
23 #import "PerceptHandler.mod2g".
24
25 event module {
26   program[order=linearall] {
27     if true then idNameMapping+clearSent.
28   }
29 }

```

Listing C.15: src/GOAL/TerranAcademy.goal

```

1  main module{
2    knowledge{
3      #import "GeneralKnowledge.pl".
4    }
5    goals {
6      train('Terran□Marine',50).
7    }

```

```

8     train('Terran□Firebat',50).
9     train('Terran□Medic',50).
10
11 }
12 program{
13     if true then train.
14
15 }
16 }
17 #import "TrainHandler.mod2g".
18
19 #import "Communication.mod2g".
20
21 #import "PerceptHandler.mod2g".
22
23
24 event module {
25     program[order=linearall] {
26         if true then idNameMapping+handlePercepts+clearSent.
27     }
28 }

```

Listing C.16: src/GOAL/TerranBarracks.goal

```

1 main module{
2     knowledge{
3         #import "GeneralKnowledge.pl".
4
5
6
7         nearestField(vespene, Id, X, Y) :- findall( (BestId,BestX,BestY
8             ),
9             percept(friendly(_, 'Terran□Refinery',BestId,_,_
10                ,BestX,BestY)),
11                [M1|Rest]),
12                nearestField(M1, Rest, (Id, X, Y)).
13
14         nearestField(MineralField, [], MineralField).
15         nearestField((BestId,BestX,BestY), [(Id,X,Y)|Rest],
16             MineralField) :-
17             distanceTo(BestX,BestY,Res),
18             distanceTo(X,Y,Res1),
19             Res <Res1 -> nearestField((BestId,BestX,BestY),
20                 Rest, MineralField)
21                 ; nearestField((Id,X,Y), Rest,
22                     MineralField).
23
24         shouldMineVespene(Id,VesId, X, Y) :- percept(unit('Terran□
25             Refinery',RefCount)),
26             NeededWorkers is RefCount * 3,
27             aggregate_all(count, percept(workerActivity(_
28                 ,vespene)), Count),
29             Count < NeededWorkers,
30             chooseWorkerForVespene(Id,VesId, X,Y).

```

```

25     chooseWorkerForVespene(BestId, VesId, VesX, VesY):-
26         findall( (Id,X,Y), percept(friendly(_, 'Terran□SCV', Id, _, _, X
           , Y)), [W1|Rest]),
27         nearestField(vespene, VesId, VesX, VesY),
28         chooseWorkerForVespene(W1, Rest, VesX, VesY, (BestId, _, _)).
29
30     chooseWorkerForVespene(Worker, [], X, Y, Worker).
31     chooseWorkerForVespene((BestId, BestX, BestY), [(Id, X, Y)|Rest],
           VesX, VesY, Worker) :-
32         distance(BestX, BestY, VesX, VesY, Res),
33         distance(X, Y, VesX, VesY, Res1),
34         Res < Res1 -> chooseWorkerForVespene((BestId, BestX, BestY),
           Rest, VesX, VesY, Worker)
35         ; chooseWorkerForVespene((Id, X, Y), Rest, VesX, VesY
           , Worker).
36
37     beliefs {
38     }
39     goals{
40         train('Terran□SCV', 25).
41     }
42     program[order=linearall]{
43
44         if true then handlePercepts+train.
45
46         if bel(shouldMineVespene(Id, VesId, X, Y), nameMapping(Mail, _, Id))
           then print('Sending□message□to□SCV')+Mail.send(gather(
           vespene, VesId, X, Y)).
47
48     }
49 }
50
51 #import "TrainHandler.mod2g".
52
53 #import "Communication.mod2g".
54
55 #import "PerceptHandler.mod2g".
56
57
58 event module {
59     program[order=linearall] {
60         %if true then clearSent.
61         if true then idNameMapping+clearSent.
62         if bel(not(hasSent)) then idNameMappingBroadcast.
63
64     }
65 }
66 }

```

Listing C.17: src/GOAL/TerranCommandCenter.goal

```

1 main module{
2     knowledge{
3         #import "GeneralKnowledge.pl".
4

```

```

5     shouldUpgrade('Terran␣Infantry␣Weapons').
6
7     shouldUpgrade('Terran␣Infantry␣Armor').
8
9     }
10    beliefs {
11    }
12    goals{
13
14    }
15    program[order=linearall]{
16        if true then handlePercepts+upgrade.
17    }
18 }
19
20 #import "UpgradeHandler.mod2g".
21 #import "Communication.mod2g".
22 #import "PerceptHandler.mod2g".
23
24
25 event module {
26     program[order=linearall] {
27         if true then idNameMapping+clearSent.
28     }
29 }

```

Listing C.18: src/GOAL/TerranEngineeringBay.goal

```

1  main module{
2      knowledge{
3          #import "GeneralKnowledge.pl".
4
5      }
6      beliefs {
7      }
8      program{
9          if true then attack.
10     }
11 }
12
13
14 #import "AttackHandler.mod2g".
15
16
17 event module {
18     program {
19     }
20 }

```

Listing C.19: src/GOAL/TerranFirebat.goal

```

1  main module{
2      knowledge{
3          #import "Buildings.pl".

```

```

4     #import "GeneralKnowledge.pl".
5
6   }
7   beliefs {
8   }
9   goals{
10    spotEnemyBase.
11    spotVespeneGeyser.
12    scouting.
13  }
14  program{
15    if true then attack.
16    %Spot:
17
18    % spot goal :
19    if a-goal(spotVespeneGeyser), bel(percept(vespeneGeyser(_,_,_,
20    BX,BY)),not(percept(freindly(_, 'Terran_Refinery',_,_,_,_)))
21    ) then allother.sendonce(vespene(BX,BY)) + insert(
22    spotVespeneGeyser)+ print("sender_-vespene").
23
24    if a-goal(spotEnemyBase),bel(percept(enemy(Type,EId,Ex,Ey,_,_)
25    ,isBuilding(Type)) then allother.sendonce(lastSpottedEnemy(
26    EId,Ex,Ey)) + insert(spotEnemyBase) + insert(scouting).
27
28
29    %Scout:
30    if a-goal(scouting),bel(not(percept(enemy(_,_,_,_,_))),
31    percept(map(Width,Height)),WWidth is Width*4,WHeight is
32    Height *4,random(0,WWidth,RWidth),random(0,WHeight,RHeight)
33    )then{
34      if bel(not(percept(attacking(_))),percept(enemy(_,_,_,_,_
35      ,_))) then move(RWidth,RHeight).
36      if bel(not(percept(enemy(_,_,_,_,_)))) then move(RWidth
37      ,RHeight).
38    }
39  }
40  actionspec {
41    move(X,Y) {
42      pre {true}
43      post {true}
44    }
45  }
46 }
47
48 #import "AttackHandler.mod2g".

```

Listing C.20: src/GOAL/TerranMarine.goal

```

1  main module{
2    knowledge{
3      #import "GeneralKnowledge.pl".
4
5    }
6    beliefs {
7    }

```



```

8   program{
9     if bel(paired(Id), percept(friendly(_,_,Id,X,Y,_,_))) then move
      (X,Y).
10  }
11
12  actionspec {
13    move(X,Y) {
14      pre { percept(position(SelfX,SelfY)), distance(X,Y,SelfX,
      SelfY,Res), Res >=10}
15      post {true}
16    }
17  }
18 }
19
20 event module {
21   program {
22     if bel(not(paired(_)), percept(friendly(_,'Terran_Marine', Id,
      X, Y, _, _)), percept(id(SelfId)),
23     string_concat('Medic_', SelfId, Res), string_concat(Res, '
      paired_up_with_Terran_Marine:', Res2),
24     string_concat(Res2, Id, Res3)) then print(Res3)+insert(paired
      (Id)).
25     if bel(paired(Id), not(percept(friendly(_,_, Id, _, _, _))))
      then delete(paired(Id)).
26   }
27 }

```

Listing C.21: src/GOAL/TerranMedic.goal

```

1   main module{
2     knowledge{
3
4     }
5     beliefs {
6     }
7     goals{
8
9     }
10    program[order=linearall]{
11      if bel(percept(workerActivity(Id,Activity)),
12      string_concat(Id, Activity, Res)) then print(Res).
13      if true then print(sDFS).
14    }
15  }
16
17  #import "Communication.mod2g".
18
19  event module {
20    program[order=linearall] {
21      if true then idNameMapping+clearSent.
22    }
23  }

```

Listing C.22: src/GOAL/TerranRefinery.goal

```

1 main module{
2   knowledge{
3     #import "GeneralKnowledge.pl".
4     nearestField(mineral, Id, X, Y) :- findall( (BestId,BestX,BestY
        ),percept(mineralField(BestId,_,_,BestX,BestY)), [M1|Rest])
5     ,
6       nearestField(M1, Rest, (Id, X, Y)).
7     nearestField(vespene, Id, X, Y) :- findall( (BestId,BestX,BestY
8       ),percept(friendly(_, 'Terran_Refinery',BestId,_,_,BestX,
9         BestY)), [M1|Rest]),
10      nearestField(M1, Rest, (Id, X, Y)).
11
12 nearestField(MineralField, [], MineralField).
13 nearestField((BestId,BestX,BestY), [(Id,X,Y)|Rest],
14   MineralField) :-
15   distanceTo(BestX,BestY,Res),
16   distanceTo(X,Y,Res1),
17   Res <Res1 -> nearestField((BestId,BestX,BestY),
18     Rest, MineralField)
19   ; nearestField((Id,X,Y), Rest,
20     MineralField).
21 }
22
23 beliefs{
24 }
25 goals{
26   delivered(minerals).
27 }
28
29 program[order = linearall]{
30   if true then handlePercepts+handleConstruction.
31
32   if bel(not(percept(constructing))) then {
33
34     if bel(gather(vespene,Id,X,Y), not(percept(gathering(vespene)
35       ))) then print('Started_printed_vespene')+gather(Id).
36
37     if bel(not(gather(_,_,_,_))) then {
38
39       if bel(not(carrying),percept(carrying)) then insert(
40         carrying).
41       if bel(carrying, not(percept(carrying))),
42         goal(delivered(Resource)) then delete(carrying)+
43         insert(delivered(Resource))+
44         drop(delivered(Resource)).
45
46       if a-goal(delivered(Resource)),
47         bel(not(gathering(Resource))),
48         nearestField(mineral, Id, X, Y) then
49         insert(gathering(Resource))+
50         gather(Id).
51
52     }
53 }
54 }
55 }

```

```

46         if bel(delivered(OldResource)) then delete(delivered(
47             OldResource))+
48             adopt(delivered(mineral)).
49     }
50 }
51
52 actionspec {
53     gather(Id) {
54         pre { true }
55         post { true }
56     }
57 }
58
59 }
60
61 #import "Communication.mod2g".
62 #import "ConstructionHandler.mod2g".
63 #import "PerceptHandler.mod2g".
64
65 event module {
66     program {
67         if bel(received(Sender, gather(vespene, Id, X, Y))) then
68             insert(gather(vespene, Id, X, Y))+
69             delete(received(Sender, gather(vespene, Id, X, Y))).
70         forall bel(not(percept(gathering(Resource))), gathering(Resource
71             )) do delete(gathering(Resource)).
72
73         if true then idNameMapping+clearSent.
74         if bel(not(hasSent)) then idNameMappingBroadcast.
75     }
76 }
77 }

```

Listing C.23: src/GOAL/TerranSCV.goal

```

1  main module{
2      knowledge{
3          #import "GeneralKnowledge.pl".
4
5      }
6      beliefs {
7      }
8      goals{
9
10     }
11     program[order=linearall]{
12     }
13 }
14
15
16 #import "Communication.mod2g".
17
18

```

```

19 event module {
20   program[order=linearall] {
21     if true then idNameMapping+clearSent.
22
23     if bel(not(hasSent)) then idNameMappingBroadcast.
24   }
25 }

```

Listing C.24: src/GOAL/TerranSupplyDepot.goal

```

1  module attack{
2    knowledge{
3      findEnemy(Id):- percept(enemy(_,_,_,_,_,_)),
4        findall( (EId,BestX,BestY),percept(enemy(_,EId,BestX,BestY,
5          _,_)), [C1|Rest]),
6        findEnemy(C1, Rest, (Id,X,Y)).
7
8      findEnemy(Enemy, [], Enemy).
9      findEnemy((BestEId,BestX,BestY), [(Eid,X,Y)|Rest], Enemy) :-
10         walkDistanceTo(BestX,BestY,Res),
11         walkDistanceTo(X,Y,Res1),
12         Res >Res1 -> findEnemy((EId,X,Y), Rest, Enemy)
13         ;findEnemy((BestEId,BestX,BestY), Rest,
14           Enemy).
15
16   }
17   goals{
18     defend.
19     charge.
20   }
21   program{
22     if bel(received(_,lastSpottetEnemy(Id,X,Y))) then insert(
23       lastSpottetEnemy(Id,X,Y)) + insert(defend).
24
25     if goal(charge),bel(lastSpottetEnemy(Id,X,Y)) then attack(Id).
26     if goal(charge),bel(lastSpottetEnemy(Id,X,Y),findEnemy(EId),
27       percept(enemy(_,EId,_,_,_,_))) then attack(EId).
28
29     if goal(defend),bel(findEnemy(EId), percept(enemy(_,EId,Wx,Wy,_,
30       _)),percept(freindly(_,'TerranCommandCenter',Wxc,Wyc,_,_
31       )),distance(Wx,Wy,Wxc,Wyc,Distance),Distance<1000) then
32       attack(EId).
33
34   }
35
36   actionspec {
37     attack(Id) {
38       pre {percept(enemy(_,Id,_,_,_,_))}
39       post {true}
40     }
41   }
42 }

```

Listing C.25: src/GOAL/AttackHandler.mod2g

```

1 module idNameMapping{
2   knowledge {
3
4   }
5
6   program[order=linearall]{
7     forall bel(received(Sender, nameExchange(Type, Id)), percept(id(
8       MyId)), percept(unitType(MyType))) do insert(nameMapping(
9       Sender, Type, Id)) +
10      delete(received(Sender, nameExchange(
11        Type, Id))) +
12      Sender.sendonce(name(MyType, MyId)).
13
14     forall bel(received(Sender, name(Type, Id))) do insert(
15       nameMapping(Sender, Type, Id)) +
16       delete(received(Sender, name(Type, Id))).
17   }
18 }
19
20 module idNameMappingBroadcast{
21   program[order=linearall]{
22     if bel(not(hasSent), percept(id(MyId)), percept(unitType(MyType)
23       ))) then allOther.sendonce(nameExchange(MyType, MyId))+
24       insert(hasSent).
25   }
26 }
27 % Clears out received messages and sent messages, these are now
28 % processed and irrelevant, hence slowing down the queries for no
29 % reason
30 module clearSent{
31   program[order=linearall]{
32     forall bel(sent(Agent, Message)) do delete(sent(Agent, Message)).
33   }
34 }

```

Listing C.26: src/GOAL/Communication.mod2g

```

1 module handleConstruction{
2   knowledge{
3     #import "BuildingDatabase.pl".
4     #import "ResourceKnowledge.pl".
5   }
6   goals{
7     build('TerranSupplyDepot', 24).
8     build('TerranBarracks', 3).
9     build('TerranRefinery', 2).
10    build('TerranEngineeringBay', 1).

```

```

12     build('TerranAcademy',1).
13
14 }
15 program[order=linearall]{
16     %supply depot
17     %request for resource
18
19     if bel(percept(unit(Type,Amount)),build(build(Type,Old))),goal(
20         build(Type,_)) then delete(build(build(Type,Old))+insert(
21         build(build(Type,Amount))).
22
23
24     if a-goal(build(Type,Amount)),bel(shouldConstruct(Type),
25         isBuilding(Type),
26         chooseWorker(Type,WorkerId,X,Y),
27         percept(id(Id)),
28         WorkerId = Id) then {
29         if bel(gather(vespene,Id,X,Y)) then delete(gather(
30             vespene,Id,X,Y)).
31         if true then build(Type, X, Y) .
32     }
33 }
34 actionspec{
35     build(Type,X,Y) {
36         pre {canAfford(Type)}
37         post { true }
38     }
39 }
40
41 }

```

Listing C.27: src/GOAL/ConstructionHandler.mod2g

```

1 module handlePercepts{
2     beliefs{
3         minerals(0).
4         gas(0).
5         supply(0,0).
6     }
7     program[order = linearall]{
8
9         forall bel(percept(minerals(M)), minerals(CM), M \= CM) do
10             insert(minerals(M))+delete(minerals(CM)).
11         forall bel(percept(gas(G)), gas(CG), G \= CG) do delete(gas(CG)
12             )+insert(gas(G)).
13         forall bel(percept(supply(S,TS)), supply(CS,CTS), (S\=CS;TS\=
14             CTS)) do delete(supply(CS,CTS))+insert(supply(S,TS)).
15     }
16 }

```

Listing C.28: src/GOAL/PerceptHandler.mod2g

```

1 module train{
2   knowledge {
3     #import "UnitCostDatabase.pl".
4     #import "ResourceKnowledge.pl".
5
6   }
7
8   program[order = linearall]{
9     %Manage Goals
10
11    if a-goal(train(Unit,X)),bel(train(Unit,Y),percept(queueSize(
12      Size)),Size<3>Total is Y+Size>Total<X,canAfford(Unit)) then
13      train(Unit).
14
15    if a-goal(train(Unit,X)),bel(not(train(Unit,_)),percept(
16      queueSize(Size)),Size<3) then train(Unit).
17
18    %manage beliefs
19    if bel(percept(unit(Type,Amount)),train(Type,Old),Old\=Amount)
20      then delete(train(Type,Old))+insert(train(Type,Amount)).
21    if bel(percept(unit(Type,Amount)),not(train(Type,_))) then
22      insert(train(Type,Amount)).
23
24  }
25
26  actionspec {
27    train(Type) {
28      pre {canAfford(Type)}
29      post {true}
30    }
31  }
32 }

```

Listing C.29: src/GOAL/TrainHandler.mod2g

```

1 module upgrade{
2   knowledge {
3     #import "UpgradeDatabase.pl".
4     #import "ResourceKnowledge.pl".
5
6   }
7
8   program[order = linearall]{
9
10    if bel(shouldUpgrade(Type), percept(id(Id)), string_concat(Id,
11      '␣Trying␣to␣upgrade:␣', Res), string_concat(Res, Type, Res2
12      )) then print(Res2)+upgrade(Type).
13
14  }
15
16  actionspec {
17    upgrade(Type) {
18      pre {canAfford(Type)}
19      post {true}
20    }
21  }
22 }

```

```

17 }
18 }

```

Listing C.30: src/GOAL/UpgradeHandler.mod2g

```

1  isBuilding('Terran_Command_Center').
2  isBuilding('Terran_Barracks').
3  isBuilding('Terran_Supply_Depot').
4  isBuilding('Terran_Academy').
5  isBuilding('Terran_Engineering_Bay').
6  isBuilding('Terran_Refinery').
7  isBuilding('Terran_Armory').
8
9  %buildings
10 cost('Terran_Supply_Depot',100,0,0).
11 cost('Terran_Barracks',150,0,0).
12 cost('Terran_Engineering_Bay',125,0,0).
13 cost('Terran_Academy',150,0,0).
14 cost('Terran_Command_Center',400,0,0).
15 cost('Terran_Refinery',100,0,0).
16 cost('Terran_Bunker',100,0,0).
17 cost('Terran_Factory',200,100,0).
18 cost('Terran_Missile_Turret',75,0,0).
19 cost('Terran_Starport',150,100,0).
20 cost('Terran_Armory',100,50,0).
21
22
23 shouldConstruct('Terran_Supply_Depot'):-percept(supply(S,Totals
    )),CompareS is Totals-14,S>=CompareS,Totals<400.
24
25 shouldConstruct('Terran_Barracks'):- percept(supply(S,Totals)),
    Totals > 20,
    percept(unit('Terran_Barracks', Count)),
    percept(minerals(M)),
    shouldConstruct('Terran_Barracks',Count,M).
26
27
28
29
30 shouldConstruct('Terran_Barracks'):- percept(supply(S,Totals)),
    Totals > 20,
    not(percept(unit('Terran_Barracks', _))),
    percept(minerals(M)),
    shouldConstruct('Terran_Barracks',0,M).
31
32
33
34
35 shouldConstruct('Terran_Barracks',0,M):-M>=150.
36 shouldConstruct('Terran_Barracks',1,M):-M>=1000.
37 shouldConstruct('Terran_Barracks',2,M):-M>=2000.
38
39
40 shouldConstruct('Terran_Refinery') :- percept(vespeneGeyser(_,_,
    ,_,_)).
41
42
43
44 shouldConstruct('Terran_Engineering_Bay'):-percept(unit('Terran
    _Barracks', _)),
45     not(percept(unit('Terran_Engineering_Bay', _))).

```



```

46
47     shouldConstruct('Terran□Academy'):- percept(unit('Terran□
48         Barracks', _)),
49         not(percept(unit('Terran□Academy', _))).
50
51
52     %For some reason the position received from EISBW is off
53     %since we receive the center building block and the bottom
54     %left one is needed.
55     someConstructionSite('Terran□Refinery',X1,Y1):- vespene(X,Y),X1
56         is X - 2 , Y1 is Y-1.
57
58     someConstructionSite(_,X,Y):-
59         findall( (BestX,BestY),percept(constructionSite(BestX,BestY
60             )), [C1|Rest]),
61         findall( (BuildX,BuildY),(percept(friendly(_,Type,_,_,_,
62             BuildX,BuildY)), isBuilding(Type)), BuildingList),
63         percept(friendly(_, 'Terran□Command□Center', _,_,_,CX,CY)),
64         someConstructionSite(C1, Rest, CX,CY,BuildingList, (X,Y)).
65
66     someConstructionSite(ConstructionSite, [], CX,CY,BuildingList,
67         ConstructionSite).
68     someConstructionSite((BestX,BestY), [(X,Y)|Rest], CX,CY,
69         BuildingList, ConstructionSite) :-
70         distance(BestX,BestY,CX,CY,Res),
71         distance(X,Y,CX,CY,Res1),
72         Res > Res1 , distanceLargerThan(BuildingList, X,Y) ->
73         someConstructionSite((X,Y), Rest, CX,CY,BuildingList,
74             ConstructionSite)
75         ;someConstructionSite((BestX,BestY), Rest, CX,CY,
76             BuildingList, ConstructionSite).
77
78     distanceLargerThan([(CX,CY)|Rest],X,Y) :-distance(X,Y,CX,CY,Res
79         ),
80         Res > 4 -> distanceLargerThan(Rest,X,Y)
81         ;!,fail.
82
83     chooseWorker(Type,BestId,ConX,ConY):-
84         findall( (Id,X,Y),(percept(friendly(_, 'Terran□SCV',Id,_,_,
85             X,Y)),not(percept(workerActivity(Id,constructing)))), [
86             W1|Rest]),
87         someConstructionSite(Type,ConX,ConY),
88         chooseWorker(W1, Rest,ConX,ConY, (BestId,_,_)).
89
90     chooseWorker(Worker, [],X,Y, Worker).
91     chooseWorker((BestId,BestX,BestY), [(Id,X,Y)|Rest],ConX,ConY,
92         Worker) :-
93         distance(BestX,BestY,ConX,ConY,Res),
94         distance(X,Y,ConX,ConY,Res1),
95         Res < Res1 -> chooseWorker((BestId,BestX,BestY), Rest,ConX
96             ,ConY, Worker)
97         ; chooseWorker((Id,X,Y), Rest,ConX,ConY, Worker).

```

Listing C.31: src/GOAL/BuildingDatabase.pl

```

1 buildTilePosition(X,Y) :- percept(buildTilePosition(X,Y)).
2 position(X,Y) :- percept(position(X,Y)).
3
4 walkDistanceTo(X1,Y1,Res):-position(SelfX,SelfY),distance(X1,Y1,
   SelfX,SelfY,Res).
5 distanceTo(X1,Y1,Res):-buildTilePosition(SelfX,SelfY),distance(X1,
   Y1,SelfX,SelfY,Res).
6 distance(X1,Y1,X2,Y2,Res):- Res is sqrt((X2-X1)**2 + (Y2-Y1)**2).

```

Listing C.32: src/GOAL/GeneralKnowledge.pl

```

1 canAfford(Type) :- cost(Type, CostM, CostG, CostS),
2   minerals(M),
3   gas(G),
4   supply(S,TS),
5   DiffS is TS - S,
6   CostM =< M,
7   CostG =< G,
8   CostS =< DiffS.

```

Listing C.33: src/GOAL/ResourceKnowledge.pl

```

1 cost('Terran□Firebat',50,25,2).
2 cost('Terran□Medic',50,25,2).
3 cost('Terran□Marine',50,0,2).
4 cost('Terran□SCV',50,0,2).

```

Listing C.34: src/GOAL/UnitCostDatabase.pl

```

1 cost('Terran_Infantry_Weapons',100,100,0).
2 cost('Terran□Infantry□Armor',100,100,0).
3 cost('U-238□Shells',100,150,0).
4 cost('Caduceus□Reactor',150,150,0).
5 %cost('StimPack',100,100,0).
6 %cost('Restoration',100,100,0).
7 %cost('Optic Flare',100,100,0).

```

Listing C.35: src/GOAL/UpgradeDatabase.pl

```

1 isBuilding("Terran□Command□Center").
2 isBuilding("Terran□Barracks").
3 isBuilding("Terran□Supply□Depot").
4 isBuilding("Terran□Academy").
5 isBuilding("Terran□Engineering□Bay").
6 isBuilding("Terran□Refinery").
7 isBuilding("Terran□Machine□Shop").
8 isBuilding("Terran□Bunker").
9 isBuilding("Terran□Armory").
10 isBuilding("Terran□Nuclear□Silo").
11 isBuilding("Terran□Missile□Turret").
12 isBuilding("Terran□Comsat□Station").
13 isBuilding("Terran□Factory").

```

Listing C.36: src/GOAL/Buildings.pl

Acronym List

ACMAS: Agent Centered MAS.

AE: Action Execution.

AI: Artificial intelligence.

AOP: Agent Oriented Programming.

AORTA: Adding Organizational Reasoning To Agents.

API: Application Programming Interface.

APL: Agent Programming Language.

BDI: Belief Desire Intention.

BWAPI: BroodWar API.

EIS: Environment Insterface Standart.

GPS: Global Position System.

IDE: Integrated Development Environment.

JNI: Jave Native Interface.

JNIBWAPI: JNI BWAPI.

MAS: Multi-Agent System.

OCMAS: Oganization Centered MAS.

RMI: Remote Method Invocation.

RTS: Real Time Strategy.

SCV: Space Construction Vehicle.

Bibliography

- [1] Andreas Frank. On Kuhn's Hungarian Method – A tribute from Hungary.
- [2] Andreas Schmidt Jensen. AORTA: Adding Organizational Reasoning to Agents. <http://www2.compute.dtu.dk/~ascje/AORTA/>, 2014. [Online; accessed 02-January-2015].
- [3] Andreas Schmidt Jensen and Jørgen Villadsen and Virginia Dignum. The AORTA Architecture: Integrating Organizational Reasoning in Jason. 2014. Draft.
- [4] Bertsekas, D. P. The Auction Algorithm: A Distributed Relaxation Method for the Assignment Problem. *Ann. Oper. Res.*, 14(1-4):105–123, June 1988.
- [5] Blizzard. Entertainment: Star Craft. <http://us.blizzard.com/en-us/games/sc/>, 1998. [Online; accessed 17-November-2014].
- [6] Rafael H. Bordini, Michael Wooldridge, and Jomi F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.
- [7] BWAPI team. Brood War Application Programming Interface.
- [8] C. Kaysø-Rørdam. Implementing Intelligent Agents in Games, 2014. DTU Master Project, supervisor: Jørgen Villadsen, DTU Compute.
- [9] EIS team. eishub/eis. <https://github.com/eishub/eis>, 2009. [Online; accessed 07-October-2014].
- [10] Enrico Denti. tuProlog Manual, tuProlog version: 2.9.0. <http://apice.unibo.it/xwiki/bin/download/Tuprolog/Download/tuprolog-guide-2.9.0.pdf>, 2014. [Online; accessed 27-December-2014].

-
- [11] Galton, Antony. Temporal Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2008 edition, 2008.
- [12] Garson, James. Modal Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2014 edition, 2014.
- [13] GOAL Team. GOAL Documentation. <http://mmi.tudelft.nl/trac/goal/wiki/WikiStart#Documentation>, 2009. [Online; accessed 03-January-2015].
- [14] Jacques Ferber and Olivier Gutknecht and Fabien Michel. From Agents to Organizations: An Organizational View of Multi-agent Systems. In *In: LNCS n. 2935: Procs. of AOSE'03*, pages 214–230. Springer Verlag, 2003.
- [15] JNIBWAPI team. Java Native Interface Brood War Application Programming Interface. <https://code.google.com/p/jnibwapi/>, 2010. [Online; accessed 16-December-2014].
- [16] Jomi Fred Hübner and Jaime Simão Sichman and Olivier Boissier. A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems. *Proceedings of the 16th Brazilian Symposium on Artificial Intelligence*, 2002.
- [17] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *CoRR*, cs.DC/0006009, 2000.
- [18] Madhavan Mukund. Linear-Time Temporal Logic and Büchi Automata. 1997.
- [19] Shoham, Yoav. Agent-oriented Programming. *Artif. Intell.*, 60(1):51–92, March 1993.
- [20] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.
- [21] Wooldridge, Michael. *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd edition, 2009.