Alma Mater Studiorum Università di Bologna

Seconda Facoltà di Ingegneria con sede a Cesena

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

BDI agents for Real Time Strategy games

Tesi di Laurea in Sistemi Multi-Agente

Presentata da:
Andrea Dallatana

Relatore:
Chiar.mo Prof. Andrea Omicini

Correlatore:
Ing. Danilo Pianini

Sessione II
Anno Accademico 2011/2012

Abstract

While the use of distributed intelligence has been incrementally spreading in the design of a great number of intelligent systems, the field of Artificial Intelligence in Real Time Strategy games has remained mostly a centralized environment. Despite turn-based games have attained AIs of world-class level, the fast paced nature of RTS games has proven to be a significant obstacle to the quality of its AIs.

Chapter 1 introduces RTS games describing their characteristics, mechanics and elements.

Chapter 2 introduces Multi-Agent Systems and the use of the Beliefs-Desires-Intentions abstraction, analysing the possibilities given by self-computing properties.

In Chapter 3 the current state of AI development in RTS games is analyzed highlighting the struggles of the gaming industry to produce valuable. The focus on improving multiplayer experience has impacted gravely on the quality of the AIs thus leaving them with serious flaws that impair their ability to challenge and entertain players.

Chapter 4 explores different aspects of AI development for RTS, evaluating the potential strengths and weaknesses of an agent-based approach and analysing which aspects can benefit the most against centralized AIs.

Chapter 5 describes a generic agent-based framework for RTS games where every game entity becomes an agent, each of which having its own knowledge and set of goals. Different aspects of the game, like economy, exploration and warfare are also analysed, and some agent-based solutions are outlined. The possible exploitation of self-computing properties to efficiently organize the agents activity is then inspected.

Chapter 6 presents the design and implementation of an AI for an existing Open Source game in beta development stage: 0 a.d., an historical RTS game on ancient warfare which features a modern graphical engine and evolved mechanics.

The entities in the conceptual framework are implemented in a new agent-based platform seamlessly nested inside the existing game engine, called ABot, widely described in Chapters 7, 8 and 9.

Chapter 10 and 11 include the design and realization of a new agent based language useful for defining behavioural modules for the agents in ABot, paving the way for a wider spectrum of contributors.

Chapter 12 concludes the work analysing the outcome of tests meant to evaluate strategies, realism and pure performance, finally drawing conclusions and future works in Chapter 13.

Contents

I Introduction **6**

1 Real Time Strategy games **7**
1.1 Elements and Mechanics . . . 7
1.2 Warfare . . . 8
1.3 Economy . . . 8
1.4 Common Strategies . . . 8
1.5 Playing modes . . . 8

2 Multi-Agent Systems **10**
2.1 Belief-Desire-Intention Agents . . . 10
2.2 Self-Computing . . . 10

II Artificial Intelligence in Real Time Strategy games **12**

3 RTS games and AI **13**
3.1 Current state of artificial opponents in commercial games . . . 13
3.2 Omniscience and Cheating . . . 14
3.3 Realism . . . 14

4 Towards and agent-based AI for RTS **15**
4.1 Distributed or Centralized . . . 15
4.2 Omniscience or Locality . . . 15
4.3 Optimal or Human-like . . . 15
4.4 Multi-Agent Systems as AIs . . . 16
4.5 Objectives of an agent-based approach . . . 16
4.5.1 Decentralization of the decision making process . . . 16
4.5.2 Realization of realistic behaviours . . . 16
4.5.3 Modularity . . . 17
4.5.4 Self-Organization properties . . . 17
4.5.5 Better performances . . . 17
4.5.6 Possible disadvantages . . . 17

5 A framework for an agent-based AI **18**
5.1 Every game entity as an agent . . . 18
5.2 Game world perception . . . 18
5.2.1 Locality of perceptions . . . 18
5.2.2 Distribution of knowledge . . . 19
5.3 Existing MAS platforms and RTS . . . 19
5.4 A new “ad hoc” MAS framework . . . 20
5.4.1 Reasoning Cycle . . . 20
5.4.2 Beliefs . . . 20
5.4.3 Perceptions . . . 21
5.4.4 Plans . . . 21
5.4.5 Messages . . . 21

5.4.6 Modules . . . 22
5.5 Managing the Economy . . . 22
5.5.1 Workers and Jobs . . . 23
5.5.2 Creating a Virtual Market . . . 23
5.5.3 Wealth and Choice . . . 23
5.5.4 The Stimulus-Response model . . . 24
5.5.5 Self-Configuration and Job Creation . . . 24
5.5.6 Self-Optimization and Balancing . . . 24
5.5.7 Self-Healing and Hazards . . . 25
5.6 Exploration . . . 25
5.6.1 Scouts and Game Map . . . 25
5.6.2 Self-Configuration and Dynamic Exploration Routes . . . 25
5.7 Military Mechanics and Warfare . . . 26
5.7.1 Military Hierarchy and The Top-Down approach of managing military units 26
5.7.2 Military-Economy interaction . . . 27
5.7.3 Tactical Awareness and Dynamic generation of Strategies . . . 27
5.7.4 Self-Protective reactions . . . 27
5.8 Creating unpredictability . . . 28
5.8.1 Introducing Random elements . . . 28
5.8.2 Courage and Morale . . . 28

III A practical application 29

6 0 a.d. : an Open Source Historical RTS game 30
6.1 Game elements and goals . . . 30
6.1.1 Civilizations . . . 30
6.1.2 Units and Buildings . . . 30
6.1.3 Resources and Economy . . . 31
6.1.4 Warfare . . . 32
6.2 Technical characteristics . . . 32
6.2.1 Pyrogenesys and Spidermonkey . . . 32
6.2.2 Game Logic . . . 32
6.2.3 Current AI Bots . . . 33

7 A Multi-Agent AI in 0 a.d. 34
7.1 Game Entities as BDI Agents . . . 34
7.2 Agents coordination . . . 34
7.3 Workers and Trainers in a virtual Job Market . . . 35
7.4 Game map and Exploration . . . 35

8 Design of ABot 37
8.1 The AI central module and the MAS Framework . . . 37
8.2 Behaviours as Modules . . . 39
8.3 Optimizations and coordination agents . . . 40
8.4 The Job Market . . . 40
8.4.1 Agents and Elements . . . 41
8.4.2 Job Templates . . . 41
8.4.3 Self-Configuration . . . 43
8.4.4 Value, Wealth and Choice . . . 43
8.4.5 Job Spots . . . 43
8.4.6 Resource dependent Jobs . . . 44
8.4.7 Training Jobs . . . 44
8.4.8 Build Jobs . . . 44
8.4.9 Stimulus-Response System mechanics . . . 45
8.4.10 Self-Optimization in Job Market Economics . . . 46

8.4.11 Self-Healing . . . 46
8.5 Exploration . . . 47
8.5.1 Exploration Grid . . . 47
8.5.2 Scouts . . . 47
8.5.3 Self-Configuration and Routes Generation . . . 47
8.5.4 Map interaction and Self-Healing . . . 48
8.6 Military . . . 48
8.6.1 The Military Grid and Battle Tactics . . . 48
8.6.2 Self-Protection and Base Defense . . . 49

9 Implementation **50**
9.1 Base Framework Modules . . . 50
9.1.1 Optimizations . . . 52
9.1.2 Unit Agent . . . 53
9.1.3 Building Agent . . . 54
9.2 Job Market . . . 54
9.2.1 Job Market Agent . . . 56
9.2.2 Worker-JobMarketAgent interaction . . . 57
9.2.3 Cost Jobs . . . 58
9.2.4 Stimuli . . . 59
9.3 Exploration . . . 59
9.3.1 Exploration Registry Agent . . . 59
9.3.2 Exploration Grid generation and properties . . . 60
9.3.3 Scouts and Reports . . . 60
9.4 Military . . . 61
9.4.1 Captain and Squad creation . . . 63
9.4.2 Commander and Platoon creation . . . 64
9.4.3 Generals . . . 65

IV Botalk: an agent-oriented Language for ABot **66**

10 Design of Botalk **67**
10.1 Introduction and motivations . . . 67
10.2 The language Botalk . . . 67
10.2.1 General Design considerations . . . 67
10.2.2 Use of + and $ . . . 68
10.2.3 Init Sections . . . 68
10.2.4 Beliefs . . . 68
10.2.5 Perceptions . . . 69
10.2.6 Plans . . . 69
10.2.7 Variables . . . 69
10.2.8 IF . . . 70
10.2.9 FOR . . . 70
10.2.10 Commands . . . 70
10.2.11 Assignment . . . 71
10.2.12 Removing Beliefs, Plans and Perceptions . . . 72
10.3 Grammar and Parsing . . . 72
10.4 Checking . . . 74
10.4.1 Variable declaration . . . 74
10.4.2 Perceptions Perceive clauses . . . 74
10.4.3 For cycle correctness . . . 74
10.4.4 Commands . . . 74
10.5 Translation . . . 74

11 Translator Architecture **76**
11.0.1 JavaCC parser and tokenizer . . . 77
11.0.2 Prolog and Java integration . . . 78
11.0.3 Tree Display . . . 78
11.0.4 Checking . . . 78
11.0.4.1 CheckPerceiveExp . . . 79
11.0.4.2 CheckVar . . . 79
11.0.4.3 CheckFor . . . 79
11.0.4.4 CheckCommands . . . 79
11.0.5 Translation . . . 80
11.1 Final considerations on Botalk . . . 80

V Results, Conclusions and Future Work **82**

12 Tests **83**
12.1 Performance metrics . . . 83
12.1.1 Strategy . . . 83
12.1.2 Realism . . . 84
12.1.3 Performances . . . 84
12.2 Test Results . . . 84
12.2.1 Strategy . . . 84
12.2.2 Realism . . . 87
12.2.3 Performances . . . 91

13 Conclusions and Future Work **94**

Bibliography **95**

Part I

Introduction

Chapter 1

Real Time Strategy games

A Real Time Strategy (RTS) game is a military strategy game in which the primary mode of play is in a real-time setting [3]. Two or more players control units and structures inside a game map to secure locations or destroy their opponent's assets. It is possible to create additional units and structures during the course of a match, but this is generally limited by the necessary expenditure of accumulated resources, which are gathered controlling special points on the map or using certain types of units and structures devoted to this purpose[11, 3].

The first use of the term was coined with Dune II and the games that followed it between 1992 and 1998, like Blizzard's Warcraft and Westwood Studios' Command & Conquer, helped define this genre's core concepts and mechanics, clearly setting them apart from the other strategy games that were mainly turn based [11].

1.1 Elements and Mechanics

In a typical RTS game, the screen is divided into a map area displaying the game world and terrain, units, an interface overlay containing command and production controls. Units that are issued orders start completing their assigned task and continue to do so on their own accord, allowing the player to divert its focus in another direction[11].

The gameplay is fast-paced and requires quick thinking and reflexes; to be effective the player needs to manage the forces under its command in various sections of the map and be ready to respond to multiple events at once. While in turn-based games it is possible to formulate complex plans, in real-time the decisions need to be made in a fraction of the time and usually the type of response is tactical more than strategic.

Gameplay generally consists of the player being positioned somewhere in the map with a few units or a building; from there the usual focus of the game is to start gathering resources and increase the number of owned forces. Resource gathering spots are usually finite and force the player to explore the map to find more, often competing for them with the opponent.

Game maps are usually diverse in size based on the number of players and feature different types of terrains. Being able to see only areas of the map in the vicinity of owned entities, while the rest remains obscured in a "Fog of War", is another mechanic usually found in RTS games. There are two kinds of fog: the areas never visited are pitch black and make it impossible to know what resides under it; while areas visited in the past, but where there aren't any own units or buildings, are only covered in a lighter gray that permits to see the terrain features of the map and its key elements. Since the player starts surrounded by the fog it's important to send units exploring to discover other extraction points and the position of the enemy base and units.

Every game features its own unique settings and elements, varying from the design and art of the scenarios to the different implementation of units, buildings and mechanics. Some focus on managing military units and warfare, while others introduce a more complex economic infrastructure; although mechanics can vary greatly from product to product, the military aspect must be present to call it an RTS game [11].

1.2 Warfare

The main goal of RTS games is often the destruction of the opponent's base or the complete annihilation of its units and buildings and means to produce them, then amassing a considerable force and moving it to destroy enemy buildings is key to achieve victory. Defending owned location is also equally important since the loss of extraction and production hubs can hinder the player's ability to field more units to aid in the attack and replace fallen ones, but also in preventing the enemy player in winning the game by destroying the base. Units independently battle enemy ones once engaged, leaving the players free to focus on different areas of the map simultaneously to achieve different objectives. There is usually a variety of unit types with different strengths and weaknesses and the usual approach is a "rock-paper-scissor" system where certain types are stronger versus one type while being weaker against another in a circular pattern.

1.3 Economy

In RTS games the production of units and structures is possible with the expenditure of accumulated resources, usually of various types. Resource gathering spots are located inside the game map and are harvestable from specialized units or buildings [11]. The more extraction points the player control the higher it is its production output, consequently their capture and secure are key to winning the game. Buildings are the main source of units and are usually associated with a production queue. Various buildings allow the production of different types of units where higher cost ones unlock more powerful units. The rate at which the player can create units is proportional to the number of training buildings in its control. Since a stronger economy means a higher output and quality of military units it is an important element to consider, while keeping it balanced with the military aspect of the game as resources are usually finite.

1.4 Common Strategies

Although every match has different conditions and players try to adapt to and counter specific tactics, there are some strategies that fall under common patterns. Some of the most common and simplest are "rushing" and "turtling". A rush consists in training a few units as fast as possible and sending them towards the enemy trying a surprise attack aimed at crippling its production capabilities overwhelming it and thus ending the match. "Turtling" is the opposite to rushing and consists in taking a completely defensive stance trying to fend of enemy attacks while amassing units to muster a vast army and attack the opponent in one big attack.

Other strategies include raiding the adversary's forces in small groups trying to lure its forces and then attack in a vulnerability in its defences, or try to keep the enemy under constant pressure by building military structures near its base.

1.5 Playing modes

There are normally three different modes of playing RTS games: a campaign mode, a single player skirmish mode and a multiplayer mode.

In the campaign mode the player must complete a number of levels that progresses a story, each level built around a series of tasks to be performed and completed in order to proceed; this is typically a single player experience where the scenario is pre-built with specific conditions and the opponent usually consists of predetermined responses to triggered events.

In the single player skirmish mode instead, the human player and the AI controlled opponent play under the same starting conditions. The game map is often designed to avoid giving some player any initial advantage. It is possible to play against more than one AI opponent in a "free-for-all" setting or in a team-based conflict.

The multiplayer mode is like a skirmish mode but where the opponents are other human players; it's usually possible to participate in matches with both human and AI adversaries. Friendly matches, tournaments and leagues have become so popular that the multiplayer component is

usually considered the most important part in an RTS game by most players and then consequently by developers.

Although multiplayer is now considered the predominant way to play RTS games the single player modes still remain important for different purposes: it isn't always possible to find human opponents to play with or of the same level of skill to provide a fair match. The AI adversaries are then devised in different orders of challenge trying to satisfy the widest possible range of players and providing a valuable training tool.

Chapter 2

Multi-Agent Systems

A Multi-Agent System (MAS) is a system composed of multiple autonomous and interacting software parts located within an environment [20].

The agents in a multi-agent system have several important characteristics:

- Autonomy: the agents are at least partially autonomous;
- Locality: no agent has a full global view of the system, or the system is too complex for an agent to make practical use of such knowledge;
- Decentralization: there is no centralized control for the agent..

2.1 Belief-Desire-Intention Agents

A BDI agent is an agent having certain mental attitudes of Belief, Desire and Intention, representing, respectively, the information, motivational, and deliberative states of the agent. These mental attitudes determine the system's behaviour and are critical for achieving adequate or optimal performance when deliberation is subject to resource bounds [16, 6].

- Beliefs: they represent the knowledge of the agent, they can derive from internal reasoning or from the observation of the world;
- Desires: they represent the motivational state of the agent, objectives or situations that the agent would like to accomplish;
- Intentions: they represent the deliberative state of the agent, what it has chosen to do, they are desires which the agent has committed.

2.2 Self-Computing

Self-Computing refers to the self-managing characteristics of distributed computing resources, adapting to unpredictable changes while hiding intrinsic complexity to operators and users. In a self-managing autonomic system, the human operator plays on a new role: instead of controlling the system directly, she defines general policies and rules that guide the self-management process.

There are four functional areas that can be referred as Self-* properties [18]:

- Self-configuration: Automatic configuration of components;
- Self-healing: Automatic discovery, and correction of faults;
- Self-optimization: Automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements;
- Self-protection: Proactive identification and protection from arbitrary attacks.

Even though self-managing characteristics can be coded in any distributed system, they have assumed growing significance in Multi-Agent Systems.

Software agents naturally play the role of autonomous entities subject to self-organise themselves. Usually agents are used for simulating self-organising systems, in order to better understand or establish models. The tendency is now to shift the role of agents from simulation to the development of distributed systems where components are software agents that once deployed in a given environment self-organise and work in a decentralised manner towards the realisation of a given global and possibly emergent functionality [9].

Part II

Artificial Intelligence in Real Time Strategy games

Chapter 3

RTS games and AI

The concept of Artificial Intelligence in RTS games can be divided in different areas, but two main categories emerge:

- creation of artificial opponents
- automatic responses and behaviours

The second includes all those mechanics that are present in both human and artificial gameplay, whose main purpose is to aid in simple and repetitive actions: finding the best path to a location, moving all units into formation, etc. [19]. This is currently the main focus area of AI research in this genre of games, its aim is to remove various hassles that hinder the player's fun, and it is considered a top priority. Pathfinding algorithms, reactive responses to attacks, automatic accomplishment of simple tasks, all these topics have seen evident improvements over the years, being often considered as key feature in the comparison between different products.

While RTS games have become a lot smarter in aiding the player the same cannot be stated for their ability to produce a valid opponent, since they have remained roughly the same over the years with minimal improvements. Also the majority of the observations can only be done by examining the results, since game companies are very reluctant to release any of their AI research and there is no any evidence of inter-company collaboration in this field.

3.1 Current state of artificial opponents in commercial games

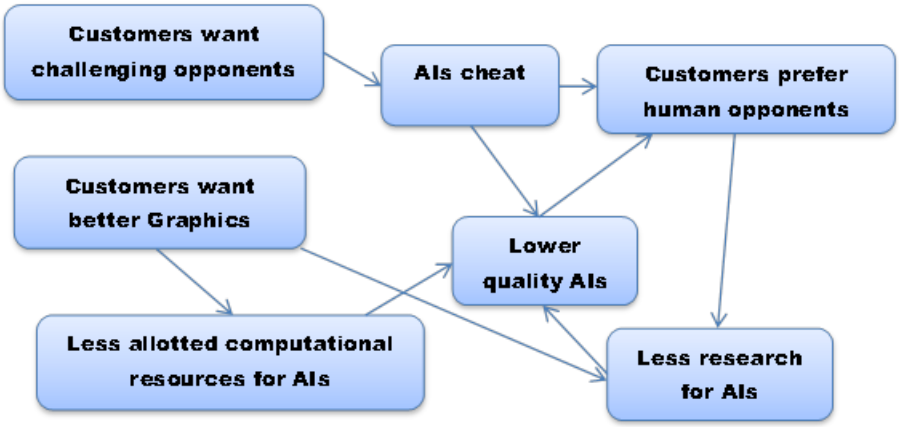
Strong and efficient AIs mostly exist for turn-based strategy games in which the majority of moves have global consequences and human planning abilities therefore can be outsmarted by mere enumeration. RTS games instead feature hundreds, if not thousands, interacting objects, often in an imperfect information scenario. With frequent micro-actions, where it is not possible to evaluate every possible decision in a weighted system; this sets the development of AIs for this type of games on a very different path from established techniques.

Video games companies create titles under severe time constraints and have not resources and incentives to seriously research in this direction; more importantly developing a good AI is not a priority in a market where the multi-player component is considered predominant [7].

Customers rate current artificial opponents very poorly and then show very low interest in this type of feature; this is both cause and effect of the current state of advancement in this field, since low attention means lower development resources and thus worse AIs.

Unappealing characteristics of current AIs are also predictability and a lack of realistic responses. The difficulty of raising the level of challenge often leads developers to take shortcuts and plainly change the game conditions in favour of their AIs making them harder to confront but also less rewarding for the player.

Figure 3.1.1: State of AI interest and reasons



3.2 Omniscience and Cheating

A simple way to increase the difficulty level of an artificial opponent, without increasing cost of development or complexity, is to make it change the rules of the game in its favour, in other words to cheat.

Quicker production of units, higher starting resource reserves, stronger units, higher difficulty levels of the AI often coincide with the adoption of one or more of these changes; usually the player can't see them because they are all hidden parameters, but an experienced one can often feel that something has been altered.

Exploration involves a certain level of complexity that developers avoid to integrate in their AIs, granting them full knowledge of the game world and thus factoring all entities in their decision algorithms. An omniscient AI ignores the fog of war system and already knows where the enemy units and buildings are and in which numbers, not allowing the player to implement misdirection tactics that would have worked with an human opponent.

3.3 Realism

Current AIs have a finite number of strategies and tactics that they implement with specific triggering conditions, their responses are always the same in a given scenario thus making them predictable. After a few matches a human player can usually predict and counter most of the AI's moves and not because the opponent is playing badly but because of its limitations. Customers then usually consider artificial opponents as poor substitutes for their human counterparts and consequently software houses rank their development as low priority. An AI that realistically emulates a player would contribute to provide a better entertainment factor and thus increase customer's interest [7].

Chapter 4

Towards and agent-based AI for RTS

It is now clear that currently used techniques for AI development in computer controlled opponents lack under various points of view. It is then possible to dissect the problem in different areas and try to approach a solution.

4.1 Distributed or Centralized

Computing systems are gradually abandoning consolidated centralized approaches to adopt distributed solutions; Artificial Intelligence research has shifted in a similar way with the introduction of Multi-Agent systems [15].

Despite a centralized AI can reach high levels of complexity and efficiency in turn-based strategy games, in RTS the decisions have to be made quickly because the AI processing is done concurrently with graphics, animations, user interface; in a real time setting where knowledge is non-deterministic they struggle to attain an high level of quality without sacrificing significant resources.

Distributed systems instead aim to solve complex problems by dividing them into smaller and lighter tasks, thus allowing more flexibility in the use of resources, also allowing for better adaptability.

4.2 Omniscience or Locality

A characteristic common to most available AIs is the knowledge of all elements composing the game simulation, including their details, their position and their status. An AI devised in this manner doesn't need to explore the map and can always consider every entity in its decision process. Even advanced AIs, that only consider elements inside their entities field of view, still enumerate and evaluate them with a global perspective.

Another way of addressing the perception of the game elements would be to use the locality principle, where each entity only knows what it can percept in a near portion of the space; this method distributes the knowledge of the system into a collection of smaller knowledge-bases. This grants the possibility to consider only a selected portion of information pertinent to the solution of a specific problem, although forcing the need for a mechanism to retrieve and coalesce the data from all different sources.

4.3 Optimal or Human-like

Algorithms are, by definition, a series of steps designed to solve a particular problem, possibly with an optimal solution. Even in AI design all choices are made aiming for the best possible result with the knowledge available. Two identical scenarios will always trigger the same response; given that AIs are normally limited to a finite number of actions, similar conditions will result in an equal effect.

The human decision making process can be analytical in the same way, although with various degrees of success, but in the time-constrained scenario of RTS games inevitably the selection of the proper action varies.

In the research for a human-like response to increase realism and to try to make the player forget that it is against an AI, the introduction of "choice" for automated systems can bring an element of unpredictability and variation in an optimal-oriented setting [12].

4.4 Multi-Agent Systems as AIs

The objective of this thesis is to demonstrate how multi-agent systems could be used to substitute standard AIs.

This approach consists in:

- mapping each controlled entity to a single agent
- providing the agents with the means to perceive the game world in their proximity
- creating a communication support to enable agents interaction
- defining a set of behaviours
- assigning each agent a subset of those behaviours
- designing the agent's plans in a way to promote cooperation and steer them towards a collective goal

4.5 Objectives of an agent-based approach

4.5.1 Decentralization of the decision making process

Each agent has its own sets of objectives and decides autonomously which to pursue. Choices are made individually based only on the knowledge possessed by the agent and not on the whole collection of available information.

There is not only one course of action being considered at the same time, but a group of disparate reasoning possibilities each with a different scope. The resulting global behaviour of the AI then is the emergent direction taken by all agents combined.

Each agent then needs to be able to:

- consider external conditions like perceptions or messages from other agents
- access its knowledge base
- discern which actions to perform based on available information

4.5.2 Realization of realistic behaviours

There are different characteristics that would make an artificial opponent more realistic in the eyes of the human player, most of them emulate what an actual player would do in a range of circumstances.

Sending explorers to scout the map is an action that identifies a player with imperfect knowledge of the world map and with limited field of vision. Human players can't see past the fog of war so they need to lift it to see the rest of the map, common AIs instead know right away the position of all enemy units causing their armies to walk straight to the enemy base without any kind of prior sign.

Another behaviour that is frequent in human players, but very rare in AIs, is to retreat units from hopeless battles to save them and regroup to safer positions.

Simply attacking with different tactics and not following the usual ones is another way to dissimulate artificiality, since even though players would often lead typical attacks they wouldn't do so in such repetitive patterns.

4.5.3 Modularity

With this approach it is possible to separate development in different modules specific to a certain task or aspect. Each agent is then the sum of a different set of behaviours.

A module could represent the typical actions of a given unit or a response common to all the units sharing some characteristic.

4.5.4 Self-Organization properties

In MASs the operation is focused on the interaction of independent entities, this leads to the attempt to find a way to autonomously organize the agents.

A RTS game can comprise of different sets of units and structures, subsequently the multi-agent system would need to be initialized appropriately each time; it is important then to design it in a way that it automatically detects the various types of entities and self-configures itself accordingly.

The agent-based AI must be able to recover from the loss of one or more agents and compensate, either by replacing them or by directing others to reorganize in order to obviate the loss.

The system must be able to detect threats and possible dangerous conditions and adapt in order to respond to them. The loss of an agent must lead towards its replacement without provoking any persistent malfunction in the system.

4.5.5 Better performances

There are two main aspects that may allow a distributed system to perform better than a centralized one: threading and distribution of workload.

A centralized AI is composed mainly of a complex calculation cycle, usually executed in a single thread. Diversely distributing the decision making process to different entities allows for the creation of a proper thread for each agent, thus increasing performances in parallel systems.

Centralized AIs' complex algorithms can be taxing and they are usually not cycled each instant of the game to avoid a too heavy load; instead they run once every fixed interval of time, creating an uneven distribution of the workload, in a way that can be represented in a series of calculation "spikes", which impact the "smoothness" of the game.

In a distributed approach instead, consisting of a collection of lighter reasoning cycles, calculations are divided into different instants, spreading their computational costs evenly throughout the game.

4.5.6 Possible disadvantages

Where a centralized AI is easily directed towards a defined set of goals, it is harder to steer a society of agents toward working on the same objectives. It is necessary to calibrate properly the initial settings in a way that the system will maintain its balance over the course of the simulation. This makes the efficiency of the system deeply binded to the quality of the design of its self-organization properties.

Distributed AIs take particular advantage in multi-threading, thus in case of centralized engines without multi-thread support the benefits to use such systems drop greatly.

Another possible disadvantage is that MASs' initial parameters impact greatly in their organization performances and it is not so easy to define what a good configuration could be like without extensive testing.

Chapter 5

A framework for an agent-based AI

In this chapter is depicted a possible framework for an agent based AI for RTS.

5.1 Every game entity as an agent

The game elements that can be controlled directly and are appropriate to be defined as agents are units and buildings, each instance of them has be associated to a corresponding agent.

Following the Beliefs-Desires-Intentions model and adapting it to the considered setting each agent should have:

- a set of beliefs that ensemble those given at its creation, those acquired through its perception of the world, those that other agents sent and those which are result of own reasoning;
- a set of perceptions that act as sensors;
- a set of desires given by the nature of the agent,plus those sent by other agents;
- intentions, namely desires, that after their conditions are verified, get executed by the agents;
- game interaction functions that work like effectors for the intentions and actuate commands to the engine.

5.2 Game world perception

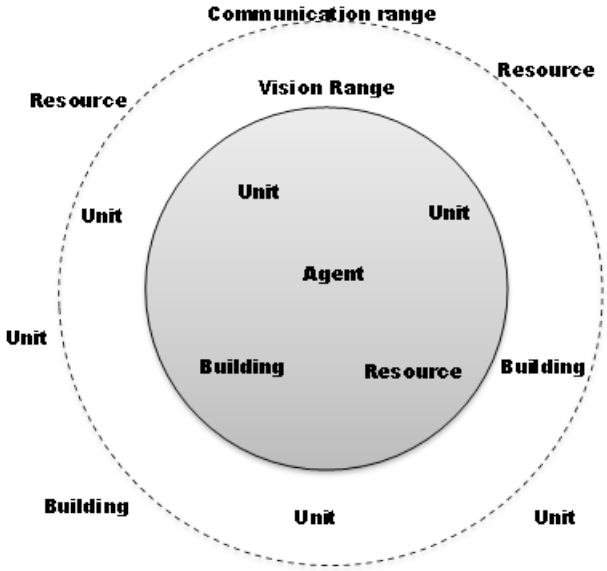
All game entities have a specific position inside the game map described as a set of coordinates. Knowing the position of these entities constitutes an important knowledge for the agents, which need a context to evaluate which actions to undertake.

it is possible to retrieve this type of information by accessing the simulation data, but this is usually unfiltered and contains all the possible entities, disregarding the position they occupy.

5.2.1 Locality of perceptions

Each unit or building has an assigned range of perception that is used to determine how far it can "see" around. The distance between the position of each game element and the agent is calculated, then only the entities that are inside the range are considered and inserted into the knowledge base. This way they only know what is around them and can decide what to do based on their surroundings. This also means that the AI as a whole doesn't know what is outside of the range of vision of its agents, but it needs to explore to know more.

Figure 5.2.1: Range of Vision and Perceptions



To know the position of something outside its view an agent must ask the other agents using messages, which can be designed not to depend on range. The existence of a communication range impacts on the ability of the agents to propagate knowledge, because it means that recently acquired beliefs need the agent that perceived them to move closer to the rest of the society to see them be integrated in the society. This kind of limitation can lead to relevant data being transmitted well after it was perceived, when it could already be obsolete and to be discarded. A scout needing to return to the base to alert of an impeding attack or a worker that wants to signal the expiration of a resource spot, both are situations where a limited communication range would hinder the efficiency of the system.

5.2.2 Distribution of knowledge

The whole sum of the AI's knowledge is divided in all its agents. Since there can be hundreds of them it is important that each agent only has the portion of knowledge it needs. Having all the agents know everything would mean steep memory costs, but it would also require significant resources just to keep them all synchronized, ultimately with data that they don't really require.

Too much information, maybe even outdated, can severely impair the agent's operation not only in terms of performances but also by tampering with its decision making. It is important then to instruct the agents to retrieve only the information they need from the other agents and keep discarding outdated knowledge.

Another mechanism that promotes communication of knowledge but without any actual messages between the agents is stigmergy. This is a form of self-organization that sees the agents leaving a trace in the environment around them in a way that affects its own behaviour or the one of other agents[13]. The use of stigmergy can then lead to the emergence of the desired global behaviour [9].

5.3 Existing MAS platforms and RTS

There are various MAS frameworks available which have arisen to implement common standards, like the one defined by FIPA [1], or to define new ones; they have been created to provide developers with an already tested and established structure for agent-based development and also aid in its standardization. Some of these are: Jade [4], Jason [5], CArtAgO [17], TuCSoN [14].

Integrating one of them in the game engine would provide numerous advantages: like the access to established protocols for defining agents, a wide range of languages and tools to aid

in the development and also, given their tested status, a clear improvement in efficiency. The downside, however, would be that it is not always feasible to integrate such platforms inside an already designed engine and their embedding could lead to unforeseen problems or a deterioration in performances caused by the bridging of different systems, thus diminishing their advantages.

Another option would be to create a new "ad hoc" framework. The efficiency of the resulting solution would however depend on how early in the engine's development cycle the MAS framework is designed and integrated. A tailored system could lead to a greater flexibility in the adaptation to the engine's mechanics and also provide with a reasoning cycle that follows more closely the needed behaviours. The downside of a custom framework would be the greater development time needed to define and test a complete and functional system.

5.4 A new "ad hoc" MAS framework

The proposed solution is to create a new framework customized to the necessities of RTS games and easily embeddable into game engines, even ones in an advanced stage of development. The new platform's design is inspired by Jason [5], from which it instantiates its reasoning cycle to the specific needs of the target system. In this section then some possible abstractions are proposed to manage the different cornerstones needed to agent-based solutions.

5.4.1 Reasoning Cycle

The reasoning cycle is composed by the succession of a series of steps that define how the agent acquires information, considers it and then acts on it based on its own agenda. The proposed solution involves four processes in order:

1. The agent perceives the game environment around it, recognizing the desired elements and adding them to its knowledge-base;
2. The agent open the received messages and evaluates their content;
3. The agent selects from the available desires the ones it wants to pursue;
4. The agent pursues its intentions by acting on them.

Figure 5.4.1: Reasoning Cycle

5.4.2 Beliefs

Beliefs represent the knowledge the agent has at its disposal. There can be a set of beliefs already present at the initialization of the agent, then the others are acquired with perceptions and messages.

A possible representation for a belief is:

- Name
- Key
- Data

Every combination of Name and Key represents the identification for a Belief. Data is structured containing all the desired information.

5.4.3 Perceptions

Each agent has a collection of sensors that let it perceive the state of the game around it. Perceptions constitute a direct access to game data from which new Beliefs are extrapolated and then added to the belief-base.

Every agent has a series of perceptions at its disposal, but only the active ones are destined to enter the reasoning cycle. It is important to access only a specific type of information with each perception, doing so it is possible to reduce the amount of useless data being registered.

A perception can be set to be run only once after activation or to be continuously running until deactivation.

5.4.4 Plans

The "desires" of the agents are represented by constructs named "Plans", inspired by the ones used in AgentSpeak [5], which constitute all the possible actions they can consider to undertake.

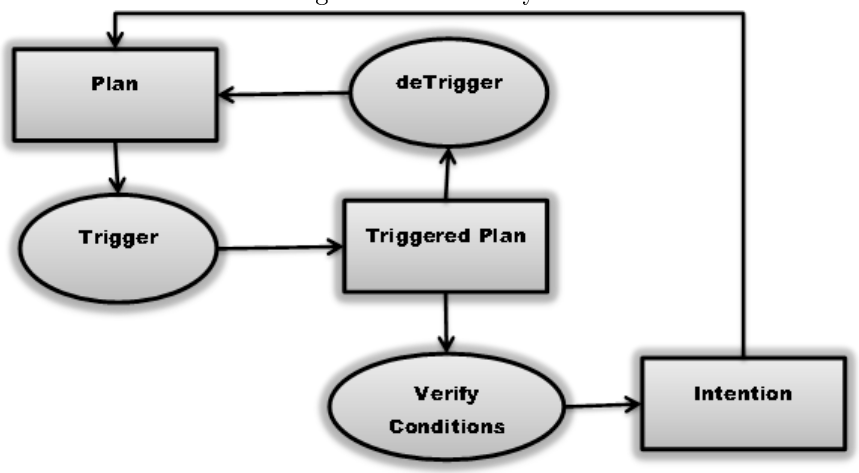
A plan can be in three different states:

- dormant;
- triggered;
- active.

When a plan is triggered the desire becomes an intention, then, if certain conditions are verified, an action is scheduled to be executed.

Each triggered plan's conditions will be verified in every reasoning cycle until they are met or the plan is detriggered. Once a plan is active it will carry out its activity and then return to a dormant state.

Figure 5.4.2: Plan cycle



5.4.5 Messages

Agents can communicate with others using Messages: these, once created and sent, will be deposited in the inbox of the respective receivers.

A Message has four elements:

- Sender, containing the ID of the agent sending the Message;
- Receiver, containing the ID of the agent to which the Message will be sent, there can be multiple receivers;

- Type of message
- Content

Table 5.1: Types of messages inspired by AgentSpeak [5]

Type	Description	Content
Achieve	Triggers a Plan in the receivers planbase	Name of the Plan to trigger
Abandon (unachieve)	Detriggers a Plan in the receiver's triggered plans list	Name of the Plan to detrigger
Tell How	Sends a new Plan to be added to the receivers plan-base	Plan
Forget How	Makes the receiver delete a Plan inside its planbase	Name of the Plan to be removed
Tell	Adds a Belief to the receivers' belief-base or updates an existing one	Belief
Forget	Removes a Belief from the the receivers's belief-base	Name and key of the Belief to be removed
Ask	Forces the receiver to send a Tell message to the sender, containing the asked Belief	Name and Key of the Belief to retrieve

5.4.6 Modules

As said in the premise, devising the agents in a way that each of them is the composition of different modules is a good way to grant interoperability of different behaviours and creating custom responses to specific situations.

A baseline Agent module is shared by all agents and grants common perceptions, like looking for nearby entities, or common plans, like the initial presentation to other agents.

A module needed could be the one shared by all Units that enables basic movement plans, refreshes the agent's position, monitor its current status; another one that could be shared by all Buildings that initialize repair warning plans or garrison information.

Then each specific operation has its own module: for example a unit that is both a worker and a soldier is initialized with both modules.

5.5 Managing the Economy

A strong economy management is fundamental for the success of any AI in the field of RTS games, since a higher rate of resource gathering and an efficiently balanced construction plan mean higher units production rates and a stronger army.

There are different actors that participate the economic aspect of an RTS game match:

- Gatherer units or structures
- Builders
- Training structures

Gatherers represent the income in resources and since there are usually more than one type of resource, they often come in different forms. More gatherers bring more resources, but it is often important to discern which ones to build in a particular instant of the match.

Building and training is effectively how the accumulated resources are spent. Usually each structure that can be built accounts for a different advantage, like the possibility to gather faster,

train stronger units, defend an area; while training units is necessary to both increasing the number of gatherers/builders and the number of soldiers available for military operations.

In centralized AIs it is easier to implement an economic strategy since everything is managed by a single process and it is simply a case of weighting different factors and sending commands to the actors. Problems arise if the complexity of the strategy gets too high since it has to be executed in a very limited time frame, then the choice algorithm is usually dumbed down trying to find a balance between efficiency and quality.

In an agent-based AI every actor decides what to do and how to proceed, so it is necessary to devise an effective way to organize the agents.

5.5.1 Workers and Jobs

The first step is to associate the agents the actions they can accomplish. Together they create a job. Then it is important to define the conditions that are necessary for undertaking each job and the effects it has on the economy.

Jobs can then be classified based on their connection with resources in:

- jobs that consume resources;
- jobs that increase resources;
- jobs that neither consume nor increase resources.

Under the first category we'll place building and training jobs, while on the second all gathering jobs. The third category is for activities, like repairing for example, that aren't linked to the income/outcome aspect but still are to be considered in the time management of the agents.

5.5.2 Creating a Virtual Market

If every worker agent can do any job it wants without any consideration of the global effects of its actions we would have an uncontrollable economic system that acts randomly and without any coalesced purpose. Instead if we create a central system that tells every agent what to do we could fall back to the centralized solution, losing all advantages of an agent-based approach.

A possible way to influence agents, without imposing on them strict commands, is to attribute a value to each job they can undertake. The higher the value of a job, the higher the incentive for the agent to choose it. Also the value of the job must vary depending on its usefulness and the number of other workers already doing it.

The solution proposed then is to create a virtual market for jobs, in a way that each agent can see the values of the available jobs and chose the desired one. Values then can be stored in a shared system or continually exchanged between agents in a decentralized market [10].

5.5.3 Wealth and Choice

Here another concept mentioned earlier returns: should agents pick the best choice ?

Allowing workers to always pick the highest rated activity creates a system which, without external influences, could reproduce the same pattern each match, with no variation and no unpredictability. Even the best RTS players never follow the same identical plan each time they play; they can memorize an optimal initial pattern but eventually they begin to chose trying to guess the optimal solution[12].

A proposed mechanic, inspired by a real economic combination, that can be used to add a randomness factor to job selection is pay and wealth.

Classic RTSs do not usually include an option for single units to buy anything for themselves, in the proposed system higher wealth means higher choice. Every time a job is finished its value is added to the worker's wealth. Every time a worker is choosing a new job it considers not only the best valued but also the second, the third and so on depending on its wealth. The choice is made randomly and if the optimal job is not selected the difference in value with the chosen one is subtracted from the agent's wealth.

This way at the start of the match, when it is important to chose efficiently, the workers start with no wealth and have to chose the best jobs, with time their options pool increases.

5.5.4 The Stimulus-Response model

The values of the jobs need to vary to respond to both internal and external conditions. If a type of resource is getting low or missing entirely, its related gathering jobs need to be ranked higher; in the same way if the current army is poor in numbers, solder training values must be raised accordingly.

It is also important to link the job market system with the rest of the agent society, not only with workers, this way a scout spotting an enemy army can directly influence the production of more military units or defensive structures.

A system devised to influence the economy is then created following the Stimulus-Response model[8]:

$$E(Y) = f(x)$$

The objective of the Stimulus-Response model is to establish a mathematical function that describes the relation f between the stimulus x and the expected value of the response Y.

This model proved to be effective in various fields of application to relate various types of events with their effects on a given system.

In the current context each presented stimulus has a response value represented by the variation of the corresponding job's value. A stimulus that manifests the lack of a particular resource raises the value of the related gathering job by a specific value. This mechanic represents a stigmergic way of collaboration between the agents, since they don't communicate directly; the act of sending a stimulus affects the environment, in this case the job market, thus altering the conditions upon which the other agents select their next job [13, 9].

5.5.5 Self-Configuration and Job Creation

Every job is about the interaction between the worker and a game element without which it wouldn't even exist. A gathering job exists only if there is a known resource to be harvested; in the same way a construction or training job only exists if the worker has the possibility to do such activity. Then it can be said that without any worker the job market would contain no jobs.

It is necessary to design a self-configuring system that autonomously adds and removes jobs depending on their availability. It must consider the workers and the offerers, which register their request triggering the creation of a job.

Some cases of autonomous job creation and removal may be:

- a scout reports the finding of a gathering spot → creation of a gathering job of the corresponding resource.
- a gathering spot is depleted → gathering job removal.
- a new builder enters the system → creation of a building job for each buildable structure.
- a structure needs repairs → creation of a repair job, that is removed once the structure has been fully repaired.

5.5.6 Self-Optimization and Balancing

To create a market that suitably fluctuates responding to different events, but which also maintains a plausible general direction, it is important to set correctly the initial parameters; it also has to be designed with self-optimization characteristics that maintain it balanced during the match. Changing the response function of a particular stimulus can deeply alter how the AI economy fares.

This way it is also possible to give a certain "personality" to the AI economy. For example setting high response values to military production can lead to a stronger army but with weaker support infrastructure; setting low decrement values for a gathering jobs can lead to less structures and a surplus of resources.

Once the parameters are set the system then needs to auto-regulate during the match: high value jobs get chosen and decrease in value letting the neglected ones surface. The lack of a

particular unit can raise the value of its training job then the jobs needed to gather the resources required and so on.

5.5.7 Self-Healing and Hazards

The system needs to be self-aware of any anomalous occurrence, the most common are:

- a job value that raises uncontrollably high;
- a job value that sinks too deep;
- a cyclic reaction that causes certain job values to raise as a group above the rest and out of control

To solve the first two problems it is sufficient to introduce maximum and minimum values; then the random component in the choice of the jobs normally solves situations where all the jobs are flattened on the floor value or at the top.

To avoid the third kind of hazard it is important to always create an "escape route" in cyclic job responses, allowing one of the involved jobs to break the chain.

5.6 Exploration

Autonomous exploration is an important part of a multi-agent AI that uses localized knowledge. Without sending units to explore the map it would be impossible to discover new resource gathering spots or the location of the enemy bases or units, incontrovertibly negating any chance to win the match.

The agent society then needs to elect its explorers and send them away from the starting location to know what lays beyond, but also requires a method to determine the various routes to be taken, to spread the search in all possible directions and reach every location.

5.6.1 Scouts and Game Map

Scouting behaviours should be assigned to the fastest units to be effective; and the vision range is also an important factor.

An effective way to coordinate exploration between agents is to divide the map in a grid, with each cell sized slightly less than the vision range of the scouts, so that sending each scouting agent to the center of one would reveal all its contents.

If all the explorers start to visit different cells, communicating with each other the data of the ones already visited, the map's contents are gradually registered in their knowledge base. Then it is only a matter of sending the findings to the interested agents and visiting again outdated cells.

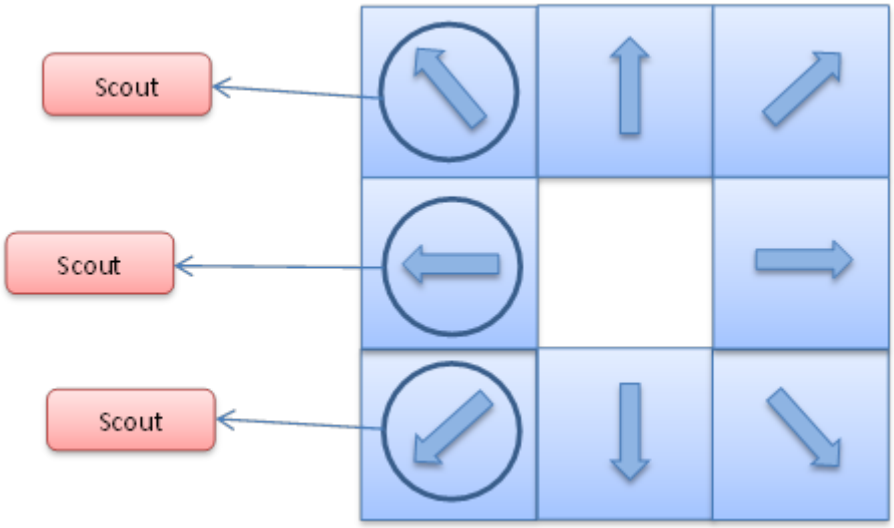
5.6.2 Self-Configuration and Dynamic Exploration Routes

Each map is usually sized differently; it is critical then to employ self-configuration mechanics to autonomously create the grid, considering the vision range of the available scouts for each match.

The starting position changes too in different scenarios and can be near the center of the map or in a corner in order to direct the exploration towards likely enemy locations, but also in order to deter scouts to retrace the others' steps, a possibility is to use a circular direction matrix.

Each explorer is then assigned a different preferred direction starting with the one directly opposite to the starting position; it then tries to explore in that direction until it finds the end of the map and then chose another. This way every route is generated dynamically and it prevents non linear itineraries.

Figure 5.6.1: Direction matrix



5.7 Military Mechanics and Warfare

The organization of military units and buildings is fundamental in pursuing victory conditions. Even though a basic strategy simply involves selecting a large enough force and sending it towards the enemy, an effective approach involves the use of a diverse array of tactics both defensive and offensive. In an agent-based AI each military unit is an agent with its own mind, but this cannot mean units attacking on their own in a chaotic pattern since it would mean certain defeat. On the other hand a strictly centralized command would mean to abandon the flexibility and adaptability of the agent system.

5.7.1 Military Hierarchy and The Top-Down approach of managing military units

An efficient way to manage military agents is to divide them using a hierarchy, attributing each agent a rank that defines its set of possible military actions and its grade of independence on the battlefield[2]. Each rank implies different objective and capabilities.

The required basic roles can be:

- Soldier
- Captain
- Commander
- General

Another concept that comes with the hierarchical system is the division of the units in different types of combat groups:

- Squads: composed of soldiers;
- Platoons: combination of two or more squads;
- Army: covering all military units under the AI control.

Every combat unit is also a soldier. They are the core that composes each squad and they have no combat initiative other than to return to base in case they are dispersed in the map alone or in case of a general alarm. Their main purpose is to follow the orders of their superiors and accomplish their given objectives.

The captain is the officer that forms and gives orders to squads. Once a captain is ordered to form a squad it tries to find among the unassigned soldiers the ones that fit the squad type and sends them formup orders. Captains have only basic missions like moving to a location, attacking a target, defending a position; they also have the responsibility to decide the current squad formation based on the situation they are in.

The commander is the first to actually have some form of independent combat initiative. In the absence of a general the commander responds autonomously to threat detection and form basic attack plans to cripple enemy bases. The force under its command is a platoon, composed of two or more squads, usually of different types based on the mission purpose. The commander tries to find an eligible captain for every squad it needs and instructs it to form a squad and get into position. The formed platoon can then be moved as a whole or divided, but the smallest group is always a squad.

The general is the only type of agent that maintains a complete strategic view. Its purpose is to evaluate global conditions and then generate combat plans to be conveyed to the commanders.

5.7.2 Military-Economy interaction

Since sometimes happens in certain RTS games that units can be both workers and soldiers, it is important to create a system that allows them to switch their current status. It is then a matter of considering actual necessities and trigger the conversion of soldiers in workers and vice versa.

Another important interaction between the military system and the economy is the dispatch of stimuli based on the lack of the necessary force. A captain that can't form a squad of the desired type will send a stimulus asking for more units of that kind. For example a general preparing an attack on the enemy base will then send a stimulus for the production of all military units or frequent attacks at the AI's base will send a stimulus for the construction of defensive units or structures.

5.7.3 Tactical Awareness and Dynamic generation of Strategies

The general needs a way to analyze the battlefield and plan movements and positioning, doing so requires a way to organize its knowledge of owned forces and bind it to their position on the map. Like with the exploration system it is possible to divide the map in a grid and then maintain recorded all information needed for military evaluations, each cell with information of ally and enemy forces, buildings, rendezvous points.

This is a way for the general to be aware of possible incoming attacks or weaknesses in the enemy lines, but it also makes it possible to utilize known strategies and contextualize them in the game map. A strategic pattern, like for example a two pronged attack, can then dynamically generate the practical strategy instantiated with current conditions and parameters.

5.7.4 Self-Protective reactions

Another crucial aspect to consider is defense and responsiveness to enemy threats.

The automatic positioning of certain units halfway between bases can constitute an early warning mechanism for incoming attacks. This also needs to activate certain alarm conditions that put the agents on alert. A base under attack can send a call to arms to defensive positions, contacting all available agents to join up.

Even during battles it is possible to devise self-protection behaviours to protect the units involved from disadvantaged situations. A concept that is rarely seen in common RTS AIs is strategic retreat: if the battle is going badly or the enemy numbers are overwhelming it serves no purpose to simply let the soldiers die, instead a simple order, from a captain or a commander, to retreat to a safe position, can change the course of the skirmish. This implicitly allows trap tactics, like when a small group escapes from an enemy contingent only to regroup with bigger forces and reverse the conditions.

5.8 Creating unpredictability

5.8.1 Introducing Random elements

As said before a way to create non-repetitive behaviours is to give agents the power of choice. But this isn't simply applying a random factor to selection algorithms, but also creating situations where multiple options arise and it isn't clear how to compare them [12]. A way to do this is to introduce discordant intentions that negate each other randomly or introducing a random condition inside certain plans to avoid or accelerate their activation.

5.8.2 Courage and Morale

A simple mechanic that allows to experiment with unpredictability is the introduction of the courage factor. Each combat unit is assigned a random courage value at birth, then every time it finds itself in a harmful situation, like during a battle, it checks its status against its courage and decides to flee or to remain on the battlefield.

If the decision is made considering the difference in forces between allies and enemies, the flight of a single fearful unit can have a chain effect causing all the party to flee.

Part III

A practical application

Chapter 6

0 a.d. : an Open Source Historical RTS game

6.1 Game elements and goals

0 A.D. is a free, open-source, cross-platform real-time strategy (RTS) game of ancient warfare. In short, it is a historically-based war/economy game that allows players to relive or rewrite the history of Western civilizations, focusing on the years between 500 B.C. and 500 A.D. The project involves state-of-the-art 3D graphics, detailed artwork, sound, and a flexible and powerful custom-built game engine.

The game has been in development by Wildfire Games (WFG), a group of volunteer, hobbyist game developers, since 2001.

6.1.1 Civilizations

The game allow the player to control any of six ancient civilizations from the pre-common era:

- The Roman Republic
- The Carthaginian Empire
- The Celtic Tribes
- The Hellenic States
- The Iberian Tribes
- The Persian Empire

Each civilisation has a unique set of buildings and units, and has notable strengths and weaknesses that must be learned and exploited by the player to be used effectively.

6.1.2 Units and Buildings

Each civilization has its own set of units and buildings but the main categories remain the same.

The units are divided in:

- Infantry: worker-soldier units that move by foot and have low combat abilities; they can be both melee and ranged.
- Cavalry: fast mounted units; they can be both melee and ranged.
- Champions: these constitute the "professional" soldiers, incapable of gathering or building, they are elite foot soldiers; they can be both melee and ranged and need a special building to be trained.

- Heroes: these unique units represent historic figures and are the best units available, they project an aura around them that benefit nearby units; they can be mounted and both melee and ranged.
- Support units: special units that perform specific tasks; female units gather and build, healers can restore other units hitpoints.
- Mechanical units: these are the siege machines, rams, ballistas, trebuchets, they are used to bring down buildings faster or to kill a great number of units from distance.

Buildings vary greatly from one civilization to the other; especially training buildings for advanced units like heroes, champions and siege machines, come with different names and purposes.

The structures common to all civilizations are:

- Civil centers: these represent the player's base, they are a resource dropsite and the starting building with which each player starts the game.
- Houses: each house built grants an increase of the population pool.
- Mills and Farmsteads: these are resource dropsites, the first for Metal, Wood and Stone, the second for Food.
- Towers: defensive buildings that shoot incoming enemies from a distance.
- Fields: these are buildable Food resource spots.
- Temples: for training healers and healing garrisoned units.
- Stables: these allow the training of cavalry units.
- Barracks: these allow the training of infantry units.
- Fortresses: defensive buildings that also allow the training of champion units.

6.1.3 Resources and Economy

There are four types of resources in the game:

- Food: collected by hunting animals, plowing fields or picking wild berries.
- Wood: collected cutting down trees.
- Stone: collected from stone mines.
- Metal: collected from metal mines.

Each of these resources needs to be gathered from the appropriate point and then returned to a dropsite.

The maximum number of units permitted at any given time is binded to the population resource, there are certain buildings, like houses, that once built raise this cap allowing the production of more.

All infantry soldiers and a female support units can gather each resource with varying degrees of speed. they constitute the bulk of the player's units and are also capable of constructing buildings. It is important to have a good amount of these worker units since they are the key for increasing resource supplies and expanding the player's base.

6.1.4 Warfare

Units constitutes the forces that the player can field to attack the enemy. Once selected one or more units can be commanded to move to any accessible position on the map or to attack a visible enemy unit or building.

There are different formations for armies that let them move in determined patterns, like Single line, Testudo, Column, Wedge.

Each military unit is combat efficiency not only depends from its category, like infantry or champion, but also by the type of the attacked enemy. There is a "rock-paper-scissors" system in place that grants weaknesses and strengths against the other types in a circular pattern. It varies with particular units but the general idea is: Cavalry tramps Melee Infantry, Melee Infantry tramps Ranged, Ranged tramps Cavalry.

Units have different available combat "stances" that determine how it should respond to the presence/attack of enemy units. The possible stances are:

- Violent: will attack any enemy unit that wanders into vision range. Will relentlessly pursue retreating enemy units into the Fog of War for a short distance running depending upon stamina. Units in this stance tend to not hold their formation.
- Aggressive: will attack any enemy unit that wanders into vision range. Will not pursue enemy units out of vision range.
- Defensive: "Default" stance. Will attack enemy units who come within half vision range distance. Will pursue the enemy until they leave that range. If no more enemies are within range they will return to their original positions.
- Stand Ground: Will not move unless tasked to move or attack. Soldiers will not pursue enemy units and will only fight back when attacked or when their comrade next to them in formation is being attacked.
- Passive: Default (and only) stance for support units. Will not attack. Runs away from enemy soldiers in the direction of the nearest Civic Centre or Fortress. A unit with this stance, if tasked to move, will attempt to avoid contact with enemy units.

6.2 Technical characteristics

6.2.1 Pyrogenesys and Spidermonkey

The game engine, written in C++, provides the underlying structure for the whole game, managing graphics, sound and user inputs.

All gameplay mechanics are instead written in javascript and interpreted by the engine using Mozillas Spidermonkey. Unit basic behaviors like stances, moving into formation, pathfinding are all part of the game logic. It's also provided a set of objects containing the current state of the game with the conditions of all entities and all relevant data used by the game logic.

The game state is updated every 150ms, actually dividing the game into turns; it's the engine that gives visual continuity to these turns by displaying the correct movement animations, allowing the graphics to appear fluid.

6.2.2 Game Logic

The game logic provides the underlying structure to support a set of higher commands by the player: it establishes a set of possible commands and then gives them a determined result into the game. For example the Walk command is executed by analyzing the map and possible obstacles for the unit that wants to move and then a pathfinding algorithm calculates the best course.

All possible commands are provided directly with an engine call or by using the appropriate APIs.

Game logic is common between all players, humans and AIs, the difference stands in the fact that these commands are executed by the game interface in case of a human player or by the AI scripts in case of a computer controlled player.

A common AI API gives the basic functions to interact with the game entities, but a Gamestate object is also created containing all entity arrays and relevant data, with also a way to quickly access them.

6.2.3 Current AI Bots

All the current AIs are centralized algorithms that evaluate the current game state and then try to actuate a set of plans adapting to it. They are omniscient, meaning they know enemy and resource positions everywhere on the map, even through the fog of war or outside vision range.

The main AI, QBot, works by doing a cyclical analysis of the game conditions and acting on a set of plans; it also adapts to enemy movements and strengths by trying to improve its army. It builds a strong economy able to support a constant flux of soldiers and provides a strong challenge, although having a unmatchable knowledge of all that is on the map. All units and structures usually receive orders in batches matching a selected plan of action.

Chapter 7

A Multi-Agent AI in 0 a.d.

A new AI bot for 0 a.d. was created as a practical application of the mentioned mechanics and techniques.

The choice of this particular game was made for different reasons:

- Open Source license, making it possible to see and change any part of the code from the AI APIs to the game engine;
- beta development stage, it isn't still complete but all its base mechanics are present and functioning correctly, it isn't a prototype but a working game;
- the development of AIs is well supported and doesn't require changing any of the base code.

The new AI called ABot is then developed using javascript and the provided common APIs.

7.1 Game Entities as BDI Agents

The only controllable entities are units and buildings, then as soon as one of them is created by the engine and inserted into the simulation a corresponding agent is created, which directs its actions. The API gives access to the entity's logic and current state, which is used by the other AIs, but it was necessary to add another layer to manage the agent's behaviour.

Then each entity exists as:

- a model inside the game engine, used for its graphical representation;
- a logic entity inside the gamestate;
- an agent inside the MAS framework.

The MAS framework was then created embedded into the normal AI infrastructure, without the need to change any logic interaction function.

Each agent is then defined by the beliefs, perceptions and plans given at its creation.

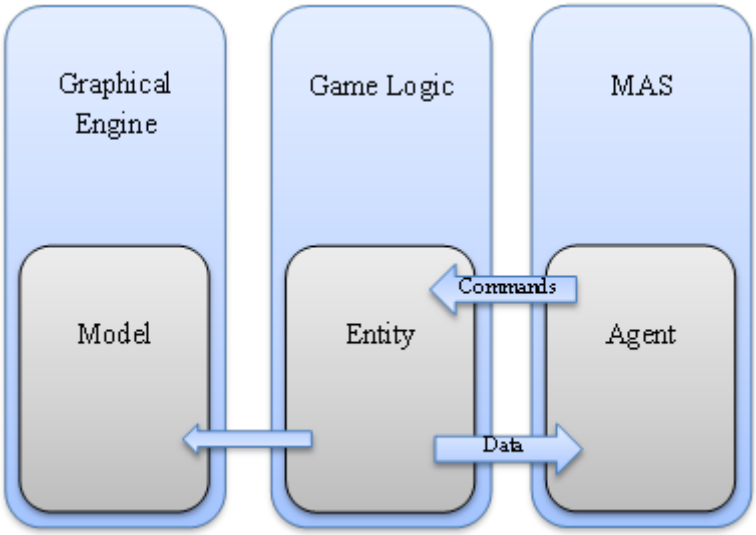
7.2 Agents coordination

Agents and entities interact on separate levels and information can only go one way from the entity to the agent that requires it. There aren't any actions that an entity can pursue that can directly affect an agent. In the same way if two entities communicate with each other at the game logic level this won't translate directly on the agent level but its effects can be retrieved, though only by the agent request.

Information from the game logic level can be retrieved only with the agent's perceptions. On the other hand every action from the agent that doesn't translate into an action on the game level is completely invisible from that level.

Since the simulation's data is only retrieved by request it is important to access key information every game turn. Like when an entity dies it is necessary to detect it immediately and begin the corresponding agent's termination routine.

Figure 7.2.1: Model-Entity-Agent interaction



Inside the MAS framework it's possible for the agents to communicate with each other using messages exchanging knowledge or plans. This is done using a "mailbox" system where an agent sends its messages in one particular turn that the receiver opens on the next, with its other messages.

Each agent is then identified by an Id that is its name in the agent society and it's derived from the entity's Id.

7.3 Workers and Trainers in a virtual Job Market

In 0 a.d. there are two types of worker units:

- citizen-soldiers,
- females.

It is necessary then to assign citizen-soldiers both the roles of workers and soldiers, also introducing a mechanic that would allow them to switch between the two during the match in order to respond to emerging necessities. On the contrary females are only assigned the role of workers.

Structures aren't designed as workers but are considered as training agents or trainers; meaning that they don't partecipate directly in the selection of working jobs but are assigned training orders on necessity.

The AI economy is then managed creating a virtual job market as seen in Chapter 6, where gathering, repairing and building jobs can be assigned to workers, while training jobs to trainers.

7.4 Game map and Exploration

Game maps in 0 a.d. are two dimensional planes where each entity's position is represented by two coordinates: x and z. They come in different sizes and feature different morphological characteristics.

In the game map there are not only the players' units and buildings, but a serie of neutral elements, owned by a fictional player named "gaia", like trees, animals, bushes, ruins, treasures. Some of these are resource spots, others are merely props.

The agents then, using their perceptions, can see and record the position of the elements they are interested in. Creating a perception for every group of similar entities it's possible to select what to see and what to ignore.

Chapter 8

Design of ABot

8.1 The AI central module and the MAS Framework

The AI's main entity is ABot, extending the Base AI interface. It's responsible for the infrastructure that interfaces the agent system with the game logic and it also provides the MAS framework.

The common API provides a Gamestate object that contains all the entities currently present in the game, with their details, their state, their current orders. Using this the AI's central module can then access all the information it needs to populate and keep updated the agents society.

The game simulation is divided into turns, in the same way the ABot module reitarates every turn to refresh the agent-base and to trigger the various steps of the agents' reasoning cycle, which is divided into:

- Agents array update: searches into the entity array for owned entities, then creates new agents; if they are missing from the current list its removes them triggering their dying method; each new agent is created with an unique Id that is the same as the one its associated entity has inside the game logic.
- Perceptions: every perception for each agent is executed.
- Mail delivery: collects all messages from the agents' outboxes, then using the recipients' ids it delivers them in the corresponding inboxes.
- Mail check: every agent checks its inbox and actuates the content of the message.
- Thinking: all triggered plans are checked to see if their conditions are met and in that case they are activated as intentions.
- Acting: all intentions are executed.

Each step is executed for each agent in the system before proceeding to the next. Even though actions are executed with the order of the agents this doesn't impact the results since all game commands are all activated at the same time at the end of the game turn. The only precaution needed creating plans is to avoid conflicting orders activated in the same reasoning cycle.

Figure 8.1.1: Reasoning cycle

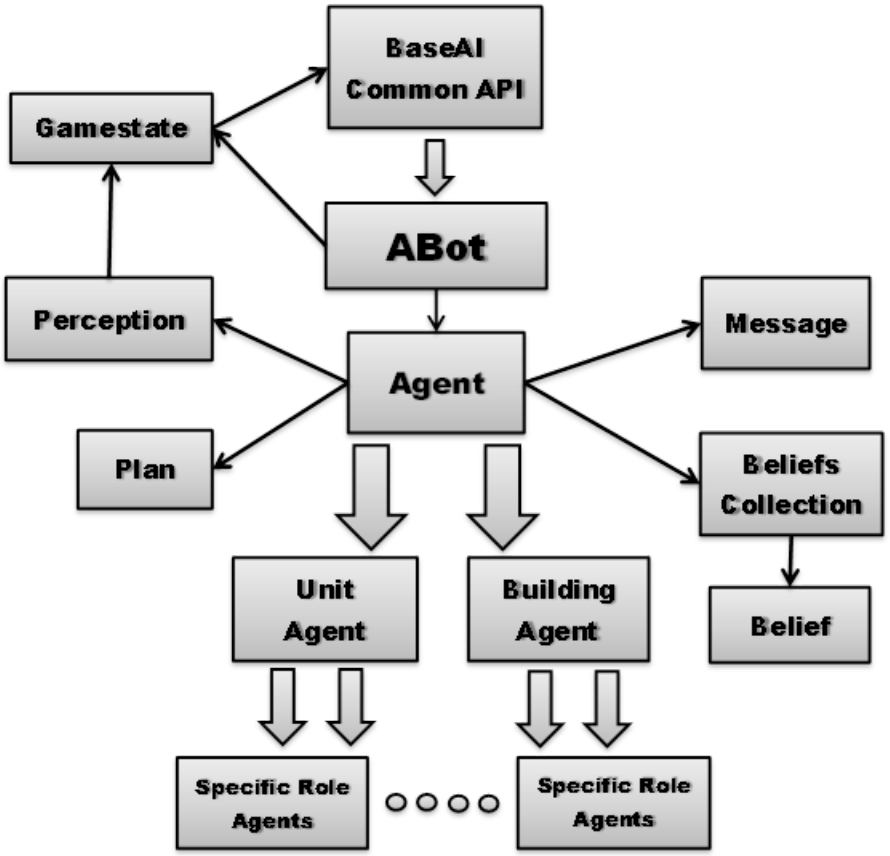
The Agent base module contains all the methods needed by the framework all common beliefs, perceptions and plans.

Each agent has its own knowledge bases:

- Belief Collection
- Plan-base
- Perception-base
- Message inbox
- Message outbox

Once a new entity enters the system the ABot central module analyzes its template and determines automatically its roles, activating the necessary modules. All agents inherit the agent module, then units the Unit Agent module and buildings the Building Agent module.

Figure 8.1.2: Framework elements and modules



8.2 Behaviours as Modules

Dividing behaviours into different modules is a way to promote reuse and standardize common actions and reactions. Analyzing an entity object it's possible to extrapolate all its characteristics and classify it into different roles, then an initialization module is loaded for each.

The system's modules are:

- Agent: behaviours common to all agents;
- Unit: behaviours common to all units;
- Building: behaviours common to all buildings;
- Worker: behaviours for worker units;
- Soldier: behaviours common for all combat units;
- Captain, Commander, General: rank specific behaviours for combat units;
- Training Building: behaviours for buildings that can train units;
- Scout: behaviours for units that can become explorers;

- Building specific modules: some special buildings often have their own dedicated module, like civil centers, fields, depots.

8.3 Optimizations and coordination agents

All the AI's code is embedded into the game logic and therefore written in javascript and interpreted, moreover the game engine still doesn't support multithreading. These conditions aren't ideal for optimal exploitation of MAS's performance bonuses, but still the biggest bottleneck is accessing the entity array in the gamestate. Every time an agent perceives something it has to iterate an array containing all the entities of the game, draining a consistent amount of resources if done every turn by tens or hundreds of agents.

A way to optimize this aspect is to divide the global entity array into specific categories already in the ABot module, allowing then the agents' perceptions to only cycle through smaller arrays. Another significant solution is to fractionize as much as possible perceptions and keep the ones that aren't needed inactive.

Another cause of concern is the amount of communication between agents while distributing knowledge. Keeping all agents updated on the position of all objects found in the map during exploration or seeding all job values every time one change, these are all situations where the messaging system, that isn't in an optimal infrastructure, can seriously impact system performances.

Coordination agents are a solution to this problem: immaterial agents that haven't a game entity counterpart but that function as knowledge keepers or service coordinators for different mechanics.

The agents created for this reason are:

- the Job Market Agent that coordinates the interaction with the job market and servers as a broker for jobs;
- the Exploration Registry Agent that keeps a record of all entities seen while exploring and directs the scouts around the map;
- the Build Planner Agent that is responsible for finding locations to place buildings;

8.4 The Job Market

The Job Market is where workers chose what to do, if gathering wood or food, if repairing a building or constructing a new one. The citizen units are independent in their choices but they are directed towards the jobs which are most needed.

The aim of this mechanic is to be:

- responsive to dynamic conditions
- flexible
- efficient
- realistic
- adaptive
- self-configuring
- self-healing
- self-optimized

Every agent in the system can influence the market manifesting needs that will influence the market but the main elements managed are:

- workers

- training buildings
- resource gathering
- constructions
- repairs

8.4.1 Agents and Elements

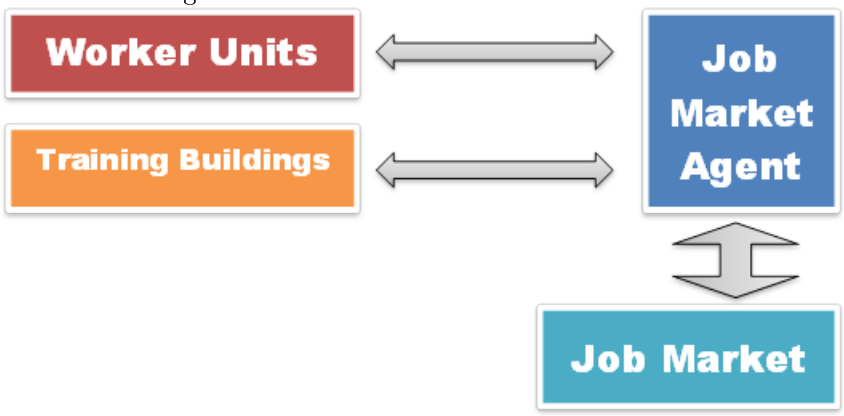
The main agents and elements in this systems are:

- the worker units that work on jobs like gathering, building, repairing;
- the training buildings that produce units of various types;
- the Job Market a central resource where all jobs are registered with their values;
- the Job Market Agent (JMA) that is an immaterial agent that works as a broker in the job market responding to requests and interacting with the other agents.

All possible jobs and their values, all registered workers and their abilities, all current available and reserved resource are contained inside the Job Market object, all actions taken influence this simulation element. The only agent that can interact with the Job Market is the Job Market Agent, all the others need to pass through its mediation to access the market.

Job Templates and Jobs are the main elements of this system: the templates describe the various jobs and are unique, while the jobs represent the actual action to take and are specific to the context. A worker cannot take a job template directly, there needs to be an actual job created under the template.

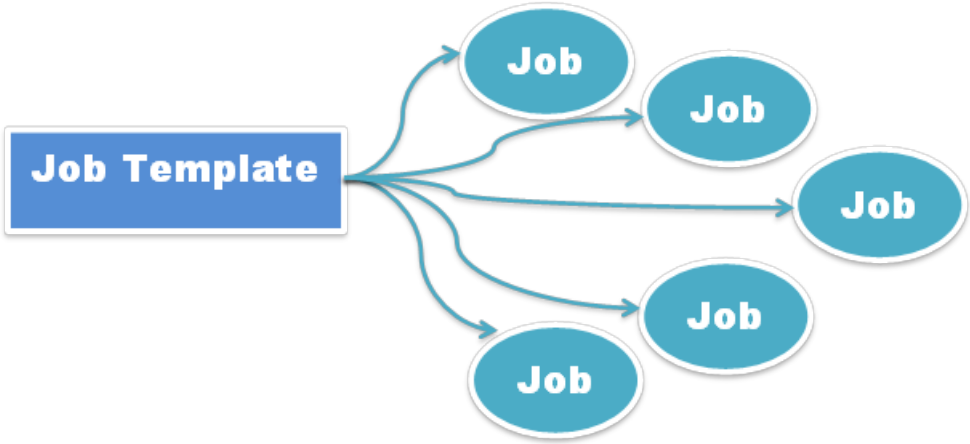
Figure 8.4.1: Job Market actors and relations



8.4.2 Job Templates

If a working action is not represented by a Job Template it can't generate jobs and then it can't be assigned. The Job Template also defines what a job could be about and its initial parameters, some can be changed once the job is initialized but they will always start with the baseline defined in the template.

Figure 8.4.2: Job Templates and Jobs



The possible Job Template types are:

- Building
- Training
- Repair
- Gathering Wood Resources
- Gathering Food Resources
- Gathering Stone Resources
- Gathering Metal Resources
- Gathering Wood Treasures
- Gathering Food Treasures
- Gathering Stone Treasures
- Gathering Metal Treasures

Under Building and Training types there are a multitude of Job Templates, one for each possible structure or unit templates that are buildable or trainable. Since some parameters cannot change from job to job inside the same Job Templates, these are shared in their related template:

- the Related Template which registers the template name of the desired production in case of building or training jobs
- the default Start Values, Maximum Values, Decrement Values, that will be used initializing the jobs;
- whether the jobs are location dependent and how big it is the penalty for distance
- whether the jobs cost resources and how many
- how the jobs' values should respond to Stimuli
- which worker is preferred to do a certain job
- the number of trips for gathering jobs
- how many spots the jobs should have

8.4.3 Self-Configuration

The system configures itself automatically; each worker that enters the system registers with the Job Market Agent and its inherent Job Templates are generated. Once a unit or building type is registered new ones of the same type will not trigger another template extraction, keeping the resources needed for self-configuration very low. This mechanic also permits to gradually integrate more complex jobs (mainly training Jobs) into the system.

Gathering jobs are auto-generated using reports from the Exploration Registry Agent: each new report is sent to the Job Market Agent and a new Job of the corresponding template is created, with the location and id of the gathering element. Workers will also automatically check whether a resource gathering spot, associated with a job, has been depleted and then inform the JMA, who will take charge in removing the job from the pool.

Fields are buildings that provide a steady Food gathering spot; once they are constructed they register and an appropriate job is created with a greater number of job spots.

Repair jobs are also automatically created from requests by damaged buildings or foundations: when a structure's hitpoints lower under a certain threshold a repair need will be sent to the JMA and an appropriate job will be added with a number of spots proportional to the damage. This also works for building foundations; since they start with 0 hitpoints they will trigger a job creation and attract workers.

8.4.4 Value, Wealth and Choice

Every Job is associated with a value that determines how important it is and how likely it will be chosen by a worker. Each worker unit also has a certain Wealth value that represent how much work it has done in the past and how well paid were its jobs.

Once a worker unit is free to start a new job it will make a request to the JMA, which in turn will determine to which jobs the worker is allowed and pick a handful of the best valued ones. To follow the concept stated earlier, that some randomness is better than always chosing the best optimal solution, the highest value job won't be picked straightforwardly, but the choice will be done between a certain number of jobs. The highest valued job will be selected, then its value multiplied by a fixed Wealth Factor thus creating a threshold. All jobs that rank higher that the threshold are selected and a Wealth Cost is associated to them based on how distant they are from the best job. Then the worker unit's wealth will be checked against the best jobs list and removed the ones that it cannot afford and finally it will chose randomly between the remaining ones.

This system allows a certain flexibility and randomness to the job selection process that will vary how each match will be played. The worker choosing a job with a wealth cost will see its pay diminished by the same cost even lowering it below zero. Since the number of possible jobs usually increase during the match, the increasing workers' wealth grants a comparable rise in choice flexibility. Once a job is picked by a worker its value immediately decreases based on its own Decrement Value.

8.4.5 Job Spots

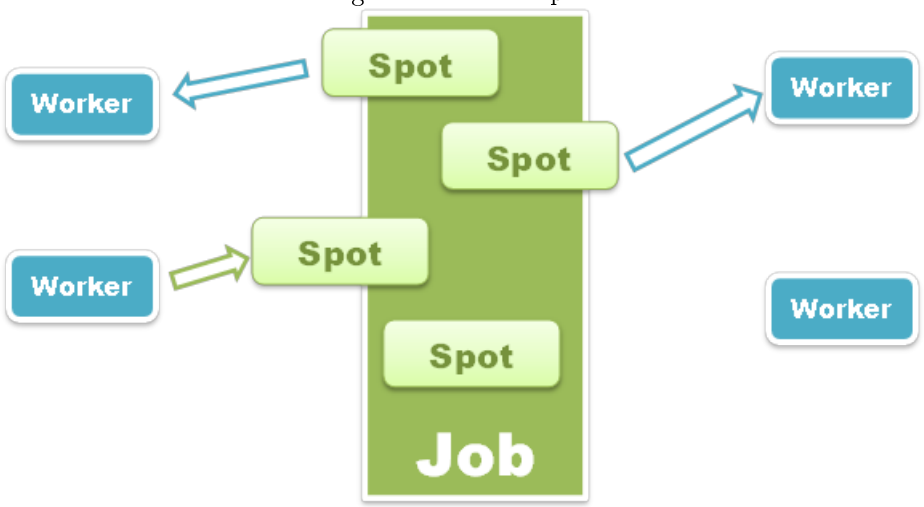
Each job has a certain number of Spots, meaning how many workers can do the same activity at the same time. The number of spots varies based on the type of action performed and how necessary it is to impede "crowding".

For example, since trees have small wood resource value and their collision sphere around them in the engine is small, they will always have only one worker at maximum; while a stone quarry that contains thousands of resources can allow six workers at the same moment.

Repair jobs configure their spots automatically based on the quantity of repairs needed.

Once the worker has selected its job the JMA will be alerted and its spot reserved. If the worker stops for whatever reason like finishing the job, wanting to change, picking up arms or even dying, it will communicate with the JMA that will restore the spots or remove the job entirely.

Figure 8.4.3: Job Spots



8.4.6 Resource dependent Jobs

Jobs are divided in two categories based on their having or not a resource cost. Since the pools of resources are shared between all agents an arbitration system was devised to protect against concurrent use.

Every Job Template that has an associated cost is considered a resource dependent job and inserted into a list that is entirely separated with the non-resource dependent jobs.

In the list of resource dependent jobs there are all possible Training and Building templates and to each it is assigned a value that determines its importance.

Once every reasoning cycle all these jobs are sorted by their value and the highest one is picked.

If its resource cost is affordable:

- the job is activated
- its resources are reserved from the available pools
- its value is decreased by its decrement value
- the list is resorted
- a new job is picked

This continues until a job with an unaffordable cost is found; then the whole process stops: the picked job's value is decreased a little to avoid stagnation and a Stimulus is generated for every missing resource.

8.4.7 Training Jobs

Training Jobs approved in the previous mechanic are directed to the buildings that can start them, choosing by the shortest training queue. If there aren't any trainers free to complete the job then its value is decreased by half the amount it would have been lowered if it was completed; also since no resources were spent the job selection cycle continues.

The lack of trainers is manifested with a Stimulus to the market.

8.4.8 Build Jobs

Build jobs are also resource dependent jobs, but they have to be initiated by worker units and therefore inserted in the non-resource dependent jobs' market; so if one of them is approved for construction it creates a new Job with the same value it had in the parallel job market waiting to be picked like other jobs.

Since it is important to check where to build structures, a complete map analysis is needed to find the best possible coordinates that are free of any obstruction. For this purpose a Build PlannerAgent (BPA) is used, its purpose is to interact with the game map's related functions and find a suitable location.

While a worker is picking up a construction job the JMA contacts the Build Planner Agent and informs it of the type of building and its preferred location; then the worker waits for a message by the BPA with the permission to proceed and the exact location where to build. If by any reason a suitable position can't be found the job is scrapped, if the reason was lack of space a Territory Stimulus is generated.

8.4.9 Stimulus-Response System mechanics

To adapt the game economy to the necessities that arise during the match and to respond to manifested needs from the agents a Stimulus-Response mechanic was developed.

There are different kinds of Stimuli, each with its own purpose and effect; here they are listed by name and description:

- Food, Wood, Metal, Stone, Population: are all Stimuli sent when a certain resource is lacking and they cause the appropriate gathering jobs' values to rise or the template of population buildings in case a Population need is manifested.
- Gatherer: if the gathering jobs are a lot more than the workers available a Stimulus will be directed at the training templates of worker units.
- Build: if there are many build jobs in the market that needs to be completed but there aren't enough workers.
- Training: a lack of training buildings will trigger a Stimulus to build Job Templates that are related to them.
- Territory: if the BuildPlanner Agent can't find enough space to build, this Stimulus will raise the value for the construction of Civil Centres that enlarge the territory limits.
- Scout: a lower than necessary number of scouts will increase the value of their training.
- Military: this is a general call to arms stimulus that will direct production towards more military active units
- Defence: the increase of the number of buildings near the base or the increasing military threat raise the value of defensive buildings.
- Attack: an Officer (see later) planning an attack on a base but lacking the desired units will send this kind of stimulus directed at the various offensive units' Job Templates, whose values will rise based on their efficiency in an attack plan.
- Cavalry, Ranged, Melee, Mechanical: are all Stimuli sent by Officers that reflect specific needs in their military formations.
- Heal: Stimuli for the construction of healing buildings and units.

Each Stimulus arrives to the JMA where it is applied causing a Response. A Response consists in the raising all the values of the Job Templates that are flagged for the corresponding Stimulus. Each Job Template has a list of values, one for each Stimulus it is programmed to respond, this will be added to the current value in the market. Each Job Template can respond differently to the same Stimulus. For example, if a Military Stimulus is received the value of training infantry units will raise by 1, while the one of cavalry units will raise by 4.

Figure 8.4.4: Stimuli

8.4.10 Self-Optimization in Job Market Economics

Choosing the correct combination of Start, Decrement and Response values is vital for the correct balancing of the game economy. Different combinations of these amounts can drastically change the behavior of the AI and provide with a way to implement different "personalities" for the computer player.

Once the match is started the job values will begin to oscillate in response to the game conditions and the general direction of the market will reflect every need with an appropriate change. The most needed jobs tend to surface to the top, while decreasing when picked.

8.4.11 Self-Healing

By putting a ceiling to the various Job Templates' values, depending on the type, it is possible to prevent them to raise too high and out of control. If a job receives constant stimuli during the match it could very well raise to values five or ten times the others and it would be very difficult to keep it down even with high decrement values.

The system also self-heal against starvation in the choice of Cost Jobs: a structure that require an high amount of resources, much more than it is available, could block the process for quite some time. Instead if a Cost Job raises to the top and it is impossible to fulfill, it will eventually decrease over time.

8.5 Exploration

Agents in ABot have limited knowledge of their surroundings based on their visual range and what they are looking for, while the bot only knows what its agent see or have seen. Since the agent society as a whole needs to know and keep track of where the various game elements are and distributed knowledge would hinder considerably the AI performances as said earlier, an immaterial agent, the ExplorationRegistry Agent (ERA), is responsible to coordinate exploration.

This agent has different purposes:

- keeping track of the elements seen on the map
- notifying the other agents of the information they need on request
- directing the scout units to explore the map

8.5.1 Exploration Grid

The ERA has direct access to the map features like height and width and it can also control if a certain destination is accessible. The area of the map is divided into a grid where a square is two thirds of the average vision range of scout units.

Every square is associated with an object that record its properties:

- if it is accessible
- its center position
- if the center has been reached
- if the square has been reached
- the date of the last report sent about its contents
- the type of terrain

Given a location on the x and z axis then there are functions that determine in which square it resides or if it is in the center. The center of a square in the grid isn't limited to the exact coordinates but it's a square with half the dimension centered around it. If this area can be reached then the whole square is considered accessible.

8.5.2 Scouts

A certain number of units, usually cavalry because of their speed and range of vision, is contacted by the ERA and instructed how to explore the map by giving it sequential locations to reach inside the exploration grid. Every destination given to the scout is always chosen from the accessible squares directly contiguous with the one it is in. The choice for the target location is also made by avoiding squares already explored recently by any explorer. This way each explorer will go its own way trying to discover as much as possible of the map.

If there are more than one possible undiscovered destination then it is chosen one that goes in the opposite direction of the starting position, but another scout will instead chose another direction entirely based on a round choice system.

Once every number of reasoning cycles the scout sends a report of what it sees to the ERA allowing it to compile a registry of all encountered entities.

8.5.3 Self-Configuration and Routes Generation

The number of scouts varies autonomously with the need to explore and the progression of the game; a lack of units eligible for the position will also trigger a Scout Stimulus to the Market. Every time that the ERA finds a new resource or treasure inside the exploration reports, it contacts the Job Market Agent allowing an automatic generation of gathering jobs. The exploration paths vary from map to map depending on the starting location and on the terrain features of the map.

8.5.4 Map interaction and Self-Healing

Since it can happen that a destination is unreachable, even if the pathfinding algorithm said it was, every time an explorer gets stuck, if center area isn't reached, the square is marked as unreachable and neither that explorer nor any other will ever try to reach it again. This way over time the scout units compile a map of the reachable terrain far more precise that the one given in the common libraries.

8.6 Military

The mechanics to manage the military aspect of the AI consist, as described in Chapter 6, in dividing the military units in different ranks, each with its own objectives and characteristics.

Since citizen-soldiers are very important during the course of the game, being the back-bone of the economy system, they also represent the majority of the controlled units. For this reason they are also the main force behind any player's controlled army.

While basic cavalry units are already trainable from the start, more advanced units like champions, heroes and siege engines are only available after the construction of expensive buildings along the course of the match. It's important then to assign military ranks in a way that is balanced and allows the coverage of all roles.

Not all types of units are always present during the match and the existing ones may die, it's necessary then to allow each type to be able to pick more than one role and assign a priority in their selection. Priorities are then described here as a number starting from 0 and raising with the decrease of the priority.

Table 8.1: Unit - Rank correspondence and priority

Rank \ Unit Type	Soldier	Captain	Commander	General
Infantry	0	3	-	-
Cavalry	0	2	2	-
Champions	0	1	1	1
Siege	0	-	-	-
Heroes	0	0	0	0
Healers	0	-	-	-

For example: if there are an infantry unit, a calvary unit and a champion unit and we need to fill the role of a commander and of a captain, the commander role goes to the champion and the captain role to the cavalry unit.

8.6.1 The Military Grid and Battle Tactics

The Military Grid is where the position of each squad and platoon is recorded and updated, it is a tool used by Generals to have a quick and easy way to evaluate the strategic situation and also to provide a serie of key location points where to regroup forces.

Commanders can use the Military Grid to keep track of the position of the squads under their control and to attain formup positions to transmit to the captains under their command.

The grid designed is a 3x3 matrix with a verbal definition of each cell representing its portion of the map:

Table 8.2: Military Grid

EnemyLeft	EnemyBase	EnemyRight
CenterLeft	CenterBase	CenterRight
BackLeft	BackBase	BackRight

The grid is self-configured at the start of the game using the starting position and speculating on the position on the enemy base; then the first explorer that reaches the enemy base triggers an update of the EnemyBase cell position and all the Enemy cells.

Generals once created start scanning the available units for the required military forces to enact the battle tactics that they have in their knowledge base; once a battle tactic activates the necessary orders are given to the commanders to form the needed platoons, whom in turn start selecting captains for the squads.

ABot is limited in the definition of only 3 battle tactics, but it provides the tools to easily design new ones by more experienced players. The tactics provided are:

- The Frontal Assault: this tactic simply creates a large enough army and sends it toward the enemy base.
- The Two Sided Offensive: this tactic creates two platoons and sends them to Enemy/Left and Enemy/Right positions, then triggers an attack on the enemy base at the same time, forcing the enemy to defend in two places of its base.
- The Siege Sneak: this tactic forms two platoons, one formed with cavalry units for their speed and another with foot soldiers and siege rams; while the first position itself just outside the enemy base gaining its attention, the second enters it from the opposite side and attacks the Civil Center with the rams hopefully without encountering heavy resistance.

8.6.2 Self-Protection and Base Defense

The main base maintains constant awarness of its surrounding terrain to spot any enemy incursion. If an army considered as a threat is spotted near the base a General Alarm is sound calling all the units and triggering responses based on their role:

- Non combatant workers are ordered to continue working but flee if attacked.
- Combatant workers are ordered to stop doing their jobs and return to base immediately in aggressive stance.
- Unassigned Soldiers are ordered to return to base.
- Squad assigned Soldiers are issued no orders since they are under their captains' command.
- Captains, Commanders and Generals are notified of the threat and respond in different ways evaluating the circumstances.

If a General is present it redirects orces towards the base evaluating the current threat; otherwise if there are no Generals then the Commanders and the Captains all return to base with their squad if the threat is consistent.

Once the base is secure Combatant workers are issued the order to return being workers and select new jobs.

Chapter 9

Implementation

9.1 Base Framework Modules

The base module is ABot, the one that interfaces with the common AI API and constitues the infrastructure from which all agents are created and their reasoning cycles are called.

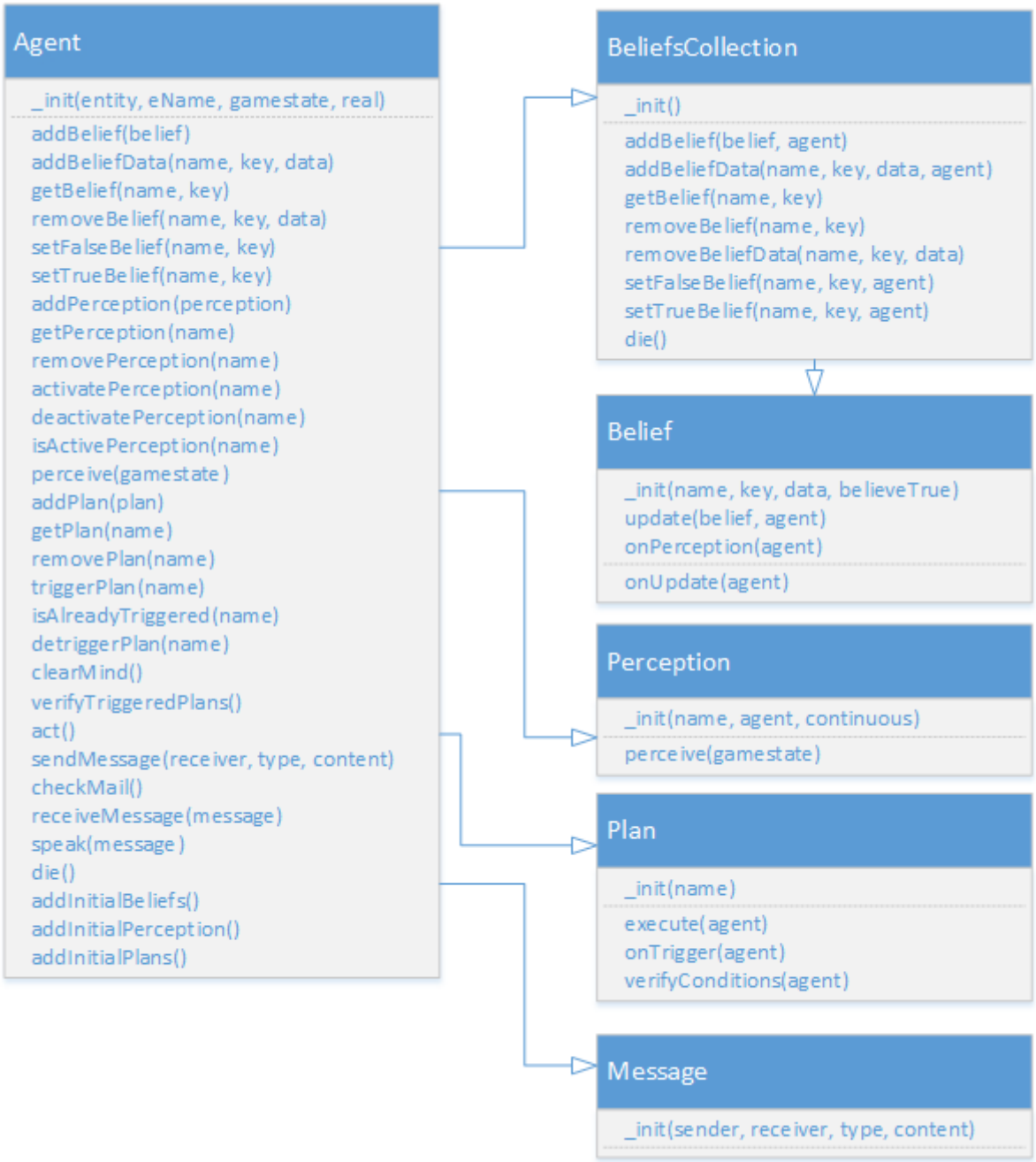
Every game turn the ABot onUpdate() function is called, within this method the following processes occur:

- if the turn is the initial one all service agents and the environment objects are created;
- the gamestate is accessed to extrapolate a list of all entities, then they are divided per type and stored as lists (the early split is due to the optimizations designed in Chapter 9);
- the current resources are updated;
- updateAgentsArray() is called: creating new agents and removing dead ones;
- lookAround(gamestate) is called cycling through all agents' active perceptions;
- mailman() retrieves all agents' messages and then deposites them in the corrisponding receivers' inboxes;
- agentsCheckMail() instructs all agents to open their messages and apply the content to their knowledge base;
- think() instructs all agents to check all their triggered plans for their verifyConditions;
- act() executes all the active plans of all the agents;

Each agent is created using the Agent class, which gives it access to all its resources like:

- Beliefs archived in a BeliefsCollection for every agent;
- Perceptions;
- Plans;
- Messages.

Figure 9.1.1: Agent Class with its interacting objects



Every agents created also shares a set of common Beliefs, Perceptions and Plans related to their interaction with the agent society and basic world interaction and perception.

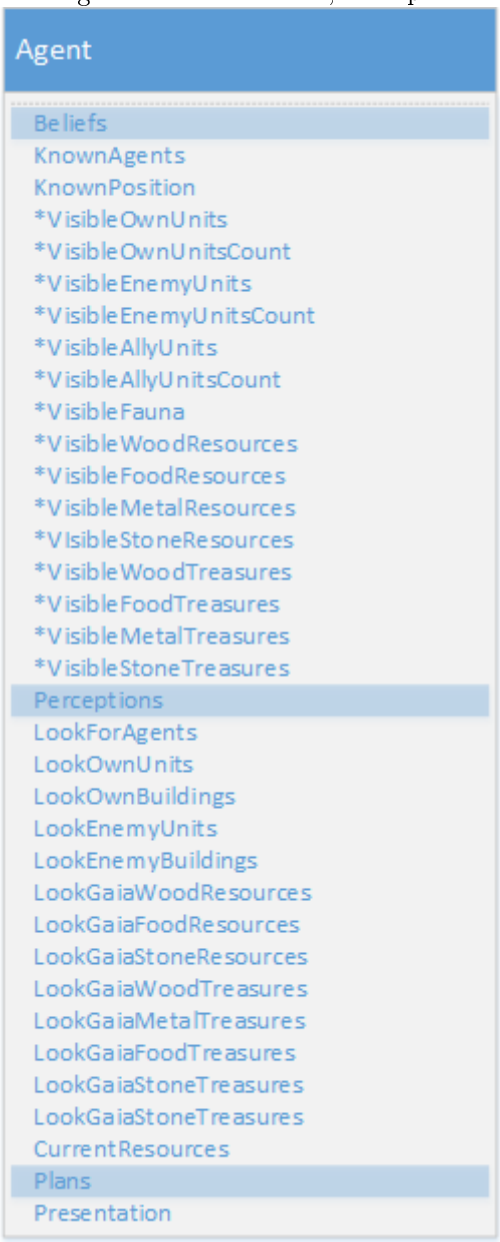
Since all units and buildings are described by a template name that is a long string difficult to interact with, the new attribute eName is created. eName is an array containing all the information derived from the template name but divided in different words:

- eName[0] can be “foundation”, “structures”, “units”.
- eName[1] is the entity’s civilization
- eName[2] is the entity’s main type, like “infantry”, “champion”, “fortress”, “house”, etc..
- eName[3], eName[4], eName[5] are the subtypes and if not used they equal 0.

It is possible to convert from templateName to eName and back with the use of the given functions.

Here are listed all Beliefs, Perceptions or Plans shared by all agents, the ones marked with a "*" in the figures are created after the agent's creation.

Figure 9.1.2: Agents common Beliefs, Perceptions and Plans



9.1.1 Optimizations

The heaviest computational costs of the AI derive from the continual cycle of perceptions that access the Gamestate object, thus cycling through a global array of entities. As said before this has been partially solved by dividing the entities into different groups in the ABot central module, thus allowing perceptions to access only the one they are interested in.

Another change done in order to boost performances was to bypass the perception framework for the most used attributes and update them in the ABot module. These attributes then are accessed directly without the need to retrieve the corresponding belief.

These frequently used attributes are:

- entity
- id
- eName
- templateName
- idle
- hitpoints

9.1.2 Unit Agent

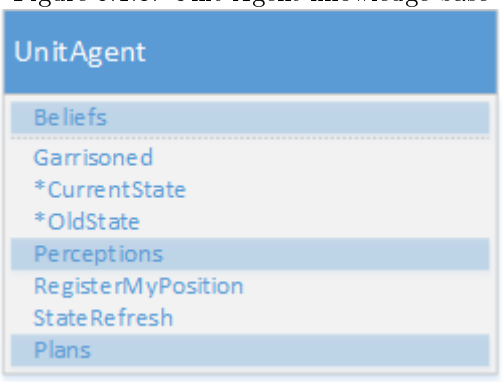
This module is shared between all unit agents, it provides some basic Beliefs and Perceptions and all their game interaction methods:

- walkTo(x,z)
- exitBuilding(garrisonId)
- attack(target)
- formation(name, entities)
- build(type, x, z, angle)
- repair(id)
- gather(id)
- garrison(id)
- changeStance(stance)

For walkTo, exitBuilding and attack there are also Group versions, that are used by squad captains to order groups of units at the same time.

Each of these commands is executed immediately overwriting any other order currently active, to put the order on queue there are Queued versions. For example walkToGroup will move all entities given in the field to the destined location.

Figure 9.1.3: Unit Agent knowledge base

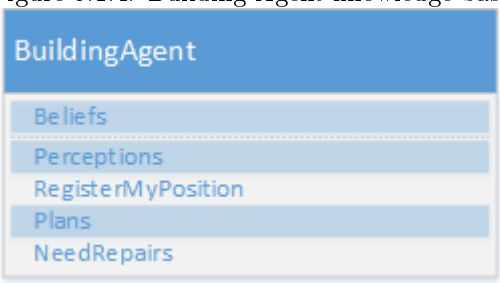


9.1.3 Building Agent

This module is shared between all building agents, it provides some basic Beliefs and Perceptions and all their game interaction methods:

- train(type, count)
- unload(id)
- unloadAll()

Figure 9.1.4: Building Agent knowledge base



9.2 Job Market

The Job Market is the entity that contains:

- all the available Jobs divided by their JobTemplates
- all the information related to the current worker templates
- a record of the current resources

There are two functions, one for buildables and the other for trainables, that once a new worker enters the system are tasked to extrapolate its possible jobs using its templateName. The Job Market then generate the new jobs and keeps recorded the worker template that generated them in order to do not trigger the process again when a new worker of the same template enters the system.

The Job Template object contains all the information common to its related jobs:

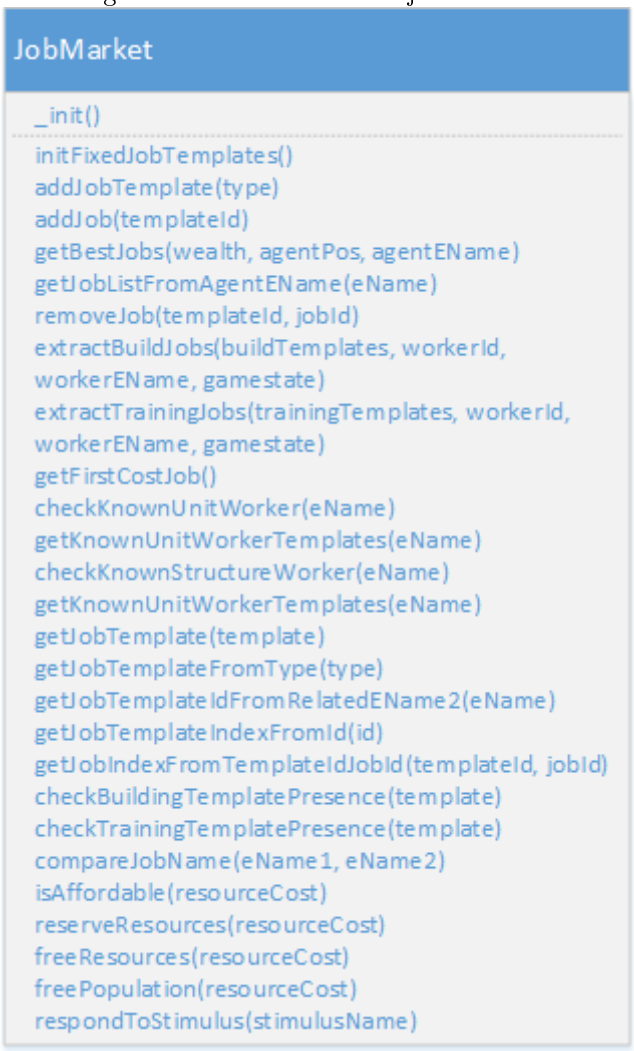
- id: this identifies the Job Template;
- type: this describes the type of job, like Build, Training, GatheringFoodResource, etc.;
- relatedTemplate: this attribute is only used by Building and Training jobs and records the templateName associated with the element created by the job;
- startValue: this value is the one given to each of job under this template at its creation;
- maxValue: this describes the ceiling for the value of the job, it is used to prevent the job to reach too high values;
- decrementValue: this is the amount that is subtracted from the the job value once a worker picks this job;
- startSpots: this defines how many workers can engage in this activity at the same time;
- trips: this is only used in gathering jobs and defines how many trips a worker needs to complete in order to complete this job: a trip is considered as the gathering of the resource and the return to the depot to drop it;

- locationDependant: this attributes informs if the job has a precise location to be enacted and if this influences the decision to pick the job;
- locationPenalty: this is the modifier to the job value that lowers it based on the job distance;
- locationPenaltyMinDistance: this value determines the minimum distance of the job to trigger a location penalty;
- preferredTemplates: this array records the template names of the agents that have a preferred for this job according them a bonus in value;
- responses: this list defines all the responses associated to this job template and how much each of them affects the value of the jobs;
- hasResourceCost and reesourceCost: these are used to determine if the job has a cost in resources and how much of each resource it needs.

Each job then is created with the following attributes:

- id: this uniquely identifies the job;
- jobTemplate: this is a reference to the associated Job Template;
- location: if the job is location dependant then this records its associated position;
- value: this is the current value of the job and it is initialized with the Job Template's startValue;
- maxSpots: this is the maximum number of agents that can work on this job;
- curSpots: this keeps track of how many available spots still remain on this job.

Figure 9.2.1: Job Market object methods

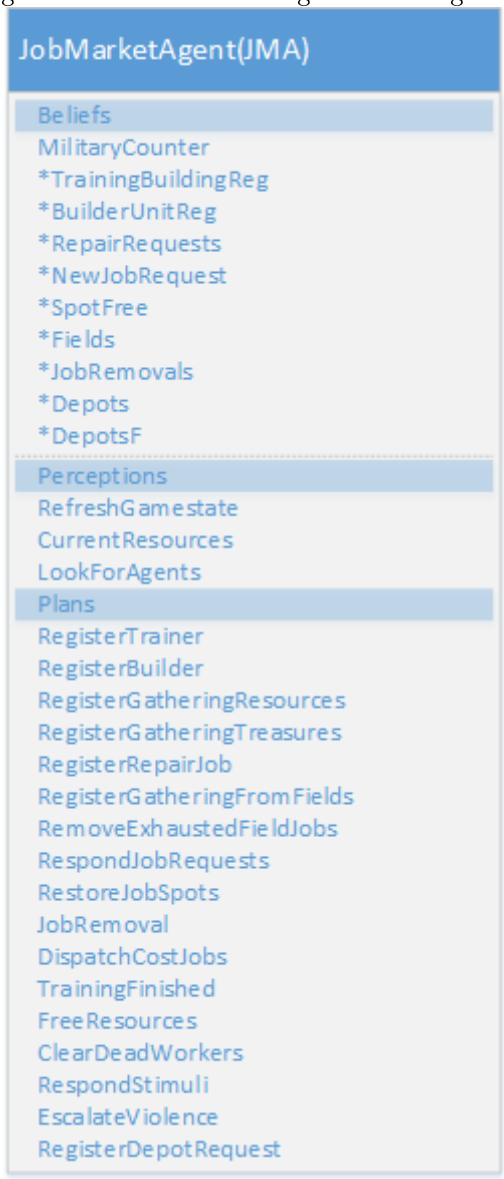


9.2.1 Job Market Agent

The Job Market agent is the only agent that can interact with the Job Market object and it is the broker that manages the various activities:

- it receives the worker presentations;
- it creates new build and train jobs based on the worker templates;
- it creates new gathering jobs based on the informations received from the Exploration Registry Agent;
- it creates new repair jobs from the requests of the buildings;
- it receives and process the various Stimuli applying them to the market;
- it manages all job management activities like restoring job spots, clearing exhausted jobs, removing clutter left by dead workers;
- it influences incrementally the market towards military production.

Figure 9.2.2: Job Market Agent knowledge base



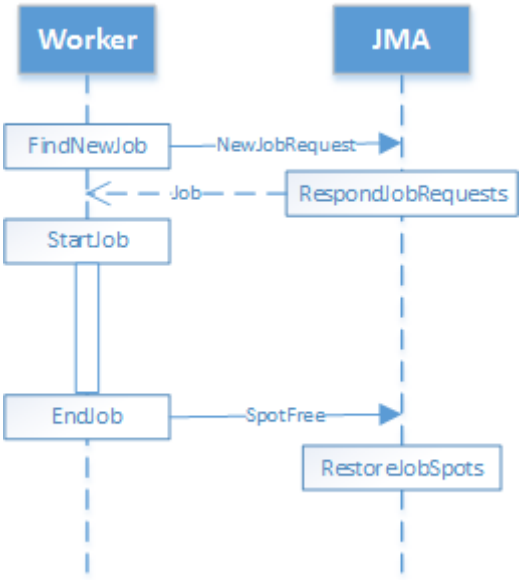
9.2.2 Worker-JobMarketAgent interaction

There are two types of agents that can take on jobs: units and buildings. Units can be employed in various activities in different locations of the map like gathering, building and repairing, while buildings can only train new units.

From this moment on we'll call working units generally as Workers, while buildings as Training Workers.

When a worker is idle and wants to start a new job it starts by asking the Job Market Agent for the available jobs providing its wealth and its position, the JMA then extracts the possible jobs and extrapolate a BestJobs list based on the evaluation done considering location penalties and wealth modifiers.

Figure 9.2.3: Worker - Job Market Agent interaction sequence for Job selection

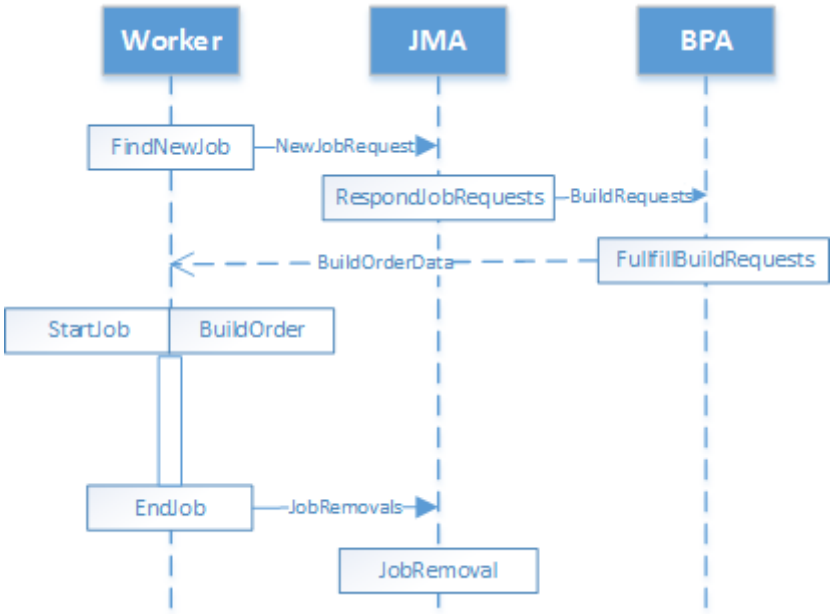


Build jobs are different from the other jobs since they involve the necessity to attain a valid position to place the foundation of the new building; a new service agent is then contacted to evaluate the best positions, the BuildPlanner Agent.

This agent mantains a representation of the game map and its elements and attempts to find a position for buildings considering their obstruction radius and the one of the other entities.

The sequence to enact a Build Job is then different from the ones of other jobs:

Figure 9.2.4: Build Job assignment sequence



9.2.3 Cost Jobs

The resource dependant jobs, from now called CostJobs, follow a different itinerary before being assigned. They are first inserted into a job list containing only CostJobs, then periodically the JMA orders the list by job value and verifies if the resource cost associated with the CostJob can be afforded.

An affordable Training Cost Job is then dispatched to an appropriate training building, selecting the one with the shortest queue. Affordable Build Cost Jobs instead are transformed into normal jobs and inserted in the job list with other non resource dependant jobs.

While selecting an affordable Cost Job triggers another job selection cycle, if the job is not affordable then its value is decremented by 1 and a Stimulus is generated related to the missing resources and also stops the cycle.

The JMA is also responsible of reserving the resources needed for the approved Cost Jobs and keeping track of their effective use by receiving CostJob completion messages by the workers and training workers.

9.2.4 Stimuli

All Stimuli are directed towards the Job Market Agent, which evaluates them and then select all the jobs that are marked to respond. Each of these jobs is then incremented in value based on the value of the assigned response.

9.3 Exploration

Exploration is managed by the interaction between the Exploration Registry Agent (ERA) and the Scout units.

There is a fixed number of callable scouts determined at initialization, then the ERA tries to contact eligible units to begin the exploration. Once contacted the ERA provides destinations for the scouts and records they reports as they are sent.

9.3.1 Exploration Registry Agent

The Exploration Registry Agent has 2 tasks:

- Recording and providing information about the existence and position of all entities seen.
- Instructing the Scouts on which locations to visit based on the data available in the Exploration Grid.

Figure 9.3.1: Exploration Registry Agent knowledge base

9.3.2 Exploration Grid generation and properties

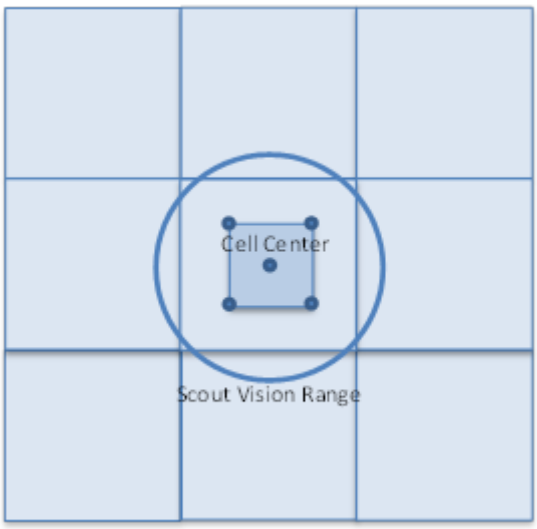
The Exploration Grid is a matrix that represent the game map divided into cells; each cell is sized by 2/3 the VisionRange of the basic cavalry unit. This is done to grant an adequate coverage of the area by the explorer reaching the center of the cell.

Each element of the matrix has the following properties:

- Accessible: is used to record if the cell is reachable;
- ExplorationCenter: this is the center of the cell or if that is unreachable then it becomes one of the auxiliary centers;
- CenterReached: true if the central area of the cell has been reached;
- Reached: true if any point of the cell has been reached;
- ReportDate: it records the turn of the last report received for this cell;
- Terrain: Land or Water, supports the recordo of terrain types.

If a destination is unreachable then the ERA tries to determine if the last position reached is inside the center area of the cell, that is a square of half the size of the main one and centered on its center; if the position is outside then the cell is marked as inaccessible and will never be selected again as a destination. Otherwise one of the auxiliary centers is selected from the 4 angles of the inner square.

Figure 9.3.2: Exploration Grid Cell



9.3.3 Scouts and Reports

All cavalry units are marked as scout units because of their extended Vision Range and their speed. A scout once called starts to accept destinations and moves around the map. When the scout starts exploring it also starts to count turns; once every a ReportRate number of turns it sends and exploration report back to the ERA containing all the entities seen until that time.

Figure 9.3.3: Scout knowledge base

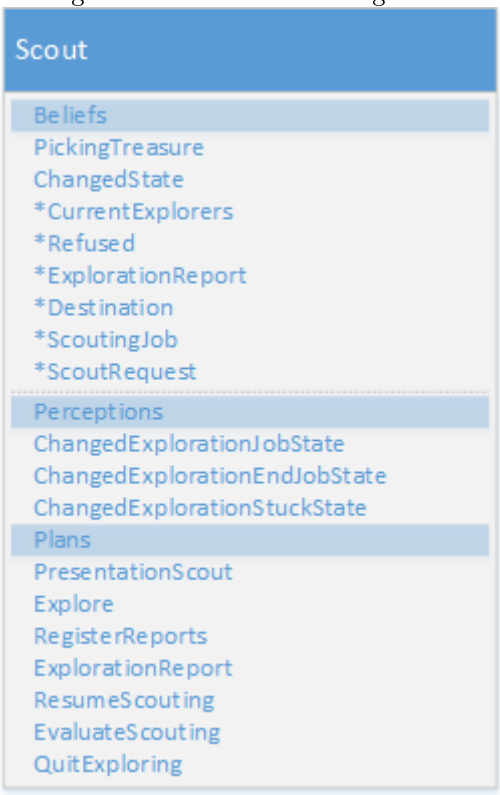
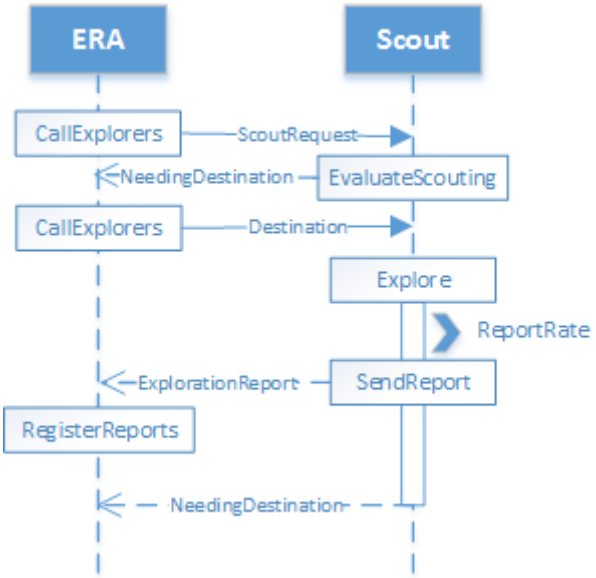


Figure 9.3.4: Exploration Registry Agent - Scout interaction sequence



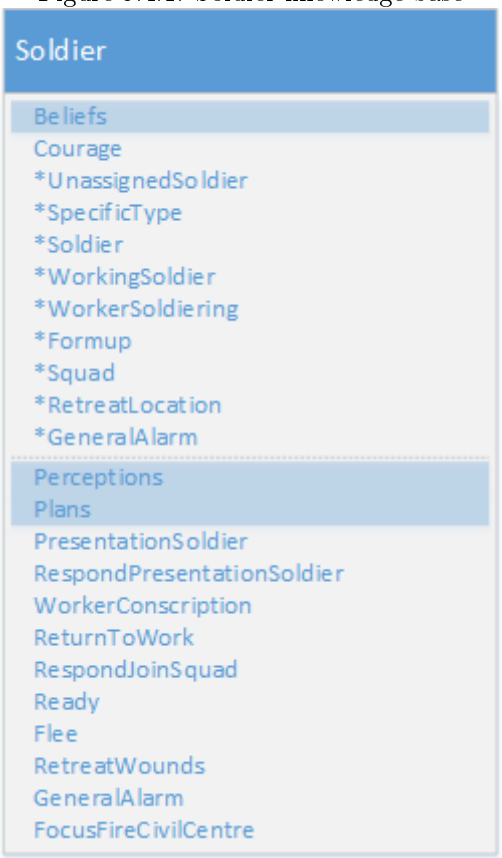
9.4 Military

The implementation of the military mechanics is done by creating sets of behaviours centered around the formation of different battle groups and their movements on the map in the response to the enactment of different Battle Tactics.

Every combat unit is a Soldier, being able to assume higher ranking positions, like Captain, Commander and General, incrementally extends its knowledge base with the related behaviours. Each combat agent is then identified with 3 different beliefs present in each agent belief-base:

- UnassignedSoldier: if a combat unit is marked as unassigned then it not in any squads and it is not in charge of any batle group;
- Type: this belief changes its name depending on the main category of the unit, like infantry, cavalry, etc..
- TypeAttack: this belief is named with the combination of the category of the unit and its attack types; an infantry unit that engages in melee is then known with the belief named infantryMelee, multiple attack types involve the creation of different beliefs, like a champion that can engage both in melee and ranged is identified with championMelee and championRanged;

Figure 9.4.1: Soldier knowledge base



Ranked agents are also identified by the beliefs CaptainAble, CommanderAble or GeneralAble depending of their main category.

Soldiers, captains and commanders can't decide on their own accord to move to different locations to engage the enemy but they have a set of responses to different situations:

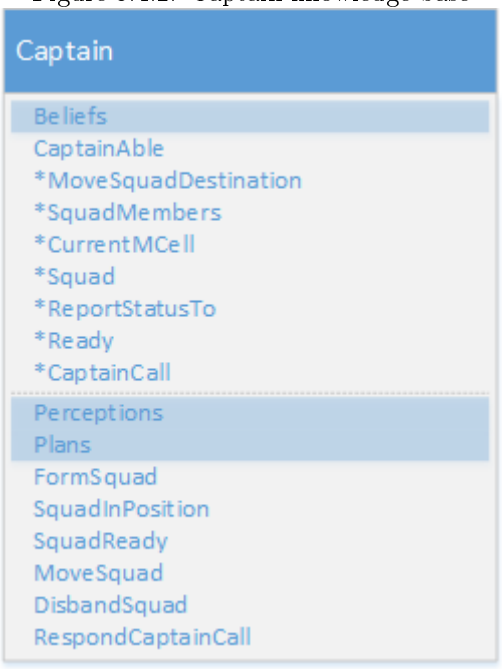
- Unassigned soldiers can respond to squad formups;
- Unassigned CaptainAble units respond to squad formup requests;
- Unassigned CommanderAble units respond to platoon formup requests;
- Unassigned soldiers respond to base general alarms;

- Unassigned soldiers if engaged in combat flee if met by superior forces;
- Squad members monitor constantly their hitpoints and flee if they reach critical levels;

9.4.1 Captain and Squad creation

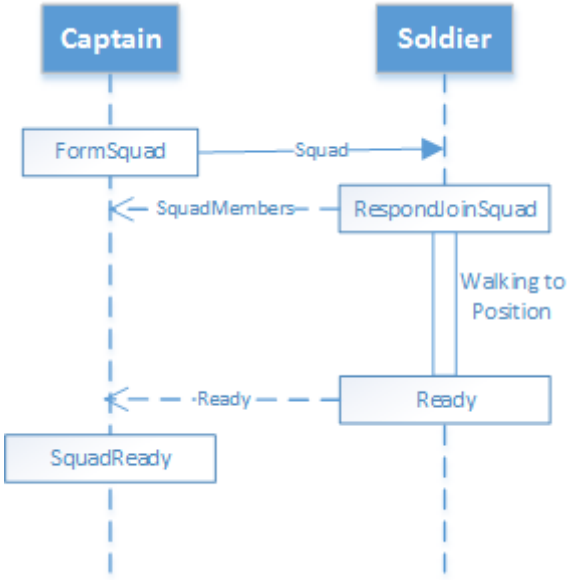
A captain is elected by a commander to form a squad and then direct its movements around the map as a unified group. All its behaviours are then limited to squad formation and movement.

Figure 9.4.2: Captain knowledge base



The squad formation sequence sees the captain calling all necessary soldiers and then reporting to it with a SquadReady message once all soldiers reach the position of the captain.

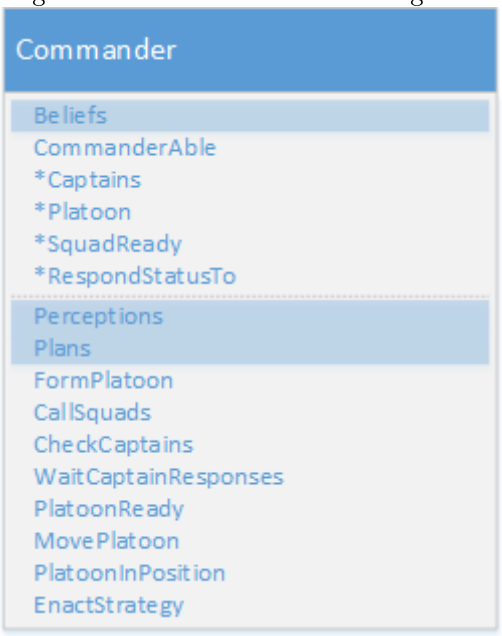
Figure 9.4.3: Squad formation sequence



9.4.2 Commander and Platoon creation

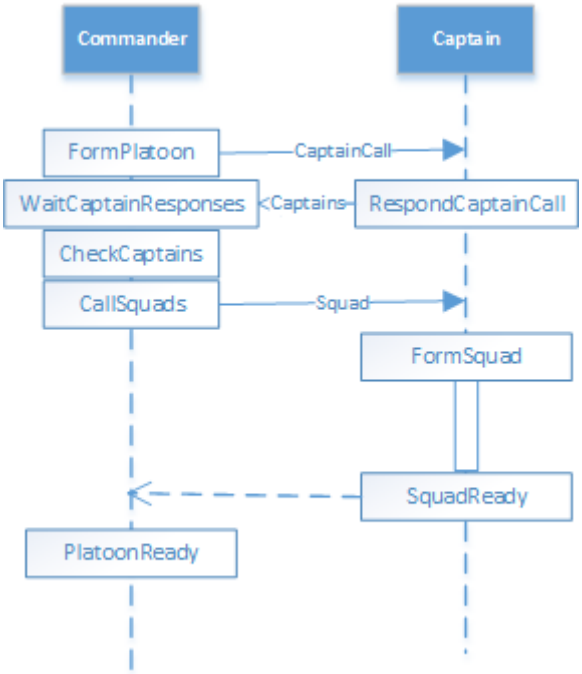
Commanders are elected by generals and are responsible for platoon creation and movements.

Figure 9.4.4: Commander knowledge base



Platoon formation consists in the selection of the necessary captains and their instruction on how to form their squads, with informations like SquadType and RequiredMembers; once all the squads are created and in position the respective captains communicate it to the commander, which in turn reports to the general who ordered the platoon formation.

Figure 9.4.5: Platoon formation sequence



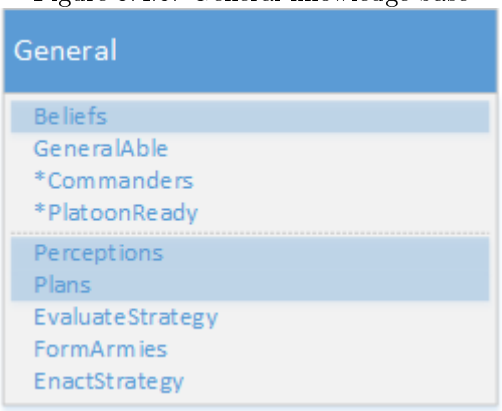
Commanders can also act on basic tactics as instructed by a general, these consists in movements

where the squads forming the platoon are directed towards different locations on the military grid. An example would be the Bait tactics which is initialized by giving a regroup location and a bait location, then a squad part of the platoon moves to the bait location and retreat once engaged with the enemy to lure them near the other units.

9.4.3 Generals

Generals are elected by the agent representing the base and once initialized continually scan the present forces to enact different Battle Tactics; once a Battle Tactic is selected the general starts to form the necessary platoons and provides them with the locations where to move.

Figure 9.4.6: General knowledge base



Every time a platoon is in position or it encounters the enemy the general responds by giving new destinations or instructing the commander to enact any of their available basic tactics.

Each Battle Tactic is then composed of a series of steps and possible reaction to different combat events, it also contains the appropriate stances and formations to use.

Part IV

Botalk: an agent-oriented Language for ABot

Chapter 10

Design of Botalk

10.1 Introduction and motivations

This section explains the creation of a new agent-oriented programming language which main purpose is to quicken and simplify the development of agents for ABot

ABot is written in Javascript to adhere to 0AD AI's mechanics and presents various repetitive constructs to access and work with the framework, so the aim of Botalk is to streamline the course of agents development and also shorten as much as possible the learning process.

Since 0AD is an open source project with a multitude of developers it is important to provide a way to quicken comprehension and to improve readability.

It is also provided a tool, developed in Java and Prolog, that converts an agent written with the new grammar into a Javascript file ready to be included into the AI.

This section then consists of:

- Conception and Design of a new Language
- Definition of its EBNF Grammar
- Development of a Tokenizer, Parser and Prolog AST tree generator using JavaCC
- Checking and Translation developed using Prolog encapsulated inside a Java application.

10.2 The language Botalk

As the MAS framework submitted in 4 was inspired by Jason, this new agent-based language follows some of the conventions introduced by AgentSpeak[5], although proposing a solution tailored to the system's necessities.

10.2.1 General Design considerations

A file written in this language is meant to describe a single module that will then be integrated with the others and selected to be started by the AI's central agent loader. The common use of a module would be to define the actions of a particular agent or common behaviors that a group of them would share.
Every script then is divided into 3 sections:

- InitBeliefs
- InitPerceptions
- InitPlans

These sections provide the knowledge base with which the agent is started and each one has very specific elements inside them; it needs to be clear in which section we are and the basic blocks, Beliefs, Plans and Perceptions, are required to be both clear to define and quick to complete.
New operators have been outlined to aid in the creation and use of the most frequently used components.
Where a new definition would take multiple lines, Botalk aims to reduce it to a single line or very small and simple sections.
Information retrieval has been the most important field for curbing and simplification; "For" cycles and "If" conditions have also been slightly modified to that end.

10.2.2 Use of + and $

Normally the use of + has been designed for the creation and addition to the knowledge base of Beliefs, Perceptions, Plans and Variables, but with Beliefs and Plans we don't always want to add them.
Since Beliefs and Plans can be created to be sent through messages to other agents it is not automatic that a new one needs to be added to the knowledge base.
$Belief, for example, only creates the belief and it can be used as a variable and sent, while *+Belief* does the same thing but also adds another line in the translation issuing a *addBelief(belief)*.

10.2.3 Init Sections

Inside InitBeliefs there can only be belief creation statements and these are always added to the belief-base, since there cannot be any messages sent from this section only *+Belief* declarations are allowed.
InitPerceptions is where new perceptions are created and added, but since not all perceptions need to be activated from the start, a command in that regard must be issued; that permits the use of an "ActivatePerception" command within this section.
The same consideration can be done for the InitPlans section where plans are created, added and then optionally triggered.

10.2.4 Beliefs

A belief is created as mentioned before with either + or $ followed by the keyword Belief and then Name, Key, BelieveTrue and Data clause, separated by a | .
While Name and Key need to be inline with the declaration, there are two possible ways to describe the other clauses: an extended form and a compact form.

Compact form:

```
+Belief:BeliefName|BeliefKey|BelieveTrue|
        { 'DataEntry1': value1; 'DataEntry2': value2; };
```

Extended form:

```
+Belief:KnownPosition|Me{
        *BelieveTrue: true
        *Data: {
                'DataEntry1': value1;
                'DataEntry2': value2;
        }
}
```

The choice between the two is non influential and it is only a cosmetic difference.
BeliefName can only be a literal.
BeliefKey could contain a value or a variable.
BelieveTrue can only be true or false, it's optional (default to true) and it allows for strong negation.
The Data clauses are always defined by a *'Label':* and then their value.
The Extended form clauses are always preceded by a * and this will be common with Perceptions and Plans that only have an Extended form.
To use a belief as a return value this syntax must be used:

```
Belief:BeliefName|Key|Data;
```

where Key and Data could be missing in case the desired value comprehend the whole set of beliefs with the same name or Name|Key tuple.

10.2.5 Perceptions

Perceptions are the means with which an agent perceive the world around it, within 0AD AIs all knowledge of map and relevant data about entities are retrievable by the "Gamestate" object.
In ABot a perception retrieves information from the Gamestate and stores it as a belief.
Since ABot implement locality knowledge and also the process of retrieving information from the game world is costly in terms of processing, Perceptions need to be well aimed and also easy to activate and deactivate as necessary.
Perceptions are created using this form:

```
+Perception:PerceptionName {
        *Continuous: true/false
        *Perceive:  Perception Code
}
```

The Continuous clause defines the behavior of the perception to continue to run every game cycle or only one cycle after activation; it's optional and it defaults to true.
The code inside the Perceive consists of the retrieval of data from the gamestate using Commands and the creation and addition of beliefs; it also allows for other constructs like variable instantiation, for cycles, if statements.
Ultimately, since Perceptions are only meant to enrich the belief base, inside the Perceive clause there cannot be plans or perception declarations.

10.2.6 Plans

Plans are the actions that agents perform and could consist of a multitude of different commands, cycles, information retrieval and conditions. A triggered plan that meets the Verify Conditions enters the execution cycle and its Execute clause runs; the OnTrigger clause's code instead gets evaluated just as a Plan is triggered. Execute and OnTrigger clauses contain all the possible expressions including the creation of new plans and perceptions. The VerifyConditions clause enclose the conditions necessary for a triggered plan to be executed and are defined in the same way as group of IF conditions.

```
+Plan: PlanName {
        *Execute:  Code expressions
        *OnTrigger:  Code expressions
        *VerifyConditions:  Conditions
}
```

10.2.7 Variables

A new variable is created using *+Var:VarName =* and it is followed by the value or other variable that will define it.

To retrieve the value of a variable it is sufficient to use its name.
Like their Javascript counterpart variables are typeless, they can store any string or number or array or object without further declaration.
If the variable is an object the way to access its attributes is by using | , while if it is an array *[Index]* will access its contents at Index position.

Where in Javascript we may write: variableName.attribute.array[index]

In Botalk: variableName|attribute|array[index]

The "." is instead used to access array properties like length in this way: variableName|attribute|array.length this will return the length of the array.

This way to retrieve data can also be used with beliefs, keeping in mind that the first 2 fields are always Name and Key.

```
Belief:Name|Key|Data|attribute|array[index]
```

10.2.8 IF

The IF statement is similar to the one in other languages with a couple of important variations.
The operators for term comparison are > , >= , < , <= , = .
The operators to evaluate multiple conditions are "AND" and "OR" .
The parenthesis to regroup conditions are "{" and "}".
Ex.

```
if( { Term1 > Term2 AND Term3 < Term4 } AND Term5 = Term6 )
```

10.2.9 FOR

The For cycle has the following syntax:

```
For( CountStartValue IncrementOperator
                CounterName
                        ComparisonOperator CountEndValue )
```

Where IncrementOperator can be >> if counting forward and << if counting backwards. While the ComparisonOperator can be the usual > , >= , < , <= .

Ex.

```
for( 0  counter < 10 )
```

Can be translated in Javascript as:

```
for( var counter = 0; counter < 10; counter++)
```

10.2.10 Commands

Commands are the way agents interact with each other and the world, the actual agent's actions; accessing the Gamestate for information also follows the same syntax.

A command follows the following syntax:

```
! CommandName : Options [ Arguments ]
```

Options identify different command with the same root CommandName and can be stacked one after the other separated by | .

Arguments are separated by a "," and are always *'ArgumentName': Argument* .

Available Commands:

- !Log[Message]
- !Speak[Message]
- !Trigger[PlanName]
- !DeTrigger[PlanName]

- !IsAlreadyTriggered[PlanName]
- !ClearMind
- !Activate[PerceptionName]
- !Deactivate[PerceptionName]
- !IsActive[PerceptionName]
- !WalkTo[X,Z]
- !Construct[Type,X,Z,Angle]
- !Repair[Id]
- !Gather[Id]
- !Garrison[Id]
- !Train[Type,Count]
- !Unload[Id]
- !UnloadAll
- !Attack[Id]

Messages and Gamestate retrieval follow a slightly different grammar:

```
!SendMessage:Type|Receiver[Message]
```

Types of messages followed by their arguments are:

- Tell[Belief]
- Forget[Belief]
- Ask[Belief]
- Achieve[PlanName]
- Abandon[PlanName]
- TellHow[Plan]
- ForgetHow[PlanName]

Receiver can either be a variable containing the Id of the receiver or the word Broadcast to send the message to all the agents.

Ex.

```
!SendMessage:Tell|FriendlyAgentId
        ['Belief': Belief:EnemyPosition|Id|Position]
```

Will send the friendly agent a belief containing the position of an enemy that will be added to its knowledgebase.

Gamestate commands are similar to accessing variables:

```
!Gamestate:Information|Attribute|Array[Index]
```

10.2.11 Assignment

Assigning a value to a variable or belief is straightforward.

```
Var:VariableName = Value
```

10.2.12 Removing Beliefs, Plans and Perceptions

With the operator "-" it is possible to remove Beliefs, Plans and Perceptions from the knowledgebase of the agent using the syntax:

−Belief : BeliefName

It is the same for Plans and Perceptions.

10.3 Grammar and Parsing

The JavaCC parser was developed following a type2 LL(2) grammar written in EBNF:

```
Agent ::= "Agent" ":" AgentName
        [InitBeliefs] [InitPerceptions] [InitPlans]
AgentName ::= <LITERAL>

InitBeliefs ::= "InitBeliefs" ":" "{" {NewBelief} "}"
NewBelief ::= ("+"|"$") "Belief" ":" BeliefName "|"
        BeliefKey ("|" BelieveTrue "|" "{" {DataClauseEntry} "}"
                                | "{" {NewBeliefBodyClause} "}")
RemoveBelief ::= "−" "Belief" ":" BeliefName [ BeliefKey ]
BeliefName ::= <LITERAL>
BeliefKey ::= <LITERAL> | "(" RetValue ")"
BelieveTrue ::= <BOOLEAN>
NewBeliefBodyClause ::= "*" (BelieveTrueClause | DataClause)
BelieveTrueClause ::= "BelieveTrue" ":" BelieveTrue
DataClause ::= "Data" ":" "{" {DataClauseEntry} "}"
DataClauseEntry ::= "'" DataClauseEntryName "'" ":" RetValue ";"
DataClauseEntryName ::= <LITERAL>

InitPerceptions ::= "InitPerceptions" ":"
        "{" {NewPerception | Command} "}"
NewPerception ::= "+" "Perception" ":" PerceptionName
        "{" {NewPerceptionBodyClause} "}"
RemovePerception ::= "−" "Perception" ":" PerceptionName
PerceptionName ::= <LITERAL>
Continuous ::= <BOOLEAN>
NewPerceptionBodyClause ::= "*" (ContinuousClause
        | PerceiveClause)
ContinuousClause ::= "Continuous" ":" Continuous
PerceiveClause ::= "Perceive" ":" "{" {Exp} "}"

InitPlans ::= "InitPlans" ":" "{" {NewPlan | Command} "}"
NewPlan ::= ("+"|"$") "Plan" ":" PlanName
                        "{" {NewPlanBodyClause} "}"
RemovePlan ::= "−" "Plan" ":" PlanName
PlanName ::= <LITERAL>
NewPlanBodyClause ::= "*" (ExecuteClause
        | OnTriggerClause
        | VerifyConditionsClause)
ExecuteClause ::= "Execute" ":" "{" {Exp} "}"
OnTriggerClause ::= "OnTrigger" ":" "{" {Exp} "}"
VerifyConditionsClause ::= "VerifyConditions" ":"
        "{" IfConditions "}"

Exp ::= IfConstruct | ForConstruct | Command ";"
```

```
        | NewVar ";" | NewBelief ";"
        | NewPlan | NewPerception
        | Assignment ";"
SingleLineExp ::= Command ";"| NewVar ";"
        | NewBelief ";" | NewPlan
        | NewPerception | Assignment ";"

IfConstruct ::= "if" "(" IfConditions ")"
        (SingleLineExp | "{" {Exp} "}") {
                ("elseif" "(" IfConditions ")" | "else")
                        (SingleLineExp | "{" {Exp} "}")}
IfConditions ::= IfCondition { (and | or) IfConditions }
        | "{" IfConditions "}"
                { (and | or) IfConditions }
IfCondition ::= RetValue (">" | ">=" | "<" | "<=" | "=")
                RetValue

ForConstruct ::= "for" "(" RetValue ("<<"|">>")
        ForCounterName (">"|">="|"<"|"<=")
                RetValue ")"
                        "{" {Exp} "}"
ForCounterName ::= <LITERAL>

NewVar ::= "+" "Var" ":" VarName "=" RetValue
VarName ::= <LITERAL>

RetValue ::= ValueExp | { ("+" | "-" | "/" | "*") ValueExp }
ValueExp ::= ((VarAssign | BeliefAssign | Command)
        ["." ArrayProp] | Value) | "(" RetValue ")"
Value ::= ["-"] <NUMBER> | "'" <LITERAL> "'"
ArrayProp ::= "length"

Assignment ::= (VarAssign | BeliefAssign ) "=" RetValue
BeliefAssign ::= "Belief" ":" BeliefName
        "|" [ BeliefKey ["|" DataClauseEntryName
        ["[" RetValue "]"]]]
VarAssign ::= VarName ["[" RetValue "]"] { "|" FieldName
        ["[" RetValue "]"] }

Command ::= "!" CommandName
        (":" CommandOption {"|" CommandOption})
        ["[" [ "'" CommandArgName "'" ":" CommandArg
                {"," CommandArgName ":" CommandArg}] "]"]
CommandName ::= <LITERAL>
SubCommandName ::= <LITERAL>
CommandOption ::= <LITERAL>
CommandArgName ::= <LITERAL>
CommandArg ::= RetValue
```

The parser was devised to produce an OO Prolog AST tree; every node is a Prolog Term and the whole output is ready to be integrated into a Prolog query.
The application that envelopes all the different steps of the process is developed in Java and uses JavaCC for tokenizing, parsing and tree generation, then imports the Java-Prolog interaction libraries of tuProlog and embeds prolog theories for checking and translating.
A prolog AST tree displayer was also developed to facilitate debugging and comprehension.

10.4 Checking

Before being handed to the translator, the parsing output is analyzed for semantic correctness; this consists of 4 checks divided into different modules where the following are evaluated:

10.4.1 Variable declaration

This check inspects all new variable statements and compares the name of the new variable with the already declared ones and report an error in case a double is found.
Since the Counter inside a for cycle is also a variable it is inspected like the rest.

10.4.2 Perceptions Perceive clauses

Since the code inside a Perceive clause of a Perception is very similar to the one inside Plans' Execute and OnTrigger, with the only difference that no new Plans or Perceptions can be created inside a Perceive clause, the most efficient course of action was to allow them in the grammar and relegate their detection into the semantic analysis.
This check then reports an error for every NewPlan and NewPerception statements inside a Perceive clause.

10.4.3 For cycle correctness

This check investigate all for cycles to determine if there are any strings used as CountStartValue or CountEndValue.

10.4.4 Commands

This is where all commands are checked to see if they are in the correct form and also if their arguments are in the correct order.
This is where possible new commands need to be added to become valid.

10.5 Translation

If all checks pass successfully the translation process begins and an output file in Javascript is created.
Looking at the examples included as attachments it is possible to see the differences between the two languages.
The output comes already formatted using a 5 space tab margin however it is possible to adjust the setting by modifying the value in the Prolog Translation Theory.

In the following extract we can see the same belief creation first in Botalk and then in Javascript:

```
+Belief:EngagedBattle|(Belief:KnownAgents|Me|Id)|true|
        {'Target': Belief:VisibleEnemies|Me|Id[EnemiesCounter];
        };

var engagedBattleBelief =
        new Belief("EngagedBattle",
        agent.getBelief("KnownAgents","Me").Id,
        { "Target" :
        agent.getBelief("VisibleEnemies","Me").Id[enemiesCounter] }
                ,true);
agent.addBelief(engagedBattleBelief);
```

The first one takes only 1 line with 110 characters, while the second 2 lines for 200 characters.

For statements are also shorter:

```
for( 0   EnemiesCounter
        <= Belief:VisibleEnemies|Me|Distance.lenght )
{...}
```

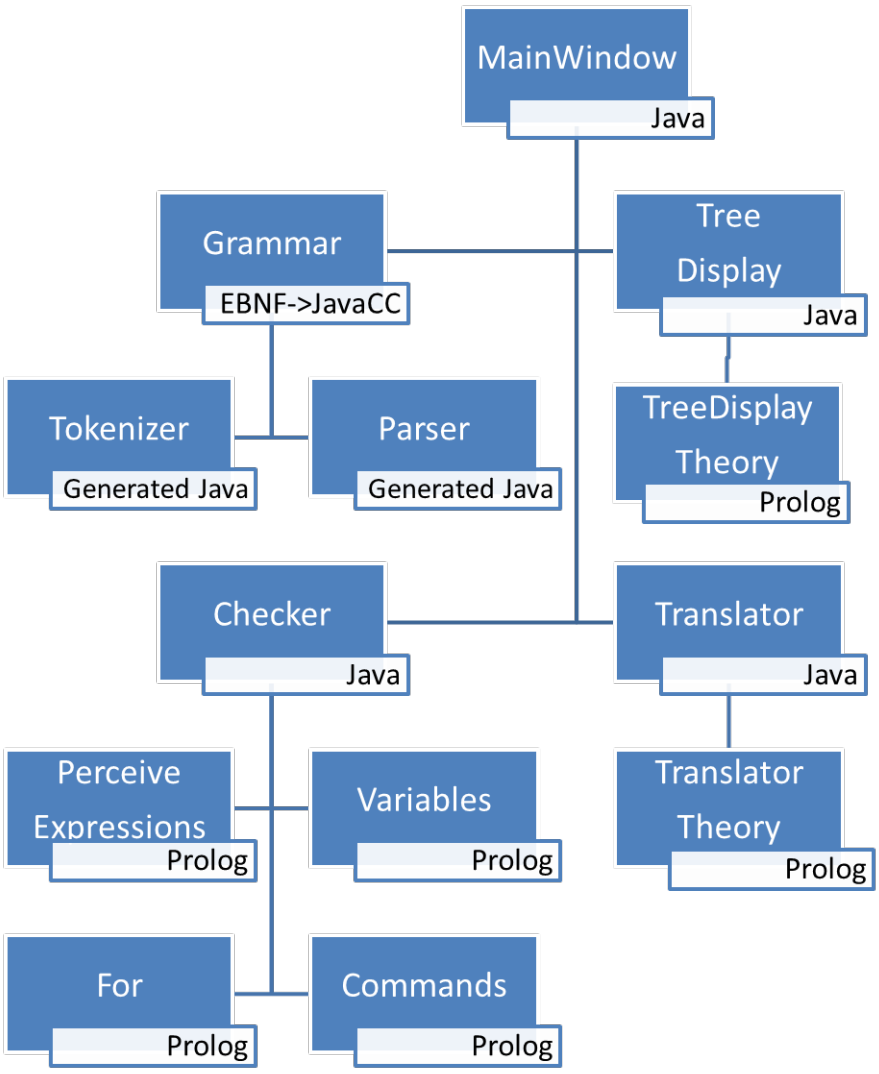
```
for( var enemiesCounter = 0;
        EnemiesCounter <
        agent.getBelief("VisibleEnemies","Me").Distance.lenght;
        EnemiesCounter++ )
{...}
```

Chapter 11

Translator Architecture

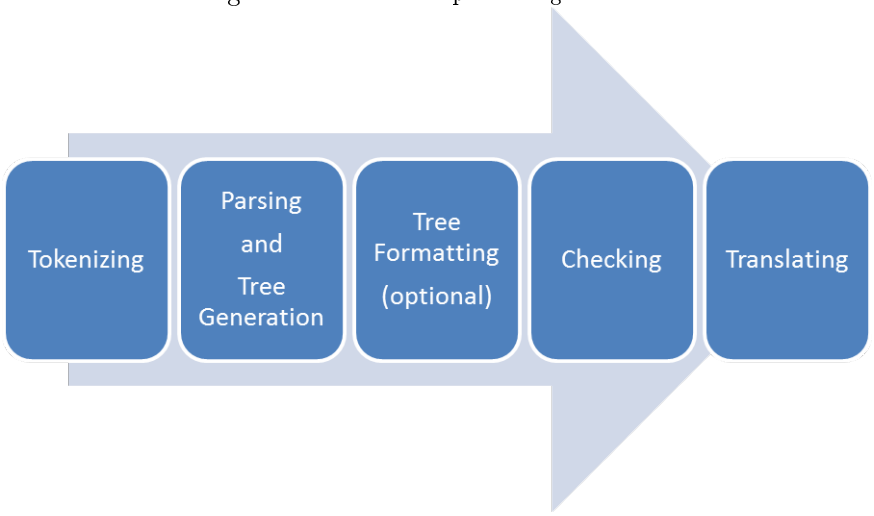
The software application is organized as follows:

Figure 11.0.1: Botalk Translator Architecture



The MainWindow manages the graphics and also calls sequentially the other modules step by step along the process; every module will only run if the previous one finished without errors.

Figure 11.0.2: Botalk processing iterations



The main is also the process that manages input and output streams, operates the various optional sections and displays the output text of all the other submodules.

11.0.1 JavaCC parser and tokenizer

The main body of the parser starts by the list of all possible Tokens and then it's a composition of the EBNF grammar enveloped within Java code.

Since the parser was devised to produce a prolog OO AST tree every production rule returns a Term object using the following constructor method:

```
new Struct(Functor, arg1, arg2, ... , argn);
```

The functor needs to be a String, while the arguments Terms.
To include a Token as an argument it is necessary to convert it to a string using the .image attribute and then into a Term.
new Struct(token.image) is a Term object.
Ex.

```
new Struct("agent",
        new Struct(agentName.image),
        new Struct("initBeliefs",initBeliefs),
        new Struct("initPerceptions",initPerceptions),
        new Struct("initPlans",initPlans));
```

Where agentName is a Token and initBeliefs, initPlans, initPerceptions are Terms; the resulting Term will be:

```
agent("agentNameString",
        initBeliefs(initBeliefsTerm),
        initPerceptions(initPerceptionsTerm),
        initPlans(initPlansTerm))
```

When the return Term is instead a list of Terms it is necessary to create a LinkedList and then to combine them with:

```
for( each Term in the LinkedList)
        listTerm = Struct( LinkedListElement, listTerm)
```

This will generate a list of Terms [elem1, elem2, ... , elemn] .

JavaCC then generates autonomously all the necessary java files necessary for analysis, tokenization, parsing, tree generation and error management.

11.0.2 Prolog and Java integration

The modules for TreeDisplay, Checking and Translating are developed using Prolog and integrating it within Java using the alice.tuprolog libraries.
Each module does the following:

- incorporates the Tree Term resulted from parsing into another Term that will form the Prolog Goal
- creates a Prolog Engine
- loads the Prolog theory from a .pl file into a Theory object
- assigns the Theory to the Engine
- assigns an output Listener to the Engine
- instructs the Prolog Engine to Solve the assigned Goal
- returns the outcome result and the output String to the Main Application

It was chosen to use the solution of the query only as a reference to the successful result of the process while using the produced string as output.
This implies that all Prolog queries only return yes or no, while the output is the result of a series of print() calls.

11.0.3 Tree Display

This module was developed to help with the analysis of the AST Tree, since the parser output is only one line of uninterrupted terms and it's hard to read and understand.
Tree Display essentially flows along all terms and prints them interleaved with tab and new line markers, trying to point out nested terms: functor(arg1 arg2(arg21 arg22))

The goal is tree(Tree) and a new counter is immediately initialized with
tree(AST, 0) , this counter will be used to accumulate spaces for indentation; in fact every type a structure is found 4 spaces are added for the nested terms.
To run through the various structures a set of rules of this form is used:

```
tree(X,I):- X =.. [ROOT,ARG1,ARG2],
        Indent is I+4, tab(I), print(ROOT) , print('('), nl,
        tree(ARG1,Indent), tree(ARG2,Indent),
        tab(I), print(')'), nl ,!.
```

There is one every possible combination of functors and arguments.
Lists are simply flattened and atoms are printed; the only exceptions are numbers and string that are labeled with a Number: or String: tag before them.

11.0.4 Checking

It was possible to check for all semantic errors with only one set of prolog rules; however during design it was chosen to separate checking into 4 distinct albeit similar modules to maintain a fair amount of clarity and modularity, allowing quicker fixes, simpler debugging and easier changes or additions.
All 4 checks have a similar structure: every node is analyzed, if the functor matches the ones related to the check its contents are inspected, otherwise they are ignored and the process flows through every node.
It's important to note that correctness control is done by matching possible errors and printing them as output, instead of giving a No result to the solution; checks both with or without errors give the same Yes solution, the difference between them is that a correct semantic outputs no error messages while the presence of a since error statement is a signal for the main application that the check failed.
Using this approach an error doesn't stop evaluation which continues along the tree to inspect all the nodes.
Every check has a set of "flow" rules aimed at opening all possible leafs of a node:

```
check(X,V):- functor(X,F,2), arg(1,X,Arg1),
        check(Arg1,V), arg(2,X,Arg2), check(Arg2,V), !.
```

There is one of these rules for every possible number of arguments, the V stands for any possible variable that could be needed in the process.

11.0.4.1 CheckPerceiveExp

Since the purpose of this program is to check for invalid statements inside a Perceive clause every time we encounter a perceiveClause node we set a flag to 1.

```
check(perceiveClause(Clauses),_):- check(Clauses,1),!.
```

This way if we encounter any forbidden node an error statement is printed.
Ex.

```
check(addNewPlan(_,_),Cont) :- Cont == 1,
        print('Error:
                Add New Plan is not allowed into Perceive Clause')
        ,nl,!.
```

11.0.4.2 CheckVar

Within this module variables' declaration and use is evaluated to avoid instantiating multiple variables with the same name or using variables without creating them first.
The initial rule initialize an empty list:

```
CheckVar(Input) :- check(Input,[])
```

This list will be used to store newly declared variables, so every time we encounter newVar(VarName,RetValue) the VarName is stored in the list.

Since counters in FOR cycles are variables they also get added to the list.

All variables names are added using the rule addVar:

```
addVar(Var,List,List):-
        member(Var,List),!. addVar(Var,List,[Var|List]).
```

It's important to note that nodes of the tree on the same level (when they are members of a list of nodes) need to be considered sequentially; so if we encounter a newVar following another when we consider the second one the var list must contain the first.
This is done using the following rule:

```
check([H|T],VarList):- functor(H,newVar,2),
        arg(1,H,VarName), arg(2,H,RetValue),
        check(RetValue,VarList),
        addVar(VarName,VarList,NVarList),
        check(T,NVarList),!.
```

After compiling the var list it is sufficient to use a member(VarName,VarList) to see if a newVar node triggers an error.

11.0.4.3 CheckFor

This is a simple check that flags CounterStart and CounterEnd values in a For cycle statement to see if they contain strings instead of numbers.
Since atoms containing strings are always inside a string(Str) Term, it is trivial to check for this kind of error with:

```
check(retValue(string(Str,_),_),1):- ...
```

Where 1 is the flag that let us know we are inside a For statement.

11.0.4.4 CheckCommands

This is where the correct use of every command is evaluated; each one has a CommandName and then a set of possible Options and Arguments whose order is rigorous.
Every time a command is found a c/3 predicate is used to compare its attributes to the knowledge base.

```
check(command(CommandName,Options,Args)):-
        c(CommandName,Options,Args),!.
```

The rest of the theory is composed of all the commands described in facts like:

```
c('SendMessage',['Forget','Broadcast'],
        [commandArg('Belief',)]),!.
```

This for example states that we can use the command

```
!SendMessage:Forget|Broadcast['Belief': belief]
```

If no command is found with the same form we get an error:

```
c(CommandName,_,_):- print('Error: format of command '),
        print(CommandName),nl.
```

11.0.5 Translation

The translation module is where the AST Tree's leaves and nodes are sequentially picked and translated to form the Javascript file.
Every node has its own form and elements to display and also a precise indentation.
The main predicate is in fact tr/2 where the first argument is the subtree still to translate while the second is a variable that store the current indentation.
The Tab value is stored in the rule

```
indentTab(I):- I is 5.
```

where the amount of spaces is changeable.
It is important to note that only expressions that start a line need to know the current tab value, so if a node is internal and all its subnodes too then it will be called by a tr/1 predicate, without the indent value. Since variable names in Javascript should start with a lowercase first letter, while in Botalk they can start with either lower or upper, every time this situation occurs the lowerFirstLetter/2 predicate will be called.

The translation is then a series of tr/2 or tr/1 predicates that explore the tree and where every term's functor is recognized in a specific rule where its correspondent Javascript counterpart is printed as output.

Ex.

```
tr(newPlan(Name,Body),I):-
        tab(I), print('var '),
        lowerFirstLetter(Name,NameL), print(NameL),
        print('Plan = new Plan('), printString(Name),
        print('');'), nl, tr(Body,I,Name), !.
```

This rule will start with the number of spaces contained in I, then if the plan name was Walk it would print:

```
var walkPlan = new Plan("Walk");
```

then continue with the plan's body where its other clauses are.

Since Body is a list of clauses of the same plan and in Javascript are separate lines we need to carry the plan's name to be able to write:

```
walkPlan.execute = function(agent){ ... };
```

Then for the translation of the body of Beliefs, Plans and Perceptions the predicate tr/3 will be used. printString simply puts quotation marks around the string passed as argument. There are some cases where the predicate tr isn't used, these happen when the structure of the tree is different from its projected translation or when we need to extract elements from a list; some context specific rules were developed for this purpouse where the most generic is expressions/2 (or /1 without indent).

11.1 Final considerations on Botalk

In the tests that followed the development of Botalk it was clear that programming agents for ABot became quicker and less confusing, without the need to worry about framework hooks and references. Writing ABot agents in Botalk is 25-35% quicker than using Javascript; but these figures are without considering auto-completion.

A development IDE or an auto-completion editor will be required to really take advantage of its features. However the easy approach still remains an high point in favor of appending the language to the release of the AI, to facilitate new developers interested in creating their own agents.

Part V

Results, Conclusions and Future Work

Chapter 12

Tests

The final part of this theses consists in the testing of the proposed application and the evaluation of the different results upon which the final conclusions are drawn.

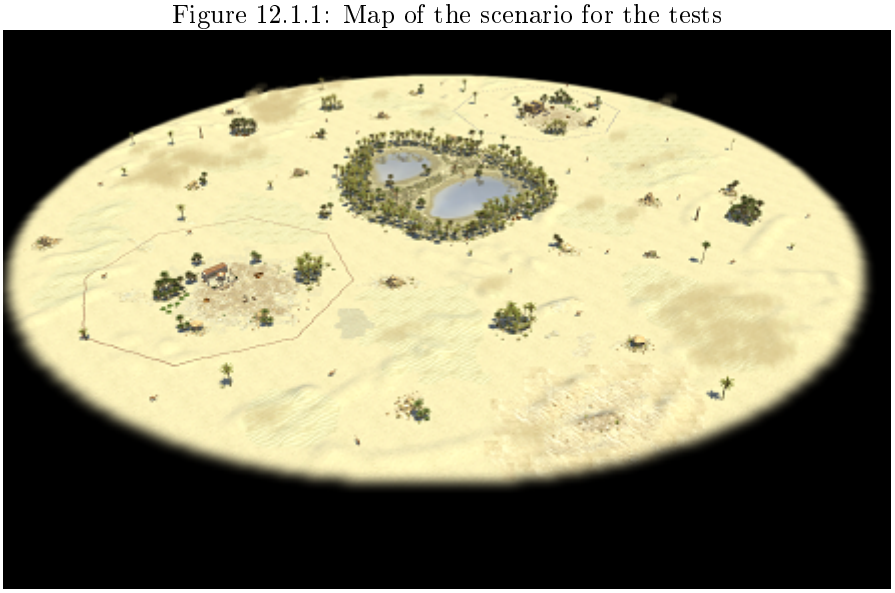
12.1 Performance metrics

The tests have been chosen to evaluate the proposed AI ABot in different contexts to see how it fares compared to the standard 0 a.d. AI, QBot.

All the tests are done using the same scenario, Oasis I, and the same playing mode, Conquest, which ends with the destruction of the enemy base. Oasis I is a scenario that pits two players against each other on different sides of a oasis in a desert setting. Each player starts with 4 basic infantry units and a Civil Centre as the base.

It is important to note that the configuration parameters set for ABot are not optimal; a deeper analysis of the initial values of the various jobs and their related stimuli would lead to better gameplay performances.

Figure 12.1.1: Map of the scenario for the tests



12.1.1 Strategy

This test evaluates how fast an AI can produce units, buildings and subsequently how well it confronts the enemy in battle. The parameters used are then contained in the snapshots of the final screen which details:

- The number of units trained and lost;
- The number of enemy units killed and buildings destroyed;
- The number of buildings constructed and lost;
- The percentage of the map explored;
- The amount of resources gathered;
- The number of treasures collected.

The snapshots are taken after 5 and 30 minutes of game time.

12.1.2 Realism

This test evaluates how the AI resembles human playing patterns considering the following areas:

- economy management;
- exploration of the map;
- management of combat units

The test is done examining an experienced player's match and confronting it with how ABot plays the same scenario. Analysing the differences in tactics and effectiveness.

The scenario chosen was Oasis X, very similar to Oasis I, and the recording of a played match was gently provided by Michael Hafer, 0 a.d. developer.

12.1.3 Performances

This test is done to see how each AI impacts on the computational costs of the system. The results are taken using the internal profiling tool of the engine which examines different aspects like memory usage and calculation time of each process.

The performance metric used is msec/frame, that is the time spent inside the examined section every frame, averaged over the past 30 frames. A snapshot of the profiler is taken every second for the first 10 seconds of playing and again another 10 seconds after 5 minutes of playing.

12.2 Test Results

12.2.1 Strategy

Here are shown the results recorded after 5 minutes of game time:

Figure 12.2.1: Screenshot of the Units/Buildings results screen after 5 minutes of game time

Figure 12.2.2: Screenshot of the Conquest results screen after 5 minutes of game time

Figure 12.2.3: Screenshot of the Resources results screen after 5 minutes of game time

After 5 minutes of game time the situation of the game is still balanced between the two AIs; although it is clear that Food gathering parameters for ABot have been set too low, the production of units and buildings is mostly on par. The two units lost by ABot are scouts that found the enemy base and have been killed by enemy defences.

The Conquest results clearly underline the efforts of ABot in the exploration the map, which can be seen in the Map Exploration percentage and indirectly by observing, in the Resources results, the 2 extra treasures taken by ABot's scouts.

As seen in the following screenshots, ABot has provided the same defensive countermeasures of QBot in building early the two towers. The other buildings present are a manifestation of the response to different stimuli: the depots are the response to workers having to walk longer to reach far resources and stables are an attempt to respond to an increasing number of military stimuli, although evidently showing a too abundant construction of this type of building.

Figure 12.2.4: Screenshot of the ABot controlled player's base after 10 minutes of game time

Figure 12.2.5: Screenshot of the QBot controlled player's base after 10 minutes of game time

Figure 12.2.6: Screenshot of the Units/Buildings results screen after 30 minutes of game time

Figure 12.2.7: Screenshot of the Conquest results screen after 30 minutes of game time

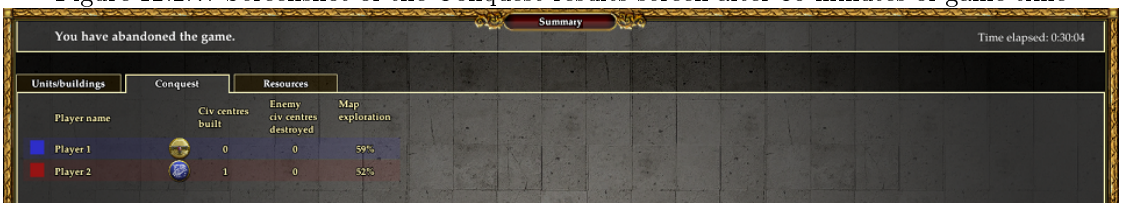


Figure 12.2.8: Screenshot of the Resources results screen after 30 minutes of game time

After 25 minutes QBot launches an offensive with an army of 20, mostly foot soldiers; the offensive is taken without any siege engine present and ends in the complete annihilation of the attacking units, but also killing more than half of ABot's workers.

The 30 minutes results' screenshots show then that the unbalanced Food gathering's initial parameters have determined a clear disadvantage for ABot, that struggled to rebuild its forces after suffering QBot's attack.

ABot focused mostly on a defensive posture, determined by its self-protection mechanism, that incentives the construction of towers in response to enemy attacks; the fact that QBot attacked first changed the way ABot managed its economy. The *Enemy unit killed* result points out that the defensive response worked and all the enemy attacks were constantly repulsed.

12.2.2 Realism

The human test player starts the match by collecting the various treasures near the starting base, exactly as ABot, as shown in the following screenshots. It is relevant to point out that QBot does not start this way but collects the treasures only after the first minute.

Figure 12.2.9: Screenshot of the first seconds of the match by the human test player

Figure 12.2.10: Screenshot of the first seconds of the match by ABot

The game progresses with the human player gathering the needed resources and building only the necessary buildings in key positions; the player is limited in only giving commands one at a time. ABot, instead, has a stronger start dictated by the independence of the actions of its units

and buildings that decide on their own what to do. Although ABot clearly has an advantage here dictated by its artificial nature, it still uses tactics that could be compared to human behaviour. The disposition of ABot's buildings is dictated by the placement algorithm and resembles the playing style of a chaotic player.

Figure 12.2.11: Screenshot of the human test player after 3 minutes of game time

Another aspect where ABot successfully emulates the human player is map exploration. Both Michael and ABot order one of their cavalry units to head out and try to discover treasures and the location of the enemy base, as shown in the following screenshots.

Figure 12.2.12: Screenshot of the human player's scout exploring the map and heading towards a treasure

Figure 12.2.13: Screenshot of one of ABot's scouts exploring the map

Figure 12.2.14: Screenshot of one of ABot's scouts heading towards the discovered treasure to collect it

The last considerations regard the tactics used to attack the enemy by both players. The human player shows patterns in selecting the composition of its attacking forces similar to ABot, but its attacks are more dynamic in selecting locations and targets. ABot uses self-configured positions to move its units around the map and this can often emulate human behaviour but it is still necessary to improve the resolution of the military grid to allow more flexible movements.

The human player is experienced and demonstrates it by never launching hopeless attacks. This is similarly done by ABot that evaluates the enemy strength by using the reports of the explorers, although sometime losing them.

The self-protective characteristic instilled in ABot's combat units by the various fleeing behaviours has proven to be effective in saving forces from hopeless attacks, as designed in section 8.6.2.

Figure 12.2.15: ABot's units fleeing from superior forces

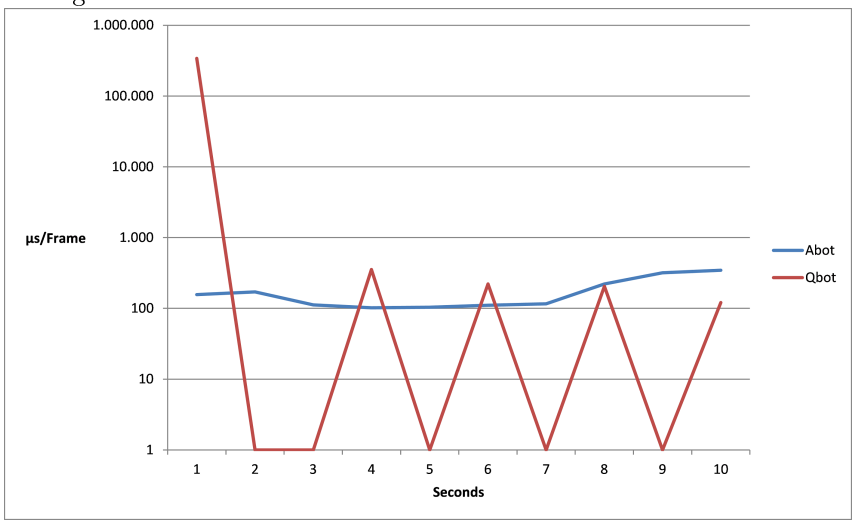
Figure 12.2.16: Screenshot of an attack group of the human player

Figure 12.2.17: Screenshot of an attack group of ABot

By looking at the screenshot of ABot's attack platoon it is possible to spot the commander, the cavalry unit, and the two squads, one infantry melee and the other infantry ranged, captained by the two infantry units in front of the formation.

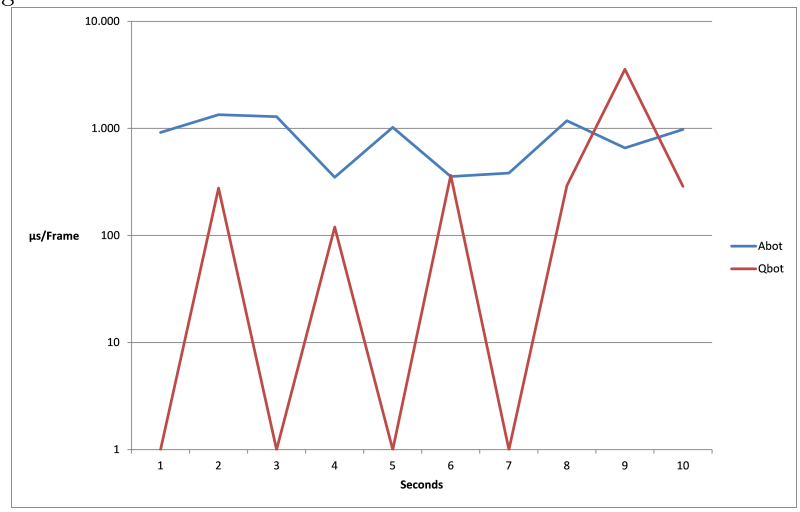
12.2.3 Performances

Figure 12.2.18: Results taken in the first 10 seconds of the match



Total amount of time spent by each AI every frame averaged by 30 frames; each snapshot is taken every second for the first 10 seconds of the match.

Figure 12.2.19: Results taken after 5 minutes from the start of the match



Total amount of time spent by each AI every frame averaged by 30 frames; each snapshot is taken every second for 10 seconds after 5 minutes into the match.

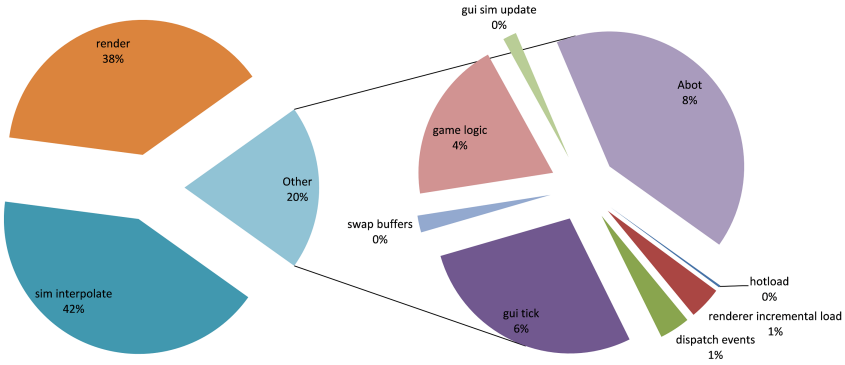
The results clearly show the characteristics that differentiate centralized AIs with distributed ones as stated in section 4.5.5.

The initial spike of QBot corresponds to its initial configuration algorithm and it is visible during play by an actual freeze of the game; although this kind of behaviour is common throughout the match, signs of performance decrease are only observable by the player after a hour of game time.

ABot doesn't suffer from performance spikes but shows a slow increase in calculation time proportional with the increase in the number of agents in play. While performances are reasonably worse than the ones of QBot, as predicted in section 4.5.6, the degrade is less noticeable by the user because of the lack of sudden spikes.

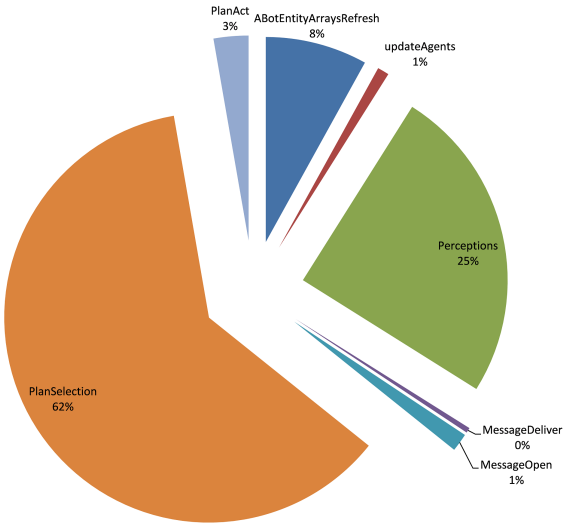
In fact if we look at the following graph, that shows the time allotted to ABot in comparison with other elements of the engine, we can see that it still remains much lower compared to the graphical component.

Figure 12.2.20: Calculation time taken by each section of the game engine



The following graph instead analyses how each section of the reasoning cycle affects the global performances of the MAS. Despite the optimizations done in section 9.1.1, the portion of time taken by interfacing with the gamestate, enacted by perceptions and entity array refresh, still comprises over 30% of the total calculation time, allowing the notion that further optimizations in that regard could greatly improve the system's performances.

Figure 12.2.21: Load division between different sections of ABot



Chapter 13

Conclusions and Future Work

Examining the results of the tests presented in the last chapter, it is possible to conclude that using Multi-Agent Systems in the creation of AIs for Real Time Strategy games is a valid alternative to centralized systems. The agent-based approach provided a greater flexibility and adaptability in numerous occasions, while still depending greatly on the quality of the initial configuration parameters. Once created the underlying agent infrastructure it was easier to create the behaviours wanted and instill in the AI a sense of uniqueness and unpredictability. The agents displayed interest in collaborating collectively towards the same goals while maintaining their independent nature.

The MAS also performed as expected regarding computational costs, keeping a linear trend over the course of the match; considering the lack of multi-threading in the engine and the use of AI APIs not optimized for this type of activity, the rise in costs over QBot remained well within anticipated parameters. Nonetheless the system stayed completely self-contained and in need of minimal maintenance to remain in par with the evolution of the game's development.

The thesis also provided the tools to continue the development of the AI and extend its functionalities by designing new agent behaviours and roles using the Botalk language.

It was clear from the tests that a better initial configuration can and need to be devised; a balanced organization is harder to reach in MASs compared to centralized systems and it will be important to involve players more experienced with the game to evaluate the various parameters.

The military mechanics provided only cover the organizational aspect of managing combat units; more complex and efficient combat tactics have to be developed to see ABot reach its full potential.

ABot represents only a first step in improving AIs for RTS games, nonetheless what was shown proved that using MASs could lead to significant advancements.

Bibliography

[1] Foundation for intelligent physical agents, 1997. http://http://www.fipa.org/.

[2] Ahmed Kaboudan Abdulla M. Mamdouh and Ibrahim F. Imam. Real-time, multi-agent simulation of coordinated hierarchical movements for military vehicles with formation conservation. 2012.

[3] Dan Adams. The state of the rts. 2006. http://www.ign.com/articles/2006/04/08/the-state-of-the-rts.

[4] Fabio Bellifemine, A Poggi, and Giovanni Rimassa. *JADE - A FIPA-compliant agent framework*, pages 97–108. The Practical Application Company Ltd.

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

[6] M. E. Bratman. Intention, plans, and practical reason. 1987.

[7] Michael Buro. Call for ai research in rts games. In *In Proceedings of the AAAI Workshop on AI in Games*, pages 139–141. AAAI Press, 2004.

[8] G. Cashman. *What Causes War?: An Introduction to Theories of International Conflict*. Lexington Books, 2000.

[9] Giovanna Di, Marzo Serugendo, Marie pierre Gleizes, and Anthony Karageorgos. Self-organisation and emergence in mas: An overview. *In this volume*, 30:45–54, 2006.

[10] Torsten Eymann. Markets without makers - a framework for decentralized economic coordination in multiagent systems. In *Proceedings of the Second International Workshop on Electronic Commerce*, WELCOM '01, pages 63–74, London, UK, UK, 2001. Springer-Verlag.

[11] Bruce Geryk. A history of real-time strategy games. 2008.

[12] Dave Mark. Choices: Not just for players anymore, April 2009. http://intrinsicalgorithm.com/IAonAI/2009/04/choices-not-just-for-players-anymore/.

[13] Leslie Marsh and Christian Onof. Stigmergic epistemology, stigmergic cognition. *Cogn. Syst. Res.*, 9(1-2):136–149, mar 2008.

[14] Andrea Omicini and Franco Zambonelli. TuCSoN: a coordination model for mobile information agents. In David G. Schwartz, Monica Divitini, and Terje Brasethvik, editors, *1st International Workshop on Innovative Internet Information Systems (IIIS'98)*, pages 177–187, Pisa, Italy, 8–9. IDI – NTNU, Trondheim (Norway).

[15] John S. Quarlerman and Smoot Carl-Mitchell. The computing paradigm shift. *Journal of Organizational Computing*, 3(1):31–50, 1993.

[16] Anand S. Rao and Michael P. Georgeff. Bdi agents: From theory to practice. In *IN PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON MULTI-AGENT SYSTEMS (ICMAS-95*, pages 312–319, 1995.

[17] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. CArtAgO: A framework for prototyping artifact-based environments in MAS. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for MultiAgent Systems III*, volume 4389 of *LNAI*, pages 67–86. Springer, may. may2006. Selected Revised and Invited Papers.

[18] Hartmut Schmeck, Christian Müller-Schloer, Emre Çakar, Moez Mnif, and Urban Richter. Adaptivity and self-organization in organic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 5(3):10:1–10:32, sep 2010.

[19] B. Schwab. *AI Game Engine Programming*. Course Technology, 2009.

[20] Michael Woolridge and Michael J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.