Group d633a

An AI Framework for Real-Time Strategy Games

DAT6 REPORT
6th of June 2006

```
void rts_game_ai()
{
    cheat_a_bit_before_game_starts(resources += 10000000, hugebase++);
    send_multiple_weak_and_pointless_attacks();
    act_like_youre_actually_gathering_resources_not_cheating_to_get_them();
    cheat_some_more();
    lose_building_to_enemy(OH_NOES);
    cheat_more(CHEAT_INFINITE_RESOURCES | CHEAT_FAST_BUILD);

    while(still_alive() == true)
    {
        send_multiple_weak_units_to_defend();
    }
    die();
}
```

Department of Computer Science
Aalborg University
Fredrik Bajersvej 7E
DK–9220 Aalborg
DENMARK

Faculty of Engineering and Science

Aalborg University

Department of Computer Science

TITLE:
An AI Framework for Real-Time Strategy Games

PROJECT PERIOD:
Dat6,
February 1st 2006 –
June 6th 2006

PROJECT GROUP:
Group d633a

GROUP MEMBERS:
Kasper Frederiksen
Kasper G. Kristensen
Anders Lauritsen

SUPERVISOR:
Thomas Vestdam

NUMBER OF COPIES: 7

REPORT PAGES: 179

APPENDIX PAGES: 108

TOTAL PAGES: 287

SYNOPSIS:

Recently computer gamers have started to focus more and more on the gameplay aspect of computer games. This has led to an increased interest in the multiplayer aspect of games as they present a challenge that the AI in the single player part cannot. So far the single player AIs have been easy to beat, because they lack dynamic capabilities. This project focuses on defining an AI framework architecture for the specific genre of Real-Time Strategy games, which will help to develop AIs with more human-like capabilities and behaviour in this genre. We present a framework based on a model of how a player plays Real-Time Strategy games. This framework is then tested by connecting it to a Game Development Framework. We show that the framework can be configured by both programmers and non-programmers, and that the framework can be used to provide complete AI solutions within the main stream RTS games. Preliminary results show that the ideas behind the framework design and a new data structure, introduced in the project for handling strategies, are very promising.

Preface

This report is the result of the master thesis developed by group d633a (E4-215) at the Department of Computer Science, Aalborg University. The project was developed under the supervision of Thomas Vestdam.

Prior to this project, analysis and preliminary designs were carried out in our pre-master thesis[FKL05]. This has the consequence that the following chapters of the report are partly based on the pre-master thesis: Introduction (Chapter 1), Motivation (Chapter 2), Human Model (Chapter 3), and Framework Design Techniques (Chapter 5).

Readers not familiar with Real-Time Strategy games and the terminology used in these are advised to read our introduction to this genre in Section 1.1 and consult the list of terms and expressions introduced in Appendix A.

In addition to the terms and expressions, the appendix also contains a description and the results from the test carried out in the pre-master thesis[FKL05], a detailed description of the architecture of each module in the framework, the questionnaire sent to the industry along with the answers that were received, and important tables and models. The appendix is separate to the report.

We would also like to thank Oddlabs, Infinite Interactive, Inhuman Games and Fireglow Games for their response to our market analysis.

The code for the prototype implementation can be found at:
http://www.cs.aau.dk/~duck/rtsaif/

Aalborg University
June 6th 2006

Kasper Frederiksen
(macross@cs.aau.dk)

Kasper G. Kristensen
(gib@cs.aau.dk)

Anders Lauritsen
(duck@cs.aau.dk)

Contents

Chapter 1

Introduction

Artificial Intelligence (AI) has for a long time been a discipline in computer science, and has been found very useful in modern computer games. Even though a lot of AI methods are used in the development of AIs in computer games, the AI is still far behind a lot of the other development in the computer game industry, like the creation of more and more realistic graphics[Bur04]. When playing computer games, the player wants to be challenged, and she will not become challenged, if the AI she is playing against is too easy to beat [SZ04]. There exist a lot of different genres of computer games, and concerning the development of AI, *Real-Time Strategy* (RTS) games are one of the more challenging [BF04b]. This is because in RTS games there are hundreds or even thousands of units that have to be controlled in a battle against an opponent. At the same time, there is a well-defined and controlled environment that gives rich possibilities to perform tests for new AI methods. RTS AIs also have their application in the real world. High-performance simulators are needed for training military personnel [BF04a]. One example is the SOAR project [Soa] that was developed for simulators with the function of training pilots. The AI was responsible of providing intelligent behaviour for enemy pilots. [LL01]

In the computer game industry, the production of games is under severe time pressure, and there are demands for continued technological progress. This time pressure in the production of games have meant that the computer game industry has taken the concept of frameworks into use. Some companies exist solely for creating *Game Development Frameworks* (GDF). These frameworks are often called *game engines* in the computer game industry, which refers to the inversion of control that frameworks provide.

The game development industry have started to ask themselves, what would happen if the AI could play like a human? [LL01]. Many games have even started to promote themselves based on the level of their AI: Black & White (2001) [Ban], Half-life (1998) [Hal] and Empire Earth (2001) [EE1], and the industry has started hiring AI researchers to help develop their games. If this development continues, the research done in the computer game industry will overtake that of the academic world [LL01].

This development has, however, not gone unnoticed in the academic society. Many researchers have also noticed that computer games present a prime environment, in which to do human-level AI research:

> *Not only is the game development at the forefront of PC-based visualisation, it is also a leading developer of applied artificial intelligence, overall interface design, persistent worlds, network interaction, and other building blocks needed for next-generation models and simulations.* -Ben Sawyer[1] [Saw02]
>
> *Games provide high variability and scalability for problem definitions, are processed in a restricted domain and the results are generally easy to evaluate.* -Alexander Nareyek[2] [Nar02]
>
> *In contrast to modelling behaviour in the real world, there are (at least theoretically) two great advantages enjoyed by a simulation/game approach: i) full control of the game universe including full observability of the state. ii) reproducibility of experimental settings and results.* -Thore Graepel, Ralf Herbrich and Julian Gold[3] [GHG04]

These are just some of the arguments that researchers have presented in favour of using computer games for AI research. They are, however, still met with skepticism from a large part of the academic community, because computer games (and thereby work related to them) are still looked upon as not being serious work. However, because of all the arguments just presented, research in relation to computer games is still on the advance.

We propose that since the tendency for computer gamers is to seek better gameplay, and because the focus on graphics is on the decline, the time is ripe to integrate more advanced AI methods into computer games. It is now possible to use a much higher percentage of the CPU time for this purpose, as the graphics card is taking over more and more work, and because the CPU in general is becoming fast enough to handle both areas without restricting any functionality.

We have examined the work done in the areas of scientific research within RTS games and AIs and concluded that there is no directly related work.

[1] Ben Sawyer is the co-founder of Digitalmill [digb] and author of several books and articles about interactive game development.

[2] Alexander Nareyek is CEO, CTO and co-founder of Digital Drama Studios [diga], responsible for Artificial Intelligence matters within the International Game Developers Association and chairperson of the IGDA's Artificial Intelligence Interface Standards Committee.

[3] Graepel, Herbrich and Gold are all researchers at Microsoft Research [msr].

This report is the continued work of our pre-master thesis [FKL05], and covers the complete design of an AI framework for RTS games, the implementation and evaluation of a prototype of this AI framework. Our pre-master thesis presented a human model which describes the tasks a human player is faced with when playing an RTS game, and the relationship between these tasks. It was discovered that RTS games could be divided into several genres, and that each genre were focusing on different areas within their AI. This reflected what was the most important part of the gameplay. The focuses that were found could be translated into different parts of the human model. This indicated that the human model could be used as a general foundation for an AI tool, which is able to handle the things that is required to make a human-like AI to an RTS game. Different useful AI methods that could be used in RTS games were also found and discussed. Furthermore, these methods were discussed in relation to the human model as well as their usage in each of the different tasks in the human model.

This raised the interesting question of whether it is possible to make this human model into a general AI tool, in which a developer is able to create AIs to all the identified genres of RTS games that were found in the pre-master thesis.

First we will introduce the reader to the problem through the Motivation in Chapter 2. Next the human model on which the framework is built is presented in Chapter 3. The design goals for the framework is introduced in Chapter 4. Chapter 5 will present the design techniques used in the framework and Chapter 6 presents the framework design itself. The implementation is presented in Chapter 7, followed by an evaluation in Chapter 8 and a discussion of the results in Chapter 9. Finally in Chapter 10 we will conclude on the project.

1.1 Real-Time Strategy Games

The genre called Real-Time Strategy (RTS) games refers to a very specific genre, and not all strategy games that takes place in a real-time environment fall into this category. The term *real-time* refers to the fact that RTS games progresses continuous rather than turn-by-turn, while *strategy* refers to the fact that a player is in control of high level war planning. RTS games are characterised by being games, where the player looks down on the map from above, and gives orders to units and buildings on the map. Moreover, the player is responsible for controlling resource gathering, base building, combat and technology advancements. These are central gameplay elements of any RTS game. The RTS genre differs from the *God Game* genre [god] by not allowing the player to interact directly with the environment. The RTS genre was defined by the first game of this type, Dune II (1992) [dun] seen in Figure 1.1. The game basically consisted of the player having harvesters to

Figure 1.1: Screenshot from Dune II

harvest the resource in the game, and then using these resources to build new buildings or units. The units should then be used to attack the enemy and thereby obtain the goal of any RTS game - to destroy the enemy. This gameplay formula has since been followed by numerous RTS games. In general, any RTS game consists of the following three states [rts]:

- The player must build up her base and her forces.
- The player must attempt to locate and secure resources, to provide a solid economy.
- The player must attack the enemy, and thereby deprive her of resources or destroy her base infrastructure.

RTS games have since Dune II used all kinds of units, buildings and weapons in their games, but the basic gameplay has stayed the same. Recent RTS games have added extra features that makes the game stand out from the rest and increase gameplay. Lord of the Rings: Battle for Middle-Earth (2004) [lot] has for instance simplified resource gathering by not using workers, but instead using buildings that automatically harvest a certain number of resources. This does not remove resource management from the game, as the player's resource gathering still depends on the number of resource gathering buildings she has, but it does simplify it, compared to other RTS games. Another example is Warcraft III (2002) [warb], which added heroes to the game and added NPC characters spread around the map. The heroes could gain experience points by killing the NPCs and thereby increase their strength. Other popular games within the genre of RTS games includes Command & Conquer (1996) [com], Starcraft (1998) [sta] and Age of Empires (1997) [age]. For further information about the RTS genre, we recommend the reader to read the wikipedia definition [rts] or Appendix A for a list of terms and expressions used in the genre.

Part I

Problem Area and Human Model

Chapter 2

Motivation

The purpose of this chapter is to introduce the reader to the problem. In order for the reader to get a good insight into the problem, it will be introduced gradually, and the reader will be able to follow the progress from two points of view: The player's perspective (the user of computer games) and the producer's perspective (the software house that developed the game). Following this, this chapter will consist of four sections:

Problem: This section will give the reader the first insight into why there is a problem in the first place.

Problem Area: This section will get closer to specific problems and briefly explain what cause them.

Current Solutions: This section explains what has been, and what is currently being done by both the players and the producers to handle the problem.

New Solution: This section will outline the solution, on which this project is based.

After reading this chapter the reader should have a thorough understanding of the problem, the problem area and the idea that forms the base of this project.

This chapter is based on a series of tests of AIs' capabilities in a number of commercial RTS games and a number of articles. The tests were made through our pre-master thesis [FKL05]. These can be found in Appendix D and E.1. The articles are on the subject of computer games and AI development: Buro *et al.* are doing work on using RTS games as a test-bed for real-time research [BF04b], Lent et al. present a number of arguments for why computer games are ideal for AI research [LL01], Nareyek also works with how games can be used for AI research [Nar02], and finally Sawyer presents work on alternative applications of computer games and techniques from computer games [Saw02] .

2.1 Problem

As mentioned earlier this section will introduce the reader to why there is a problem in the first place. The context will be computer games in general and the reader will be presented with facts partly taken from history and partly from popular games. In this and all the following sections, the player's perspective will be presented first and then the producer's perspective.

2.1.1 Player Perspective

When a player decides to play a game, it is mostly because she wants to be entertained. The entertainment itself is the result of several factors. Among the most important of these are gameplay, community and story telling.

Gameplay

Gameplay is the oldest of the mentioned factors. In fact the first computer games had little more than gameplay. Pong (1972) [pon] just consisted of two movable bars placed at each their side of the screen and a dot that moved between the sides. As everything else has developed, so too has computer games. Most games have for instance become more complex and graphics have become almost real, the gameplay however still has to fulfil a few simple rules [SZ04]:

1. The player has to make decisions.
2. The decisions have to have consequences.
3. The game itself has to present challenges to the player.
4. There must be a real danger of losing the game.
5. If the player plays a perfect game she must win.

This means that in order to have good gameplay, games have to at least posses these characteristics. Early games like Pong were essentially multiplayer games as one player played against another player. It is much easier to ensure that the gameplay criteria are met, when the greatest part of the behaviour in the environment is due to player actions. This has the direct consequence that it is much harder to ensure good gameplay in a single player game, as more behaviour is controlled by the game itself - the AI. Early single player games like Pacman (1981) [pac] were relatively easy to cope with, as the game had a few simple rules, but as games become more complex so do the behaviours needed for an AI that will ensure a good gameplay. [SZ04]

Multiplayer

Making an AI that ensures good gameplay is *extremely* hard. After the technology allowed people to link computers together in a network the players have been playing games that supported this. Games that were not able to present a good gameplay in single player could suddenly benefit from the multiplayer side of the game. Some games even went as far as to neglecting to implement a single player part and solely focusing on the multiplayer part. First person shooters like Quake (1996) [quaa] and Half-Life: Counter-Strike (2000) [cou] are designed towards a gameplay building on player versus player. The single player part in the Quake series is thus a simulation of player versus player where the adversaries are "bots"[1] controlled by an AI. Most RTS games also have an extensive multiplayer part. They have, however, not abandoned the single player part. This is mostly based on a story line where the AI behaviour can be scripted to a degree that ensures a relatively reasonable gameplay. The extreme is without a doubt the role playing game genre, where the advance of the Internet has meant an almost redefinition of the genre. Massive Multiplayer Online Role Playing Games (MMORPG) are essentially communities of players within the game itself. The entire gameplay, and to a certain degree the rules themselves, are defined by the players.

In short, if the players do not find a sufficient gameplay in the single player part of a game, they will try to find it through playing against other players in multiplayer games. The player versus player interaction again means that the bulk of the behaviour in the game is not controlled by an AI.

Single Player

Looking closer at the RTS genre, the single player parts can mostly be classified as being one of two types: They are either based on a Sci-Fi/Fantasy story or a historical event/civilisation. The story is told with the player taking part in the story itself by carrying out missions with objectives that support the story line. Supported by cut scenes, the player will experience a limited interactive story through the game. Blizzard Entertainment [bli], for instance, is renowned for their extremely well carried out stories that have even resulted in a number of books based on them and a movie is also on the way. This means that the story itself should not be underestimated.

2.1.2 Producer Perspective

The main concern of the producer is to make a satisfactory product - a product that will be a success. To be a success the product must sell and in order to sell it must possess a series of qualities that the player values. This

[1] AI controlled adversaries

can be everything from a good sound track to being based on a known story or figure, but mostly it is factors like graphics and gameplay that are most important.

Satisfactory AI

The first thing that one has to realise is that building an AI to anything but an extremely simple game, is a complex undertaking. In light of the development mentioned in Section 2.1.1, producers have to ask themselves the question: How high do we prioritise single player? In fact what is a satisfactory AI? The producer will have to consider how bad the AI can be (in reality the worse the AI, the faster and cheaper the development) and still entertain the player. One of the tools that have been utilised time upon time in RTS games, is that instead of having a complex AI, a simple AI is complemented by "unfair" advantages such as full map visibility, unlimited resources, or superior forces. While this indeed can improve the overall capabilities of the AI as an adversary, the cheap tricks are easily detected and may result in the player losing the sense of contest with the AI, or even exploit it. One such exploit can be seen in Command & Conquer, where the player, when playing against several AIs, could destroy all buildings belonging to an enemy except the resource tanks. She then had access to unlimited resources, as she could steal it from the AI that had an unlimited supply. Everything considered, making even a simple AI work well is very difficult. The smallest mistake at the wrong time or place can make the AI seem very stupid.

Lately there has been a tendency in the game development industry to try to shift the workload from expensive programmers to cheaper game designers. Blizzard has, for instance, game designers model maps to Warcraft III using the world editor made for this purpose.

A satisfactory AI is relative to the genre and the situation to which the AI is meant for.

Hardware Development

Moore's law states that the number of transistors per square inch on integrated circuits will roughly double every year (in reality every 18 months) [moo]. This means that the development in hardware is progressing in a breath taking pace. This is yet another thing that the producer will have to keep in mind and yet another reason why a product must be developed and released as fast as possible. There are countless examples of productions that have missed their chance. Among these is Tiberian Sun (1999) [tib], the sequel to Command & Conquer. Not long after the release of Red Alert (1996) [red], screenshots surfaced from Tiberian Sun that was to be the hottest thing in graphic development in RTS games at that time. The public, however, had to wait three years for the release of Tiberian Sun, at which time the development had long overtaken the once advanced technology.

Having to take hardware and technological development as a whole in to account makes the development of the product into a race against time.

Graphics Development

There is a different aspect to the technological development that is not covered in the last section. With the advance of computer graphics came a time, where a game as such "just" had to look good in order to sell. That meant that the gameplay did as such not really matter as long as everything looked pretty.

Prioritising towards this comes at two fronts: One being the amount of time spent on developing the various parts of the product, the other being the amount of execution time available to the different parts of the product. With the graphics as focus, this means that there is a very limited amount of time available to both the development of a satisfactory AI, and the execution of the many complex calculations, one such requires. Lately the development has started to turn towards gameplay once more and the graphic side has been signed a lower priority.

The producer must prioritise. She must consider what will sell the product: Screenshots from the game or a promise of good gameplay.

Primitive Techniques

With both a limited amount of time to develop the AI, and a very limited amount of available execution time, the developers are forced to use an array of primitive AI techniques. Among the most common is the use of scripting when defining behaviour. The developers will simply script a simple process that will get the AI through the starting phase of the game, and then loop a behaviour once it has reached a certain point. Some games try to incorporate some traits of complex behaviour when scripting their AI. This is both good and bad. In the game Armies of Exigo (2004) [aox] the AI will try to retreat if outnumbered, which in itself is indeed the right thing to do. The downside is that the AI will try to get back to its main base. If the superior enemy army happens to stand between the retreating army and the main base it will walk straight through this. Another example is the First Person Shooter game F.E.A.R. (2005) [fea], in which the enemy troops will try to move out towards the player's position aggressively and even covering each other in the process. However, the player is able to plant mines between the enemy troops and herself and the AI will walk blindly into these, when they are advancing.

Scripting an AI is done with patches and patches on patches.

AIs Built Late in the Process

Only tools for the AI can be made during the development of the product. The AI itself cannot be implemented before very late in the process. This is due to the fact that behaviour is very vulnerable to new design decisions and an activity such as balancing cannot take place before the product practically is done. Balancing itself is a long and thus costly process. Furthermore only very few people can be active at a time when balancing. One can for instance not let one designer tune the strategies and let another tune the build orders as one is very dependent on the other. This is yet another reason why the whole development of the AI is under an extreme time pressure.

2.2 Problem Area

In the previous section the general problem was introduced by presenting how the capabilities of the opponent affect the gameplay and why current AIs are not better than it is the case. In this section specific problems concerning AI in RTS games will be introduced and their cause will briefly be explained.

2.2.1 Player Perspective

This section will introduce some of the most common flaws that the player encounters when playing against an AI. First the AI's general lacks will be discussed, then some common bad decisions will be listed and lastly some general flaws will be introduced.

What Does the AI Lack?

In order to see what the AI lacks in general, it is necessary to look at what we call static and dynamic behaviour.

Static Behaviour: This is when the environment or rather changes in the environment does not affect the chosen strategy. This does not mean that the AI only has one strategy, but rather that once it has chosen a strategy it will follow it to the letter and nothing can change this.

Dynamic Behaviour: This is when the AI observes the environment and takes action accordingly. It may also be able to record data and store it over time.

Most AIs in RTS games are static. Among other things this means that once the player has found one way of defeating the AI, she can simply do this again and again. In Starcraft, for instance, it was possible to fast tech towards stealth units and consequently surprise the AI. In general, this just means that the AIs, currently used, are static, but what the player really needs to ensure a good game play are dynamic AIs, as they among other things will ensure the ability to adapt.

Common Bad Decisions

In this section a few common bad decisions AIs make will be discussed:

Single Units Attack: It is common that the AI lets single units attack the entire enemy forces or perhaps launch an attack on the enemy's main base. There are mainly two reasons for this. The player can in some cases lure the AI's units away from the main army by attacking a unit standing at the edge of the army. The attacked unit will then follow the unit that attacked it without the rest of the AI's army react. The other reason can be that the AI's routine somehow has been disturbed. This can for instance be in form of an attack on its base or because it is running low on resources.

Single File Movement: When moving over large distances, all units in the AI's forces will move according to the shortest path from starting point to destination, even if this means that they have to move in a single file. If the destination, for instance, is a heavily defended base, the army will arrive one unit at a time, and there is a risk of the base defence being able to kill the units as they come.

Entry From the Same Point: If the pathfinder is the only factor used to decide the direction from which to attack the enemy, the AI will always attack an enemy base from the same point. This means that if the player's base has two entrances, she only has to defend the one that is attacked while letting the other stand undefended.

All these are examples of behaviour that are unfortunate, but just as unfortunately all too common in RTS games.

General Flaws

This section will present some areas common AIs only can handle partly or not at all.

Limited Amount of Strategies: The AI in Warcraft II (1995) [wara] has three different strategies available. It can either attack by land, air or water. It chooses strategy at the start of the game and it will follow the strategy throughout the game. In its successor Warcraft III each AI (one AI for each race) only has one strategy. Generally the AIs do not have a very high amount of strategies available as each strategy more or less requires a separate script.

Countering: As a direct consequence of the low amount of strategies and the fact that most AIs have a static behaviour, countering them is very easy. Once the strategy the AI has chosen has been identified, a counter strategy can easily be picked. The AI does on the other hand not counter as it follows a predetermined strategy.

Cooperation: There are two scenarios in which the AI has to cooperate with someone: It can either be cooperating with another AI or with a player. In most cases, where the AI has to cooperate with another AI, they actually still play as they would have in a 1on1 game except that they do not attack each other. This is also the case in most games, where the player and an AI are allied. There are, however, exceptions. In Warcraft III the AI will mark the place on the map where it intends to attack and it will also assist if the partner is under attack. In Empire Earth 2 (2005) [EE2] the interaction is done through diplomacy. The two partners sign a contract, where common attack orders are described. In both cases the cooperation is not even close to that between two players. The lack of communication makes it difficult to call it cooperation at all.

2.2.2 Producer Perspective

This section will emphasise and discuss some of the design decisions the producer will have to deal with, developing an AI.

Simple AI

In some cases a simple AI will suffice. A simple AI does not have to be an easy-to-beat AI. Some games try to have an aggressive AI so that the AI does not have to counter, as it sets the agenda. An example of one such AI is the one in Starcraft. The AI has an extremely optimised build order and it attacks once it has reached a predefined amount of units. It does this a few times in a row as it reaches higher and higher tiers of units, but when the player starts to make a lot of expansions, the AI can no longer keep pace and starts to fail, as it no longer has the lead.

Static is Safe

It is very difficult to make a formal representation of how to evaluate any given situation. This, among other things, means that learning is extremely difficult. Even if an AI had a marginally usable evaluation function, it would be too dangerous to ship the AI while able to learn. Should the evaluation fail even once, it could result in the total failure of the AI. If the product is static, the producer knows what she can expect from the product.

Cheating

As already mentioned, the one widely used technique to simplify the AI while still achieving a relatively reasonable result is to cheat. Cheating is mostly done through: Full map visibility, unlimited resources, or free units. The downside of cheating is that it is easy for the player to discover that the AI is cheating, and once this is discovered, the player will not really consider losing, for losing because the game was not played on equal terms. This means that two of the criteria for providing gameplay have been severely weakened (gameplay rule 3 and 4).

Fast AI

As mentioned earlier, the AI in for example Starcraft is very optimised, that is, workers do not waste time between building etc. It is especially noticeable in the start up phase of the game. Players cannot keep pace with this. If the AI could keep this up throughout the game, it would have a sizeable advantage. If restricting the AI in this field just because players cannot keep up, the design will just have opened a huge design discussion about whether the AI should be optimal or simulate human behaviour.

Multiple Units Controlled

A player will be able to build new buildings in her base, scout unexplored territory, and move her army into battle at the same time. This level of simultaneous actions requires a bit of training and more simultaneous actions will in turn require even harder training. The AI, however, does not have this restriction. It could if needed give new orders to every single unit it owns. When deciding how many units the AI may give new orders at a time, the designer must keep in mind that there is a fine line between reasonable design decisions and what the player will consider cheating. Currently AIs do not handle this, but just give the orders that are needed.

Builder AI

Anyone who has tried to play against an AI in Age of Empires would know that a war against a "builder AI" can take a very long time. When calling the AI for a "builder AI" it is meant that the AI will constantly try to expand if it has the resources. If the player finds the AI's base and destroys it, the battle is still far from over. The AI could already have 20 small settlements spread all over the map. After that, the fight will never really be a fight, but rather a long game of hide and seek. This form of behaviour is as such the optimal way of playing - always trying to survive and hope for a comeback, but on the other hand in most cases, it is just delaying the inevitable, with the result that all players get bored. This is yet another issue where the optimal solution and the wanted solution could be in conflict.

Unit Composition

As already mentioned earlier the AI is not able to counter when it is a static solution. This means that given the chosen unit composition the player should be able to find a perfect counter for what the AI has built. In order to counter this, a widely used technique is to build a little of everything so that every thing in turn can be countered (if only by a small force). This does on the other hand mean that no matter what the player builds she will have a counter to something in the AI's army. In Warcraft III, where support units play an important role, this kind of mixed group is quite successful, especially when the army grows over a certain size. An optimal solution would of course be to let the AI find a counter to what it meets, but that would require a dynamic AI.

2.3 Current Solutions

So far the reader has been presented to a series of problems with the AI that is apparent to the player and a number of cheap solution techniques that have been used in games in an attempt to improve the AI. This section will present a number of solutions that the player has found to deal with the problems and some solutions the producer currently is using to improve the AI's performance.

2.3.1 Player Perspective

As already mentioned in the previous section, the player has found a way to ensure a relatively good gameplay: Multiplayer. This section will explain why multiplayer indeed gives a better gameplay, and it will also present some of the facilities the players use.

Dynamic Behaviour

The reason, why multiplayer gives a better gameplay than the AIs found in the games so far, can be summed up in two words: Dynamic behaviour. A player will learn from mistakes and generally make better decisions than the AIs currently found in games. When a player plays against another player it is much harder to anticipate what the opponent will do, except that it will probably be something that will bring the player into the worst possible situation. Multiplayer is not only about one player playing against another player, but just as much a team playing against another team. In team games the interaction and cooperation aspect adds to the gameplay. Having a static AI as partner usually gives a bad experience, while player-player cooperation can add a completely new aspect to a game.

LAN

Local Area Network (LAN) parties are gatherings of players that meet to play one or several different games. The number of players can vary from a few to several hundred. The small LAN parties are usually social parties among friends that meet to play against each other to have fun. The larger LAN parties often feature tournaments where single players or teams of players will compete against each other. The social aspect of LANs only add a positive effect to the gameplay as the opponents are characterised and no longer faceless entities.

Communities on the Internet

With the advance of the Internet, players have gotten together and founded communities in which they play with and against each other and share experiences. Indeed entire games genres like MMORPGs have been based on this. The communities extend the gameplay from being entirely dependent on the game itself to being heavily influenced by the players, thus making it more dynamic in nature. Some producers have embraced this idea and made servers available to the public. One example is battle.net that is the portal Blizzard Entertainment is using for most of its published games that include multiplayer facilities. The player will simply connect to battle.net [bat] where she will be able to create and join games open for other players. Here winning games will also result in a better placement in a server ranking system, which in itself can present a major challenge. It is, however, not only the producers themselves that make servers available to the players. There are indeed also a myriad of private servers. These are everything from commercial pay per use servers like Kali.net[kal] to servers owned by clans[2].

2.3.2 Producer Perspective

This section will present the solutions the producers have chosen to the problem.

Multiplayer Solutions

Following the development, some producers have chosen not to include the single player part of the game at all. Instead they have focused solely on the multiplayer part. This is mostly seen in Role Playing Games (MMORPGs) and in First Person Shooters (FPS). Doing this they will not be entirely rid

[2] Groups of players united by the common interest in a game

of making AIs to their games, but they can in turn be relatively simple as they are not vital to the intended gameplay.

Story Telling

Other producers mostly of FPS and RTS games have chosen to focus on the single player part as well. The simple AIs are complemented by extensive story telling. The player follows a story line and the AI can be heavily scripted to carry out events as the story unfolds. This way the producers can control the environment to such an extend that the AI can be scripted to an acceptable level of gameplay. Even if the player does mind that the AI is scripted and cheats, she will still play the single player part of the game, if the story is good.

Good AI

Lately a few producers have started to focus on making the AI better. This is an obvious, but difficult solution to the problem. One of the first steps towards an acceptable AI is letting the AI play using the same rule set as the player. In other words, the AI should not be allowed to cheat. Empire Earth II attacks one of the most commonly areas in which the AI cheats by actually making the AI scout instead of having the entire map visible at all times[FKL05]. Other games try to include advanced features like counters (Age of Mythology (2002) [aom]), retreating when outnumbered (Armies of Exigo) and using templates to obtain a well designed base (Warcraft III). Each feature is a step in the right direction, but there is still much to be done, before the result is good enough to compete with the level of gameplay found in multiplayer games.

The Ideal Solution

In order to improve the gameplay found in the single player part of a game drastic measures must be taken. No matter how scripted an AI is, it will still be too static to offer the player a serious challenge if playing on the same terms. In reality it will be impossible to script an AI to such a degree that it will be able to account for every possible situation. Instead the producer must ask herself what the goal really is? The player is improving gameplay by playing against other players, thus getting a dynamic opponent. Can this idea also be applied to single player? That is, could the solution be to simulate an actual player instead of a chain of events? In this way the AI should be built to simulate the actual decision process that a player goes through when playing a game. This way the gameplay should be improved in the same way as in a multiplayer game. The AI must however still be able to adjust the difficulty level so that it still follows the rules of gameplay.

2.4 New Solution

This section will present the basis of the ideal solution outlined in the previous section and consequences that the solution will have on the players and the production.

2.4.1 Player Perspective

This section will present the consequences this solution will have to the player.

AI and Story Line

An AI that plays like a human player will have a significant impact on the single player part of most games. It will mean that the player will constantly be challenged as if it was a multiplayer game and on top of this, she will be carried through a story line as it is normal for single player games. It will also mean that the player will be able to play the same game several times and though the story might be the same, one game will always differ from the last (given that the AI learns from previous games). In a non-story-related context (custom games), the player will be able to enjoy playing against an opponent in an environment defined by herself. This could for instance be useful when testing a new strategy or when playing offline in general.

Team Games

Another application for an advanced AI is team games. If the players are of an uneven number an AI would be able to step in and even the teams. This will also mean that a single player will be able to play team games offline by applying three or more AIs. When it comes to cooperation, the player must be able to communicate with the AI. In current RTS games, communication between player and AI has been very limited. In Warcraft III the AI will respond to the player being attacked. If the player is attacked, the AI will teleport to the player's base and assist in pushing back the enemy forces and if the player is attacking, it will rush to help in the attack. Empire Earth 2 presents one of the most advanced player-AI communication systems. The player can access a strategy window containing an overview map in which she can draw arrows directing friendly forces. When the orders are accepted by the AI, it will try to carry out the plan. However, in order to be able to cooperate at the same level as a player, the cooperation interface has to be more extensive allowing somewhat the same kind of communication as observed between players.

2.4.2 Producer Perspective

In this section a number of options available to the producer will be presented.

Starting On The AI

The first thing the producer will have to decide is which approach she wants to utilise to build the AI. The normal approaches are:

Making the AI From Scratch: The first thing that comes to mind is to build the AI from scratch. It is, however, also the approach that requires the most work. Not only do the developers need to figure out a way to structure the AI, but everywhere they look there will be a series of new problems, to which they have to design solutions. The developers will gain experience throughout the process, which they may put to use in a later production depending on the similarities between the two.

Libraries: The producers may not have to write the entire AI by themselves. As with experience, it may be possible to reuse code from a previous production or perhaps acquire useful libraries from a third party. Due to the fact that they have already been in use, these tools will be well-tested and if they are from a previous production, the actual developers may already be familiar with them. The downside using libraries is that the developers are still forced to design a structure - indeed the entire concept for the solution.

AI Frameworks: If the producer decides to use one of the current AI frameworks, she is faced with the other extreme to making the AI from the bottom. The producer will be working with a very rigid structure. The solution will be based on a single problem solving technique, as for instance SOAR [Soa] that is a planning system.

One of the reasons why current AIs are lacking good performance may very well be that the producer is utilising the "Making the AI From scratch" approach. If the developers indeed have to build the AI from the scratch, they will often end up with a solution that has to be simple in order to meet the deadline. Using libraries usually means that the AI will have a few advanced features while still essentially being the AI that would have been build using the "From scratch" approach. If the producer uses one of the current AI frameworks for the AI, the production team will have a lot more time to do balancing and fixes as the AI development mainly will consists of feeding the framework data so that it fits to the current game. The problem using an AI framework is that it tries to present a general solution to all games (like planning). That is the same thing as assuming that a player will be able to master all games, if she masters one game and then is told the basic rules of the rest.

Different Genres - Different Focuses

In our pre-master thesis[FKL05] we identified four different genres within the RTS genre through a series of tests. Each genre is defined by the playing style used in it. The genres are as follows:

- The Command & Conquer Genre: Command & Conquer, Red Alert, Warzone 2100 (1999) [warc], and Dark Reign 2 (2000) [dar].
- The Age of Empires Genre: Age of Empires and Empire Earth.
- The Starcraft Genre: Starcraft and Armies of Exigo.
- The Warcraft Genre: Warcraft II and Warcraft III.

The genres are named after the games that defined the genres. In most cases the game was the first popular game of that particular playing style. The test that was the basis of these definitions can be found in Appendix E.1. In the test, the capabilities of the AI in a series of games were tested in a variety of important tasks. When looking at these capabilities, it is possible to see which areas are more important than others in the different genres. This means that the producer will have to be very conscious about which capabilities are important in the game she is developing.

2.5 Discussion

The lack of challenges in the single player AI has driven the player to seek other means to find challenges. They have solved this problem by playing against other players. But what would happen if the AI was capable of playing like a human player? An advanced AI will have a serious impact on the gameplay of both single player games as well as multiplayer games. The combination of story telling and the challenges a dynamic AI will provide will strengthen the single player part and a capable AI will be able to assist in team games.

When developing one such AI, the producer will have to choose between the lack of structure in using libraries or the rigid structure of a general solution framework. Current frameworks provide a general solution technique that the producer must fit to the game and the libraries provide no structure at all [Soa]. The different playing styles in different RTS games mean that the player or the AI have to focus on different aspects of the games depending on for instance which genre it belongs to, the environment and the opponent. To our knowledge there are no frameworks that are able to handle this in the RTS genre and only a few libraries that handle various sub-problems. Under all circumstances, identifying a good structure is the first step towards a good solution. As already mentioned, the player found satisfaction in playing against other players, as the other players are able to handle most of the problems outlined throughout this chapter. We will therefore use the player herself as a foundation for building AIs able to meet the challenges. This will be done by setting up an abstract model for how a player plays. Furthermore the problems identified throughout this chapter will also be used to set up design goals for the framework.

Chapter 3

Human Model

Before trying to build a human-like AI, it is important to consider how a human player plays RTS games. What kind of knowledge does she posses before game start, and how is this knowledge used in the game? Which general tasks are the human player thinking about when playing, and how do these tasks influence each other when making decisions? The idea is not to model the actual thinking process of a human playing RTS games as it is very complex and will not provide a distinction of different tasks, but rather to build a more structured human model with a focus on defining tasks and the relationship between them. The different tasks defined and the relationship between them is partly based on previously defined areas in RTS research[Sch04] [CBS05] and partly on our own experience from years of playing and from watching professional gamers play RTS games. The model will focus on mainstream RTS games, which includes the games in the genres presented in Section 2.4.2. In previous research, Brian Schwab[Sch04] defines the areas *town building*, *opponent modelling*, *resource management*, *scouting* and *diplomacy systems* as important areas, while Michael Chung *et al.* [CBS05] defines the planning areas in RTS games as being *micro-management*, *tactical planning* and *strategic planning*. This model will use some of these previously defined areas. We are not claiming that the presented model is the only true model for a human playing RTS games, but it is in our opinion a good representation on which we can base our further work. Furthermore, it is hard to test the correctness of these kinds of models as all humans play differently and take different things into consideration when making decisions. We claim however, that the model presented in this chapter is a reasonable representation of how most people play RTS games. Section 3.1 will begin by presenting the knowledge a player has before a game starts, and then Section 3.2 will proceed to present the knowledge a player has and maintains during a game. In Section 3.3 we will identify key tasks that a player must solve to play at a human level, and then determine which tasks are influenced by other tasks or knowledge bases in Section 3.4. This will result in a human model of how a typical player plays RTS games.

3.1 Prior Knowledge

The following presents the knowledge a player has before starting a game. Each of these areas will influence how the player will play the game.

Map Knowledge: This area represents knowledge about the map terrain, map size, resource locations, strategic and tactical important locations etc.

An example is in Starcraft, where the knowledge of a high ground behind an expansion can easily decide the outcome of a game. A terran player would be able to build bunkers, turrets and place siege tanks on the high ground, making it virtually impossible for the opponent to make an expansion at this spot, because of the advantage of high ground in Starcraft.

Enemy Knowledge: Experiences against players throughout several games will give the player an idea of how the enemy player thinks and what kind of strategies she uses. This prevents the player from losing to the same strategies again and again against the same opponent, as she is capable of trying new things and thereby countering the opponent's strategy. This of course only applies to players of equal skill level in all areas, because knowing the opponent's strategy will often not be enough for novice players to beat professional players.

Knowing that an opponent has a tendency to get air units very fast, the player would most likely try to rush her or at least prepare for this strategy by building some defensive buildings or units able to hit air units.

Gametype Knowledge: Depending on whether the game played is a team game, a 1on1 game or an FFA (Free For All) game, the strategic considerations change.

For instance, in FFA games the player needs to be much more aware of things happening around the map that does not directly involve herself. Building the right counter is also much more difficult as opponents can have vastly different armies.

Known Strategies: Most players have a number of strategies they have either invented for themselves, learned from watching other players or found on the Internet. This area affects both the number and quality of strategies used by the player, but also the capability of predicting the opponent's strategy, and knowing how to counter it.

Knowing just one very effective strategy can in some games make a player win a lot of games. In for instance Starcraft, knowing only a fast air strategy can bring a player a lot of victories by surprising the opponent and requires relative little micromanagement.

Known Build Orders: In all RTS games the start of the game is very important and an effective build order can prove invaluable. The build order defines in which order to build everything such as workers, buildings and combat units, and also specifies what each worker should be doing at any given time. A build order is often used in connection with a certain strategy trying to maximise the player's resources and getting to a certain point in the strategy as fast as possible.

The importance of build orders can be easily seen in Starcraft, where an effective rush or fast-air strategy depends heavily on the build order used. These kinds of strategies requires that they are carried out as fast as possible with the largest force possible, and build orders play a vital part in achieving this.

Game Specific Knowledge: Depending on the game in question, a player will have knowledge about the different units, buildings, resources and research options available, as well as the possible actions for each particular unit and building.

Knowing the details of units, buildings, resources and research options in a particular game is essential to playing the game at a human level. Without this information a player would for instance not be able to determine how to build a certain unit, because the player would need to know the resource requirements for that unit as well as possible unit or building dependencies for that unit.

3.2 In-Game Knowledge

While the former section focused on knowledge that a player has before a game starts, this section will focus on the knowledge that a player has and collects during a game. Each of the following four defined areas will have an impact on the decisions a player makes throughout a game:

In-Game Enemy Knowledge: During a game, a player will always have an idea of what the opponent is doing. This could both be in terms of what strategy she is doing, but also enemy unit movements and activities around the map.

This could for instance be the fact that the opponent has built a certain building or that the majority of her army was spotted close to the player's main base. Furthermore, a player could have specific beliefs about when and where the enemy is going to attack, based on what it has seen from the opponent so far.

Unit and Building Information: The player will at all times know what kind of units and buildings she has, and what they are currently doing. This includes building queues, unit attributes, assigned actions etc.

This information is used during games to for instance withdraw wounded units, plan what different units should be doing after their currently assigned action etc. The player could also use this information to obtain the position of all friendly army units and from this dictate a specific tactic to be used in battle.

Own Strategy: This includes all strategic aspects the player may be considering. It may be that the player has a certain strategy she is working towards, and she may have a very specific plan of what units and buildings should be produced to achieve this.

This area will mostly be in the form of plans for different activities like how to build the base, when to expand, how many workers to build etc. It contains knowledge about both the current strategic status and what the goal for the player is, strategy-wise.

In-Game Map Knowledge: During the game, the map that is played on may change depending on game specific details or maybe the players will modify it somehow. This knowledge base will store all things on the map that changes during a game.

The use of the knowledge base will vary from game to game, but almost all RTS games features finite resource amounts and the player must keep track of where and how many resources are left around the map. In some games, for instance Warcraft III, the player must also keep track of which NPCs that have been killed around the map and on what the shops and mercenary camps have to offer.

3.3 RTS Tasks

The following describes ten RTS tasks that a player will encounter when playing any modern RTS game.

Strategic Planning: This is the overall planner. It is the task that is equivalent to that of a general. More specifically, the task decides what the overall plan is for longer periods of time and is hence responsible for long-term planning.

This can be things like deciding to attack at a certain point or creating an army consisting of a particular list of units.

Tactical Planning: The task can be considered the job of a sergeant who takes orders from the general (*Strategic Planning*). This task is mainly responsible for the tactics in battles and for keeping the tactical overview at all times.

During battles this task includes decisions like reinforcing a flank, taking advantage of the higher ground, retreating etc. Furthermore, it for instance takes care of keeping armies in formations during movement to avoid single-line formations.

Micromanagement: This task is to issue orders to each individual unit based on *Tactical Planning* and *Strategic Planning*, and the orders are carried out instantly.

This is everything from focus fire on enemy targets to pulling back hurt units and using support units.

Reasoning: This task is focused on reasoning about observations of the enemy. It analyses all activity from the opponent and thereby determine exactly what the opponent is doing or trying to do. The task's basic job is to provide solid information for the *Strategic Planning* task to base its decisions on.

For instance if the player has seen an enemy worker running past her somewhere outside the enemy base, it could mean that the enemy is trying to build an expansion or perhaps attack the player's base using the offensive capabilities of defensive buildings. Another example is if the player sees a unit production facility for air units in the enemy base, it could be wise to produce a number of anti-air units or send out scouts to find out how many air units the enemy already has and then take action accordingly.

Opponent Modelling: A player must at all times keep a model of the opponent. This includes not only what the enemy currently has in terms of buildings, units and research upgrades, but also more abstract beliefs about her chosen strategy and her current strategic situation.

When making decisions about what strategy or tactic to use in a game, it is essential to have a good idea of how the opponent's army is composed and what her situation is like. In Warcraft III for instance, the opponent model could consist of things like army composition, technology level, income rate, upkeep estimate, resource estimates etc. Essentially, everything that has a strategic influence should be a part of the opponent model built during a game.

Resource Management: *Resource Management* also includes resource gathering. This task includes determining, which resources are required, and optimising the gathering of these resources.

If the *Strategic Planning* task has decided that a number of a certain unit is to be produced, the *Resource Management* task has to make sure that this can be done as fast as possible by anticipating resource requirements. If for instance the specific unit demands a lot of lumber that is not in store at the moment, the resource manager may have to reassign some workers to gathering lumber ahead of time. The resource manager also has to figure out the optimal way to gather resources using the least number of workers.

Base Building: As the name hints this task is responsible for building the base. This has two aspects: Building the right buildings and placing them correctly. To place buildings correctly the player must furthermore have some kind of plan about which buildings are soon to be built.

Building the right buildings is closely related to the chosen strategy. It could also be in the case that the player has more money than normal and it would be an advantage to build another unit production facility. Placement of buildings can be more complex. This is often a matter of placing defensive buildings in good positions covering the base or placing harvesting buildings near resources allowing for faster resource gathering.

Scouting: Most modern RTS games have either one or two layers of Fog of War. In order to support other task's ability to make good decisions, the player has to send out scouts. This also includes the task of selecting the unit to scout with and deciding how often scouting is necessary.

Scouts can be used to figure out what is happening in the enemy base, discover expansions or keep track of the enemy army's movement. The decision about which unit to send could depend on the range of sight and speed of the unit and also which and how many resources are lost by choosing that particular unit to scout.

Learning: From game to game a player will constantly learn new things. It includes new strategies, opponent models and effectiveness of certain strategies against other kinds of strategies. The player would have to evaluate the game played either during or after the game, and from this infer which critical situations in the game determined the winner.

If a player plays against the same opponent a couple of times, she might recognise a pattern in how the opponent thinks or just find a certain way to beat her. It could be that the opponent has a tendency to rush, in which case a strategy of moving fast up the technology tree would be a bad idea. Also, by being able to learn strategies from the opponent, it is possible for the player to use these strategies at a later time when confronted with the same situation that this particular strategy was successful in handling.

Cooperation: In team games the players' task is to work together and find strategies, where each player complements each other in the best possible way. Furthermore, a lot of coordination is required to ensure the right tactical decisions from each player during big battles. Cooperation also includes the task of deciding if and when to share resources, and in FFA games, the task of figuring out when to betray an ally and when the player herself is being betrayed.

A typical cooperation scenario would be for one player to produce melee units and the other to produce ranged units. This specialisation for each player allows for building only certain kinds of buildings, as well as only requiring to upgrade a certain type of weapon. Tactical coordination could be things like letting one player lurk out the enemy from a small passage, while the other players remain hidden until suddenly coming up from the behind of the enemy. Betrayal of an ally could be for a player to indicate that they should both attack at a certain point, and then just not show up leaving the betrayed ally alone against the enemy.

3.4 Human Model

Each of the tasks mentioned above are influenced by several prior- and in-game knowledge bases, as well as other tasks. In the following a model of how all tasks influence each other will be presented. An illustration of the model can be seen in Figure 3.1 or in Appendix K.1. Circles represents tasks and arrows indicate which tasks influence each other. The small boxes attached to each task shows which knowledge bases influence that particular task. The numbers in each of the small boxes refer to the numbers in the large Prior Knowledge and In-Game Knowledge boxes on the right of the figure. Note that for illustration purposes, an observation task has been included to indicate which tasks are influenced by observations during a game. In the following each task's role in the model will be explained along with a discussion of which knowledge bases contributes to solving that particular task:

Strategic Planning: This task relies heavily on the *Reasoning* task to figure out what the opponent is doing, as most games includes ways of countering all possible strategies. When this has been determined, the *Strategic Planning* task relies on several different knowledge bases to select the best possible strategy in the given situation. The most important prior knowledge base is naturally *Known Strategies*. This knowledge base basically contains all known strategies and all counters

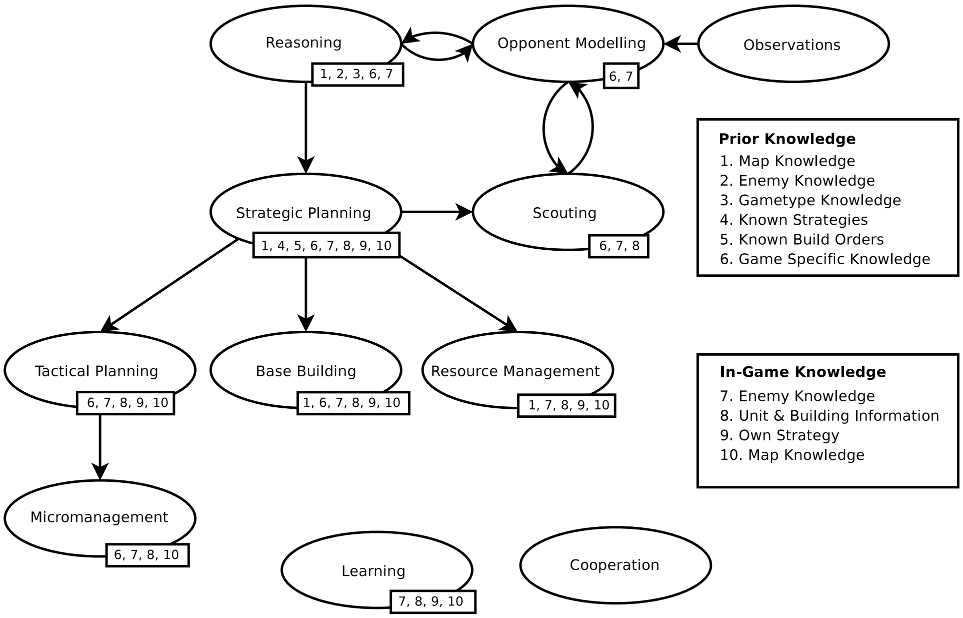


Figure 3.1: A human model for playing RTS games

to known strategies, and it is essential for this task to be solved the right way. Besides this, the task uses knowledge from three other prior knowledge bases:

Map Knowledge: *Map Knowledge* provides map specific details that influence the choice of strategy.

Build Order Knowledge: *Build Order Knowledge* provides more specific details of how to execute the beginning phase of a certain strategy.

Game Specific Knowledge: Finally, *Game Specific Knowledge* provides the details of the particular game in question, as strategies varies from game to game.

The *Strategic Planning* task likewise takes into consideration all four types of in-game knowledge, as these knowledge bases represents what is currently going on in the game, and this obviously has a great effect of what strategy to choose.

Tactical Planning: *Tactical Planning* is primarily influenced by the *Strategic Planning* task. This is because the primary objective of the player's army is given by the *Strategic Planning* task, while the part of actually carrying out the objective is left to the *Tactical Planning* task. To solve this task, the player must rely heavily on *Game Specific Knowledge*, which provides details about units in the game and the actions they are capable of performing. Furthermore, the player uses the four different in-game knowledge bases to obtain knowledge about the current situation in the game.

Micromanagement: This task primarily relies on *Tactical Planning* to indicate how it should carry out its task. Furthermore, it uses *Game Specific Knowledge* to determine unit hitpoints, armour types, attack types etc. which are essential knowledge for the task to be carried out successfully. The task does not need to know the details of the player's strategy, but it does need to use information from the other three in-game knowledge bases: *In-game Enemy Knowledge*, *Unit and Building Information*, and *Dynamic Map Knowledge.*

Reasoning: The *Reasoning task* primarily uses the opponent model built by the *Opponent Modelling* task. This is where everything opponent-related is obtained from. Reasoning about this information, however, requires that the player must use several different prior knowledge bases:

Gametype Knowledge *Gametype Knowledge* influences reasoning because an opponent's actions should be interpreted differently depending on the gametype.

Enemy Knowledge: This knowledge base is important because the player will be able to recognise patterns in an opponent's strategy, which will often indicate moving towards another strategy.

Map Knowledge: *Map Knowledge* influences *Reasoning* because some strategies are used very often on some maps and very seldom on others.

Game Specific Knowledge: Finally, *Game Specific Knowledge* provides game details such as technology trees to help reason about the purpose of different buildings and units.

Finally, *In-game Enemy Knowledge* is used to reason about the opponent's movement around the map.

Opponent Modelling: This task must consider all observations from the game as well as the influence from two tasks: *Scouting* and *Reasoning.* Results of scouting missions must be used when building a model of the opponent, and the result of reasoning about the opponent will result in new beliefs about the opponent, which should also be reflected in the opponent model. An opponent model should depend on the game in question, and this is obtained from *Game Specific knowledge.* Besides this, the only knowledge base used is *In-Game Enemy Knowledge*, from which the player can retrieve information about the enemy used to build the opponent model.

Resource Management: *Resource Management* is primarily influenced by the *Strategic Planning* task. A strategy may include specific details that this task must try to accomplish, like for instance building an expansion or gathering a lot of a certain resource. Out of the prior knowledge bases, the player needs *Map Knowledge* to determine resource locations and the amount of resources available in a certain location. Furthermore, the player will need all four in-game knowledge bases to solve this task successfully:

- **In-Game Map Knowledge:** The player needs to know how resource locations and amounts change during a game.
- **Own Strategy:** The details of the player's overall strategy will be required to better manage resource gathering
- **In-Game Enemy Knowledge:** The location of enemy units plays a role when deciding where it is possible to harvest resources.
- **Unit and Building Information:** When assigning actions to workers it is essential to know where and which workers are currently carrying out which actions.

Base Building: This task is likewise primarily influenced by *Strategic Planning*. The chosen strategy will have a large effect on how the base should be constructed. Some strategies may require a very compact base able to fend of early attacks, while others may require a lot of anticipation in terms of having room to build the required buildings in the right places. The task only requires two prior knowledge bases: *Map Knowledge*, which is used for building placement, and *Game Specific Knowledge*, which are needed to decide which buildings to build. Furthermore, a player can use all four in-game knowledge bases:

- **Dynamic Map Knowledge:** As some games include dynamic placement of resources, this knowledge base is used to keep track of this, so that a player can take this into consideration when constructing buildings.
- **Own Strategy:** Because of the strategy really dictating what buildings to build, the player must know of this to anticipate how to construct the base in the best possible way.
- **In-Game Enemy Knowledge:** When constructing new buildings, the player must be aware if any enemy units are in the area, because buildings under construction are often very vulnerable.
- **Unit and Building Information:** This knowledge base is used to determine which units are to build different buildings and to determine whether it has the resources to support producing from for instance more than one barracks.

Scouting: The *Scouting* task is influenced by two other tasks. The *Opponent Modelling* task will result in knowledge of which attributes or variables of the enemy that are unknown, and should be further investigated. The *Strategic Planning* task will on the other hand give the player a good idea of which unknown variables may reveal the opponent's final choice of strategy. The player will need *Game Specific Knowledge* to figure out exactly where to scout for different things. Finally, the player makes use of two in-game knowledge bases: *In-game Enemy Knowledge*, when figuring out where to scout, and *Unit and Building Information*, when figuring out which units to send on a scouting mission.

Learning: This task is active when a player reflects on a game being played or a game recently played. She will think about the opponent's strategy, what kind of strategy she needs to counter it and how well this strategy worked out. Moreover, she will think about the opponent's actions and keep in mind what the opponent tried to do in this game. Specific observations about for instance the map or some tactical move is also remembered, so that she can use this in a later game. For illustration purposes this task is not connected to the other tasks in the figure, because it would in reality influence and be able to improve all kind of decision making during a game and hence all other tasks. The *Learning* task could result in learning new information for all of the seven prior knowledge bases except *Game Specific Knowledge*. It will use knowledge from all of the four in-game knowledge bases as well as the special *Observation* task shown in the figure.

Cooperation: For a player to carry out the *Cooperation* task, a lot of communication with other players must be done. Players may want to consult their allies before making any kind of decision, as all tasks in the model could somehow influence the allied players. This means that not only is the *Cooperation* task influenced by the *Cooperation* task of other players, it will also be influenced by every task in the model. Moreover, it will itself influence all tasks in the model. In the following, each task will be described in relation to how it can influence, and thereby also be influenced, by the *Cooperation* task:

Strategic Planning: Allied players should choose strategies that complement each other well.

Tactical Planning: In battles, allied players should try to help each other as much as possible by for instance having one player protecting the other player's weaker units.

Micromanagement: If a player knows that her allies will heal all units in a certain area in a few seconds, she may want to modify her policy for withdrawing wounded units in that area.

Reasoning: Two players may come to different conclusions given the same data, as the knowledge bases they rely on may be different, and hence they must communicate to come to an agreement of what the opponent is doing.

Opponent Modelling: In team games it is essential that players share the knowledge they observe, so that the players are able to build more accurate opponent models.

Resource Management: Players will sometimes want to share resources, and sometimes it is beneficial for both players if one player harvest the required resources and the other uses it.

Base Building: Sometimes players will find it beneficial to build buildings in each others bases, and this of course most be coordinated.

Scouting: It would make no sense for allied players to scout for the same things, as they should rely on each other for information about the enemy.

Learning: Often players will learn from each other when playing team games.

3.5 Summary

This chapter has presented an idea of how humans play RTS games. We have defined a number of prior knowledge bases, which a player is aware of before playing, and a number of in-game knowledge bases, which a player is aware of during a game. Then we defined ten important tasks that a player must go through to play an RTS game at a high competitive level. This resulted in a model of how humans play RTS games, where tasks and their influence on each other is defined, as well as each knowledge base's influence on each task.

The human model presented in this section is the foundation on which all further work is based. If all these tasks and interdependencies are present in an AI, we hypothesise that it will be very hard to distinguish it from an actual human player. The definition of each task and its responsibility will furthermore make it easier to divide the AI into logical modules.

Part II

Framework Design

Chapter 4

Introduction

This part will present the design of an AI framework for RTS games. First the design goals followed throughout the design will be presented in Section 4.1. Then a number of techniques and methods used in the design will be discussed in Chapter 5, along with their pros and cons in developing this kind of framework. This chapter will also present RTS game specific concepts as well as examples of their use in this context. The final chapter of this part (Chapter 6) will focus on exactly how the framework is built. We will start by discussing how to convert the human model, presented in Chapter 3, into a suitable framework architecture (Section 6.1). Afterwards, the chosen data representation for the framework will be introduced in Section 6.2.2, and a discussion of how to configure and extend the framework will be presented in Section 6.3. Not all framework details of the design will be presented in these chapters as this will be too extensive. Other design details can be found in the appendix and will be referenced in the appropriate sections. In the final section of this part (Section 6.4), we will discuss how framework execution is controlled and how it inter-operates with the GDF.

4.1 Design Goals

This section will describe the design goals of the project. In Chapter 2 several problems were identified as being responsible for the relatively poor standard of AIs in RTS games. This section will translate these problems into design goals that should receive special consideration when deciding on a design of a framework aiding developers building AIs for RTS games. In the following, four different design goals will be presented, along with an explanation of what each of them means for the design process.

Improved AI: The framework should help produce better AIs than the industry standard today. This essentially means that the framework must provide advanced techniques and methods for creating strong AI opponents. Furthermore, they must be adaptable to games of different genres within RTS and be able to work when developing many different kinds of AIs. This requirement also means that the framework should attempt to provide methods for solving as many of the tasks defined in the human model as possible.

Reduced Development Cost: For an AI framework to be usable in the industry, it must be able to reduce the development cost of creating AIs. A way to do this is by creating a complete AI solution, so that an RTS game developer does not need to do anything AI related other than connect the AI solution to the GDF and configure it to work in the game being developed. This means that the framework should be created to handle all AI activities, which means that no AI programmers or developers are necessary to use the framework. Furthermore, the framework should include all parts of the AI that do not change from game to game, and provide standard implementations of the areas that are common in most games. In this way the AI developer should be able to focus on game specific details and on areas that are important for the AI in the game being developed.

Shift of Workload: In the late stages of a game project, the programmers are often very busy getting the game to work properly, while the designers have more or less already done their job. As already discussed in Section 2.1.2, shifting the workload of AI development towards designers will not only reduce development time, but also leave designers to do what they do best - creating good gameplay. To allow for this shift of workload, the framework must provide a easy-to-use configuration system, which allows for inexperienced programmers to work on it.

Structured Overview of the AI Development Process: The framework should provide a structured overview of the different kinds of tasks an AI must solve. This allows for focusing on only certain parts of the AI. The human model presented in Chapter 3 allows for this division of tasks. The clear distinction between different tasks furthermore allows for letting different people work on different parts at the same time, without them needing to coordinate their efforts.

These four design goals will guide the design for the rest of the report.

Chapter 5

Design Techniques

To achieve the design goals outlined in Section 4.1, we have decided to make use of several well known techniques for creating software. The first we will describe is *Frameworks*, which is an important technique for re-using software and the architecture behind it. The second we will present is *Event Based Systems*, which are often used as a communication technique between separate objects, classes or modules in games. Finally, we will discuss *Scripting Languages*, and more specifically their role in making applications more user-friendly. These three techniques are the foundation on which the AI framework is designed. In the following we will describe each of them in turn, and discuss the pros and cons of using them as well as trade-offs when choosing to use a particular technique.

5.1 Frameworks

This section will discuss why we have chosen to build an AI framework, explain the alternative and discuss the capabilities of AI frameworks.

5.1.1 Reuse of Software

As one of our design goals is to reuse as much of the AI as possible from one game to another, this section will discuss software reuse in games. Basically, there are two ways of reusing parts of an AI in games: Frameworks or libraries. First, the advantages and disadvantages of both will be outlined.

There is no clear definition of frameworks that everybody in literature agrees on, but one widely accepted definition is the following [FS97] [Bue98] [FSJ97] [JF88]:

> *A framework is a reusable, "semi-complete" application that can be specialised to produce custom applications.*

Frameworks have been used extensively in game development, because they provide several advantages in this area, which stem from four important concepts that frameworks provide: Modularity, reusability, extensibility and inversion of control [FS97]. Within AI framework development, one advantage is that it ensures that the AI is completely separated from the GDF. This means that an AI framework for RTS games will be able to reuse not just single modules, but entire AI architectures [Joh97]. This is also the reason why frameworks tend to be easy to use, because they have well-defined hook methods, which dictate how framework instances differ from each other [FS97]. By using this technique, the user can focus on the areas of the framework that are important for the particular instance being created. Furthermore, by having the architecture in-cooperated into the framework, one can be assured that this part is well-tested, which ensures that frameworks are less error-prone [Lew98]. The disadvantage of using frameworks is that it is often very difficult to change the internal mechanisms of a framework, partly because it is very hard to understand the internal mechanisms and partly because the framework is not built for such a modification [FCGC02]. This problem also reflects in that a framework built for a specific purpose is very hard to use for other things [JF88].

The main advantage of using a library is that it can be used to many different things, as the architecture behind is not included as part of the library [SC95]. Different library components can then be combined in different ways to achieve many different results. This flexibility is also its disadvantage as this limits how much can be reused from one application to another [Joh97]. Compared to a framework, a library also tends to be more difficult to use and it slows development, because the user has to create the architecture herself [JF88]. Furthermore, it tends to be more error-prone because the user herself is responsible for linking and using the different library components in the right way [Bue98].

Following the design goals presented in Section 4.1, the choice between the two for this project is relatively easy. Frameworks are able to provide complete AI solutions, and in a manner that should be relatively easy to use for an AI designer, as well as decrease the time it takes to develop the AI. The use of hook methods and hot spots [ML01] furthermore allows for a structured overview of the development process and the architecture can allow for a clear distinction between different areas. The primary advantage of libraries being flexible is not a requirement, as the focus of this project is solely on RTS game AIs, and we hypothesise that a general architecture can be built to do this.

5.1.2 AI Frameworks

The idea to use frameworks for AI tools in games is not a new one [Lai01]. Most noticeable is the cognitive framework named Soar [Soa] [Lai03] [LL99]. Soar has been used to create AIs in several smaller games and for creating bots in commercial FPS games such as Quake II (1997) [quab] and Descent

3 (1999) [des]. A general description of how Soar has been used to create AIs can be seen in our pre-master thesis [FKL05]. Other AI frameworks have been used in games, but none to the same degree as Soar. Previous research with AI frameworks in games has primarily focused on FPS games or small custom made games [KNYH05] [Lai01]. No one has however, focused on developing AIs for RTS games. There are two reasons for this:

- Research with AI frameworks used in games is a relatively new area.
- RTS game companies have decided not to release source code as rapidly as for instance FPS games and they have not provided programmers with an advanced open AI interface to specify their own AIs.

Recently, open source RTS GDFs have begun to mature into a state where researchers can build their own AIs to existing open source games [ORT05] [Str]. This has opened the opportunity to further research in the development of AIs in RTS games, which have many interesting aspects that other types of games do not, as is discussed in Chapter 1. Open source RTS GDFs have however, still some problems in relation to AI research, which have been identified in our pre-master thesis [FKL05]:

Stability: Some RTS specific GDFs have simply not been stable enough to be used for research purposes, and lacks testing of important functionality.

Documentation: When working with a GDF, the framework must be well-documented for the user to be able to extract the required information. Some frameworks simply lack this documentation.

Full Control of AI: As described in one of our design goals in Section 4.1, the AI framework must be in complete control of all AI actions. Some GDFs only allow for control of the high level AI actions, and has low level AI actions, such as unit movement and tactics, handled internally in the framework.

Not all open source GDFs lack all three identified problems, but at least one of them. Out of the two most mature frameworks, ORTS [ORT05] and Stratagus [Str], ORTS has the two first problems and Stratagus has the third problem.

Previous research within the area of AI frameworks have focused on *cognitive architectures* [LL02]. These are frameworks developed for the purpose of creating and understanding agents that support the same capabilities as humans. They provide a way to define an underlying infrastructure for an intelligent system, and are basically the same as AI frameworks with the intention of providing a platform for emulating human behaviour. John

Laird[1], and Pat Langley[2] have defined a number of capabilities a cognitive architecture could be able to support [LL02]. The capabilities are listed below, including their relation to RTS games. Laird and Langley hypothesise that for an AI to show truly human behaviour, the framework should support all of these capabilities:

Recognition and Categorisation: The capability to recognise for instance strategies or tactics in a game.

Decision Making and Choice: The capability of making both strategic and tactical decisions during a game.

Perception and Situation Assessment: The capability of being able to perceive information and determine the importance of this information.

Prediction and Monitoring: The capability to for instance predict an opponent's future strategy and monitor important variables that may reveal additional information about the enemy.

Problem Solving and Planning: The capability of planning the execution of a certain strategy and solving any problems encountered during execution.

Reasoning and Belief Maintenance: The capability to reason about an opponent's actions and from this determine what she is most likely doing.

Execution and Action: The capability to execute actions in the game.

Interaction and Communication: The capability to communicate with allied players and through this agree on joint strategies.

Remembering, Reflection and Learning: The capability to remember and reflect upon situations that have occurred during a game, and through this learn new strategies or tactics.

The capabilities and their relation to AIs in RTS games are further described in in our pre-master thesis [FKL05]. When describing our AI framework design we will return to these capabilities and discuss how our framework handles each area. When describing how frameworks provide these capabilities, one often talks about four separate areas [LL02]: The representation of knowledge, the organisation of knowledge, how the framework uses

[1] John Laird is a professor of computer science at University of Michigan, general chair for the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE) and one of the developers behind Soar[Soa].

[2] Pat Langley is the director for the Institute for the Study of Learning and Expertise, head of CSLI's Computational Learning Laboratory [CSL] and consulting professor of symbolic systems at Stanford University.

knowledge and finally how the framework supports acquisition and revision of knowledge. These areas have been described in detail in our pre-master thesis [FKL05] and will be discussed further when presenting the overall architecture of our framework in Chapter 6.

5.2 Event Based Systems

This section will present a way to control the different parts in the framework. Each of the parts we have in the framework corresponds to the tasks in the human model, and we will from now on refer to these tasks as modules. There must be created some way to control what modules are executed in what order, and a way to make each of these modules communicate with each other. In this section we will first argue that an event based system is the best way to control the framework, and then we will discuss why event based systems often are used in frameworks and what the advantages are. Afterwards we will present different ways of how an event based system could be designed.

5.2.1 Framework Control

Controlling the execution of framework modules can be done in a number of ways. One way is to make a procedural execution structure, which makes every module call each others sequentially. This is, however, not dynamic, as the developer must have complete knowledge of how the rest of the system is working and which modules influence each other, when adding new modules. If, on the other hand, each module can operate on its own, meaning that it does not need any knowledge about other modules, it can be executed stand-alone. This makes it possible for each module to be called by events by the modules that provide the information required. This makes the module structure much better separated. If additional modules are added at a later point, it is as simple as assigning them to the events that they need, and sending the events in the modules, where the data that should trigger the module, is generated.

5.2.2 Event Based Systems in Frameworks

The use of event based systems in frameworks is far from a new idea[SG86]. Often it is used in GUI frameworks, because the code is only run when the user interacts[SG86] [HNOR88] [CCT89] [jsw05] [Feu97]. In this case it is an advantage that the interaction with the GUI is handled in one place, so that each element does not have to check for interaction. However, as also stated by Hansen *et al.* [HF04], event based systems are far from restricted to this area. Event based systems are also very common in network applications that should only react, when data in sent to them. The different modules in the model should not be run all the time, only in certain periods of the game, or when certain actions have occurred in the game world. Therefore it is advantageous to make the system event based.

In very modular systems it is an advantage to use event based systems because each module does not necessarily need to know of any other module in the framework, but only of the shared data registries and a central event manager that transmits the events to the appropriate modules. To activate each other, a module just has to send an event with the appropriate data, and then it knows that at some point the modules that subscribes to this type of event will be run.

The event manager is assigned the function that should be run when an event occurs. This function will be used to trigger the module into acting on the event, and is also known as an event handler.

In a basic event system there is a central loop that listens for events. When an event is sent, the functions that are assigned to that kind of event is activated. When a lot of events are sent, they might start to queue up, making it necessary to schedule and prioritise what function to run next. When a new event is sent, the event manager can either create a thread with the function that should respond to the new event, or if the already running function has higher priority, stay in the thread of the current function.

An event is basically a notification that some action has happened in some context. Events are also often referred to as messages, which is why event based systems also some times are referred to as message based systems [Sch04]. An event can also be attached data, which could be the result of the action that has happened. An event system can be interpreted as a publisher/subscription system, where there is a producer and a consumer respectively of events. All that is required to create such a system is that the modules can communicate with each other. Ludger *et al.* [FFM03] presents four communication models:

Request/Reply: In this type of communication it is the consumer that initiate the communication, and request the data from the producer. In this model the consumer must know the identity of the producer, because the data is sent directly. This has the disadvantage that each module has to know each other.

Anonymous Request/Reply: This type is the same as the normal request/reply, except that it does not need to know the identity of the producer. The communication is handled by a global event manager.

Callback: Here it is the producer that initiates the communication, sending it directly to the consumer. This means that it has to know the identity of the consumer.

Event-based: This is like the callback model, except that it does not know the identity of the consumer, because this is handled centrally. This way of cooperation in an event based system has big advantages in a loosely coupled modular system, because the modules do not need to know each other. Since it is the producer that knows when an action has been carried out, it makes sense that it is the producer that starts the communication.

As argued by Ludger *et al.* [FFM03], the event-based communication model is the most dynamic and modular approach to handle cooperation between modules. The separated tasks in the human model makes it possible to use the event-based communication in the event system.

5.3 Scripting Languages

Frameworks often become large applications that can take a long time to compile, and even small changes in the vital parts of a framework could mean a complete recompilation of the source code. When tweaking an AI to behave correctly according to what the designer wants, it is often only a few values that have to be changed. These values might just as well be loaded at load time by using a scripting language. [Daw]

Furthermore, scripting languages make it possible for novice programmers or game designers to take on the task of implementing the behaviour of the AI in the game[Hue]. Scripting in games has been used in many ways. It is used to control entire scenes in games, like a script from a movie, where the movement of every "camera" and character are planned in the script. Scripting can also be made more general and just make sure that certain things happen, when the player interacts with the environment in a certain way. This could be what would happen if a button is being pushed or more complex interactions, making sure that a series of actions are done before executing certain parts of the script, which is also known as game logics.

The advantage of letting other people than the programmers write the scripts, is that the people who know how things should work in the game are the designers and by letting the designers write the scripts themselves, nothing will be lost or misinterpreted in the communication between implementor and designer. Also, getting the game logics on a higher level than the rest of the native code[3], helps keeping the framework code clean. With the option of making rapid prototyping, instead of having to compile everything all the time, the scripts can be edited at run-time[Ous98]. It is in this way possible to test and tune the scripts much faster than having to compile and restart the program.

There are of course also drawbacks in using a scripting language. The code is not as fast executed as native code, because it has to be interpreted. Some of this lost performance can be recovered by implementing the complex

[3] The language the GDF is written in.

and computationally heavy algorithms in native code, and just give access to these through an interface in the framework. Scripting languages often use dynamic types and frameworks often use static types. Because the script code is dynamically typed, it has to be run-time type checked, which can add a lot of processing time. The data being transferred from the framework context to the scripting context also takes some resources. Furthermore, it can take some development time to implement and integrate the scripting language into the framework. When that is done, both parts have to be maintained and updated when new features are added to the framework, the developers will have two "systems" to maintain.

5.4 RTS Specific Concepts

In this section we will discuss four new ideas, which are specific to the RTS genre. The first we will present is a new data structure specifically designed to model the concept of strategies, used for learning strategies, counter methods, tricks possible on certain maps etc. We call this data structure the *strategy tree*. In the second part of this section we will discuss a matter very important for all RTS games, namely *pathfinding*. We will discuss the traditional way of doing this and present our own idea on how to optimise this computational heavy calculation. In the third section we will discuss how to represent *tactics* in RTS games. Here we will discuss elements important for tactics in RTS games, and propose a way to define a tactic. Finally, the last section will present a way to create generic *base building templates*, which makes it possible for AI designers to easily define templates specifying optimal building placement in a particular game.

5.4.1 Strategy Trees

The idea of strategy trees came into existence while attempting to discover a method that would be both suited for learning and for representing strategies in RTS games. A detailed discussion of why learning has not been provided with most RTS games can be found in our pre-master thesis [FKL05]. Previous research has primarily focused on training AIs in the development phase of a game [UGJM05] [MSWT05] [JG05] [dJSR05], but not on actually learning after the training has been completed and the game has been shipped. The latter part is one of the goals strategy trees have been designed to achieve.

This section will start by describing a simple RTS game example, and through this illustrate the idea behind the new data structure. Afterwards a more general description will be presented.

Imagine a simple RTS game with three different offensive units available:

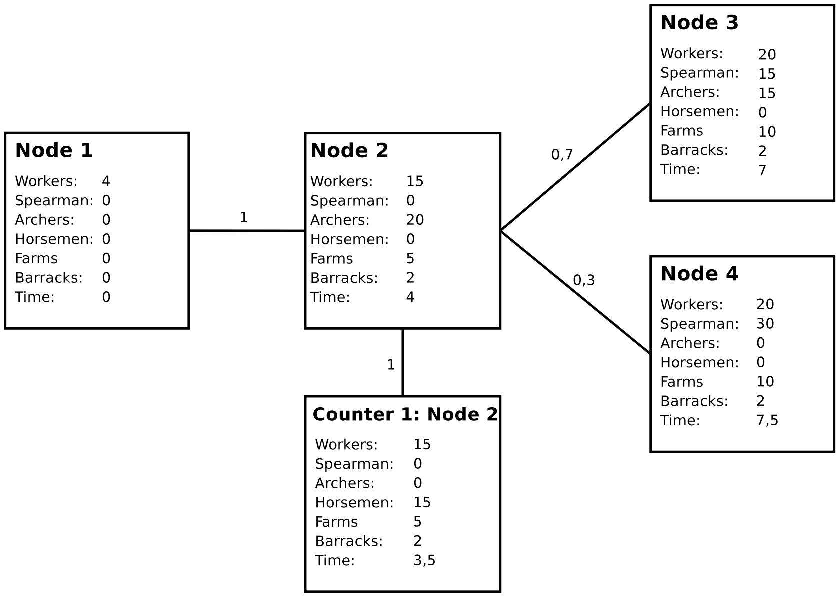


Figure 5.1: Strategy Tree for the Example

Spearmen, *archers*, and *horsemen*. The countering system[4] is as follows:

- Spearmen counter horsemen
- Archers counter spearmen
- Horsemen counter archers

In this simple game each player will start with four workers. The workers can harvest resources and build two types of buildings: Supply buildings (farms) and unit production facilities (barracks). The barracks can produce workers and the offensive units.

The idea behind strategy trees is that an RTS game usually consist of a series of states. The first significant state is the start of the game, and this will be the root node of the strategy tree. As illustrated in Figure 5.1, the root node in the simple example game, *Node 1*, consists of the values of all variables important to the game in question including a time stamp indicating when the strategy was used. In this case the player has four workers and the time stamp is 0, because it is the starting point of the player. Significant states are determined differently from game to game. Often a significant state is characterised as one where the player has carried out a certain strategy and now changes to another. In the strategy tree

[4] RTS games typically implements a countering system. This is basically a system dictating which units are best used against certain other types of units.

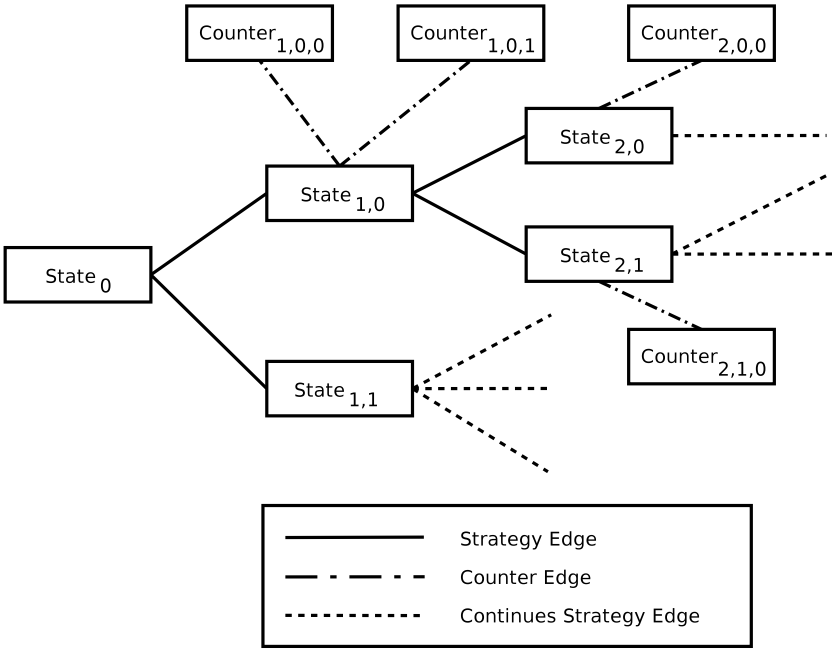


Figure 5.2: Strategy Tree Example

in Figure 5.1, *Node 2* illustrates that the player knows only one strategy following the starting point. This strategy consists of building 15 workers and 20 archers as well as a number of required buildings, and the strategy tree node dictates that the strategy can be reached 4 minutes into the game. From this point in the game, the player knows two strategies to choose from. One of them, *Node 3*, focuses on building a mixed army of spearmen and archers, while the other, *Node 4*, focuses on building a massive amount of spearmen. The numbers attached to each edge between the different nodes represent how often a strategy should be used compared to others at a given node, 1 being always and 0 being never. In this case, the strategy represented by *Node 3* is a more common strategy compared to the strategy represented by *Node 4*. Furthermore, each node can have a special kind of node attached to it as well, representing the counter strategy to the respective strategy. In Figure 5.1, *Node 2* has one counter, *Counter 1: Node 2*, consisting of building a massive amount of horsemen which counters the archers from the strategy in *Node 2*.

The above described how a strategy tree is built and how it represents the possible strategies a player is aware of. The following will provide a more general description of strategy trees and the composition of nodes. Figure 5.2 shows a more generic example of a strategy tree. Each *State* or *Counter* node in the tree consists of the same set of attributes:

Node: A node in a strategy tree consists of the number of each kind of unit the player should have, as well as the number of various buildings she should have. In some games, the node should also contain the different research upgrades purchased. Furthermore, each node has a time stamp that tells how long into the game this strategy was used and a list of tactics used with this strategy.

Two kinds of edges exists in the strategy tree:

Strategy Edge: One kind of edge connects the node to the parent node. Consequently, it also connects the node to its children. This kind of edge is the one that binds the nodes together in the tree structure. Each edge has a probability associated. This probability describes how probable the strategy, modelled in the child nodes, is, compared to the strategy of the other children. This probability is based on the frequency of observation of the different strategies. The edge can also contain a plan for how best to make the transition from one strategy to the next. This would save the planner work, as it does not have to plan the best course of action unless no plan exists.

Counter Edge: The second kind of edge leads to counter nodes. This edge also has a probability associated that is built on a combination of how often the counter has been seen, and the success of the counter.

Strategy trees are perfect for modelling known strategies and learning new strategies. They also present a way of modelling counters. Furthermore, if a strategy tree is maintained for each opponent (modelling the opponent's strategy), this can be used as part of an opponent model. This way, a player can properly reflect on the strategies and counters used by each player during the course of a game. Strategy trees also have the effect that if the AI sees the enemy army or base, it can compare this to nodes in the strategy tree and see several things, for instance: How to counter the current strategy, and which strategy the current strategy will most likely lead to. Referring to the example in Figure 5.1, if the player sees that the opponent has the units and buildings corresponding to *Node 2*, the player can see two things: The optimal counter to this strategy (*Counter 1: Node 2*), and that there is a 70% chance that the opponent will soon be using the strategy represented by *Node 3*, and a 30% chance that the opponent will soon be using the strategy represented by *Node 4*. However, the opponent could also use an entirely new strategy that the player is not aware of. If this is the case, the new strategy should be added to the strategy tree. This is done by adding a new node, a *Node 5*, to the node where the strategy varies from the known strategies, *Node 2*. At this point the probabilities from *Node 2* to its children should be re-adjusted, as it now has three children instead of just two. This way strategy trees easily support learning new strategies observed from the opponent. As strategy trees is one of the RTS specific ideas that we plan to implement, we will return to them when evaluating the prototype implementation of the framework in Chapter 8.

5.4.2 Pathfinding

The largest task usually handled by the AI is the task of finding a path from starting point to the goal for all units in the game[BMS04]. To understand the full implication of this task a series of facts must be taken into account. In most RTS games all players are in control of an army consisting of a number of units. The players or AIs move their army around in order to scout, attack and defend. Each of these units must be assigned a path from their starting point to the point they are ordered to. This means that a single move order issued by the player or AI can mean that a path must be calculated for hundreds of units. The path is found from the place the units is currently situated to a designated goal location. An example of a typical map can be seen in Figure 5.7. *Clusters* form a grid of dotted lines. In Figure 5.6 one such cluster can be seen. For this example the cluster is made of 16*16 cells. The path that must be found is a list of connected cells (cells that are adjacent to the ones next to them in the list) that start at the start location and end at the goal location. We define a low granularity as using clusters for the abstraction and high granularity as using cells for the abstraction. The abstraction that clusters provide will later be shown to be useful to reduce the search space explored in the pathfinding. This list must be found in a matter of moments to ensure fast response to the game. Furthermore it is not enough to find just any path that will take the unit from the starting point to the goal, the path must also be the shortest path possible.

This means that building a pathfinder is often a balance between the complex task of finding an optimal path from A to B and doing this in the computationally cheapest way possible.

A*

A* is a best-first search algorithm that is very popular and has been used in many variations in both the academic world and in the game development industry [BMS04]. Using a heuristic defined by the developer, it will find the shortest path if one such exist. Mono-directional search using A* will usually result in an exploration of the search space much like the one depicted in Figure 5.3. Compared to the search space explored in for instance breadth-first search the heuristic ensures a noticeable optimisation of the number of cells explored in the search. This can be optimised even more using bidirectional search as seen in figure 5.4. The fewer cells visited in the search does not just mean that the amount of memory used in the search is

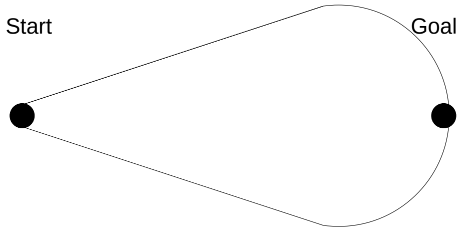


Figure 5.3: Search space explored using A*

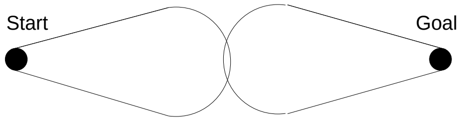


Figure 5.4: Search space explored using bidirectional A*

minimised, but it consequently also minimises the time it takes to compute the path. Both are the main points of focus for optimisation.

Hierarchical Search

As already mentioned there are two main concerns when designing a pathfinder: The computation time and the memory use. The two are not independent, but are on the contrary quite closely linked. Exploring a minimum of cells will for instance mean a minimum of computation time. The idea behind hierarchical search is to reduce the search space on grid-based maps by working on multiple levels on granularity. High granularity means a grid of cells and low granularity means a grid of groups of cells. If an optimal path can be found on a low granularity map, cells outside this path can be disregarded when constructing a path of a higher granularity. Botea *et al.* [BMS04] present one such algorithm named HPA* (Hierarchical Path-Finding A*). HPA* first systematically divides the map up into a grid of clusters. It then determines *entrances* between the adjacent clusters, which are obstacle-free common borders between the clusters as seen as the marked areas in Figure 5.5. The entrances are used to build an *abstract problem path* which is an optimal path from start to goal consisting of the entrances that the optimal path will pass through. The last step is to find the actual path between the entrances on the highest level of granularity. The actual processing is done by first finding the abstract problem path and then doing pathfinding on the first sub-path (the path between the first two entrances in the abstract problem path). By restricting the high granularity pathfinding to the first subproblem, the pathfinder will have the first few seconds of a units movement ready fast so that the unit can start moving while the

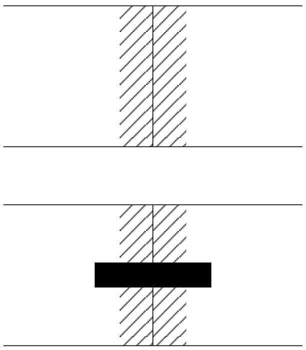


Figure 5.5: Entrances in HPA*

pathfinder computes the rest of the path.

Hierarchical JIT Pathfinder

The pathfinder we propose share the same basic principles as the one just presented. The low granularity grid will be maintained at the time of exploration, or if the map is already explored, at loading time. Instead of entrances, each cluster in the low granularity grid will share four *passable nodes*[5] with the adjacent clusters: One each at top, bottom, left and right side (see figure 5.6). There will exist a connection between two nodes if the cluster is passable from one node to the other (not unlike entrances). That means that top and bottom will be connected if a unit will be able to cross the cluster vertically. That is, the path from top to bottom must not excess for instance 1.5 times the direct path. If any obstacle is discovered this must be updated on the low granularity grid.

The pathfinding itself starts by determining a path from start to goal consisting of passable nodes. The path consisting of the clusters is called the *passable path*. The passable path is used to reduce the search space much like the abstract problem path seen in HPA*. The developer will be able to tune the length of the sub-path by defining the number of clusters this should contain. Each sub-path can be processed like in HPA*, that is in a Just In Time (JIT) fashion. In Figure 5.7 an example of a map can be seen. The example is a situation where a path is found moving around a ledge obstacle. The ledge is depicted as a broad dotted line. The trees are also considered obstacles in this example. The grid of clusters is depicted in thin dotted lines and lines have been drawn as edges between passable nodes. Using the passable path seen as a dotted line between the two "X" markings, it is possible to restrict the search space to the clusters (marked

[5] A passable node is a special cell that is placed at the border between two clusters of cells

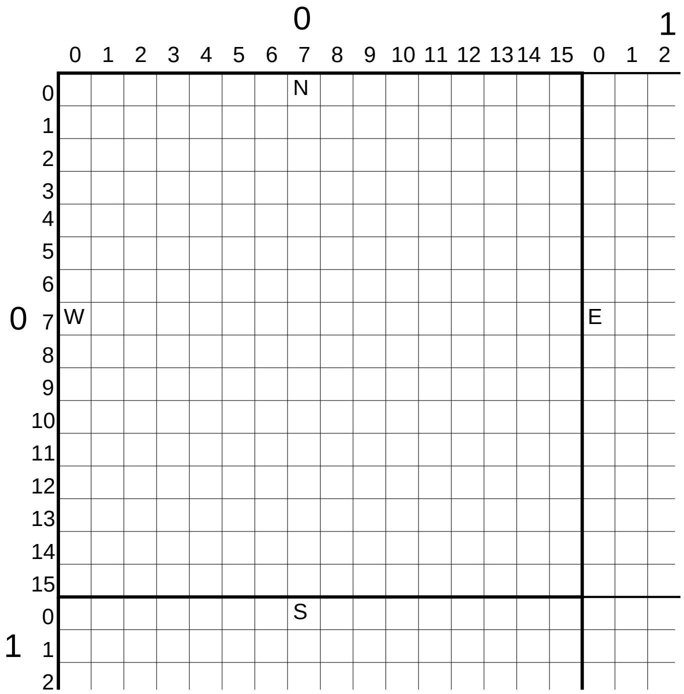


Figure 5.6: Placement of Passable nodes

in gray) found in the passable path.

Theoretical Examples

The environment used in this example is chosen to show the worst possible scenario for A*, and the best possible scenario for the pathfinder used in this project. The map is a flat fully visible grid of 1024*1024 cells without obstacles, where the pathfinder must find a path from the top left corner to the bottom right corner, that is diagonally across the map. The chosen size for clusters is 16*16 cells and the grid of clusters is thus 64*64.

An ordinary A* will find an optimal path from start to goal but in the process of doing so it will explore almost all the cells on the map. That means that the A* pathfinder will examine 1024*1024 or 1048576 cells in total. The search space can be seen at the left side of Figure 5.8.

Using the passable path for restricting the search space, the pathfinder used in this project will only explore the nodes found in the marked path seen at the right side of Figure 5.8. This path contains 64+63 clusters of cells making it a total of 32512 cells expanded.

As seen in the example the hierarchical pathfinder will explore up to 1/32 of the amount cells compared to an ordinary A*. This number can be further minimised by choosing a smaller size for the clusters but this has to

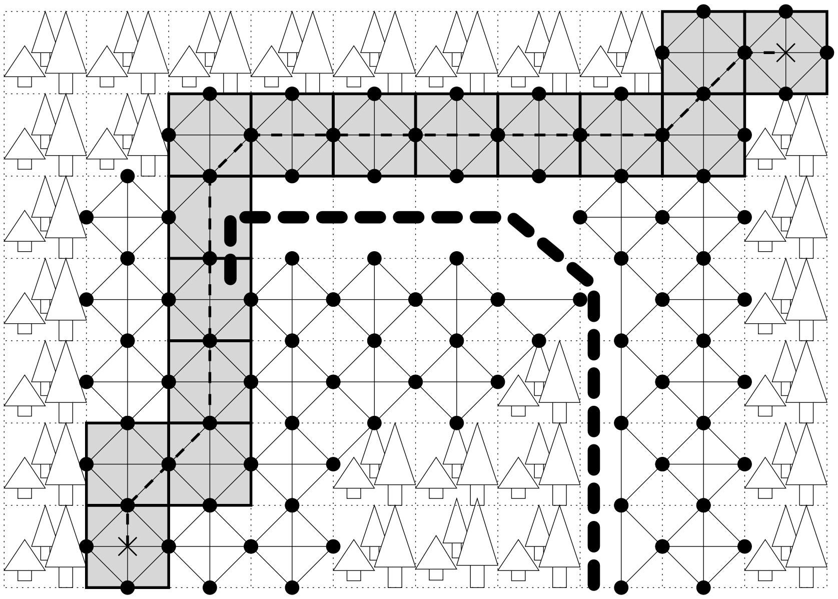


Figure 5.7: Passable Example

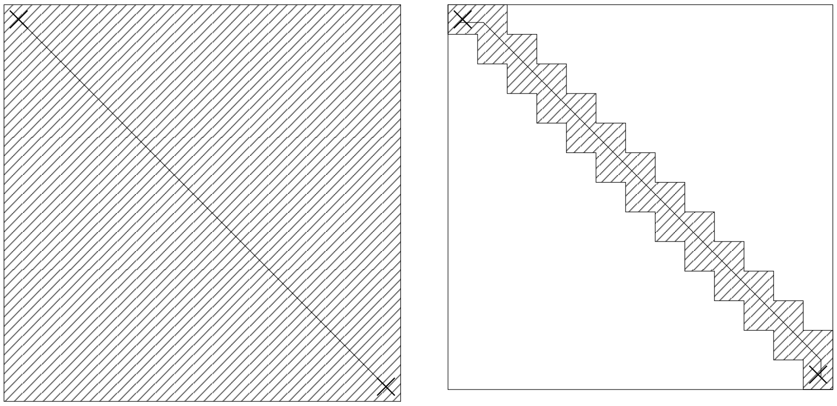


Figure 5.8: Search spaces for A* and Hierarchical search

be balanced with the effort of calculating the passable path.

The best possible scenario for A* and the worst possible scenario for hierarchical search is when the start cell and the goal cell are adjacent to each other. In this case the A* will terminate after exploring the goal state, but the hierarchical search will first have to determine a passable path. This means that the hierarchical search will explore four nodes instead of two nodes (two nodes in the passable search and two nodes in the actual path search).

In order for the hierarchical search to be better the goal cell must be in a different cluster than the start cell. This is not a problem because most of the movements done in RTS games will be from the AI's own base to the enemy base or from harvesting buildings to the resources being harvested. In both cases the path will cross several clusters.

Based on this we hypothesis that a hierarchical pathfinder will be able to reduce both the computation time and the memory use in a dynamic environment. The impact will most likely grow with the number of obstacles and the length of the path. This will be tested through the prototype.

5.4.3 Tactics

This section will discuss how to represent tactics in RTS games. Throughout this section, a lot of RTS specific terms and expressions will be used, and the reader is referred to Appendix A for an explanation of these. As already discussed in Section 5.4.1, a strategy tree node can contain tactics concerning how to execute a certain strategy. We define tactics as being the part that defines how to control units during a battle. That is, it does not decide where on the map to attack or evaluate whether the AI is in a losing battle.

In previous research on tactical planning, Reece *et al.* [RKD00] present the project DISAF that works with the tactical movement of individual soldiers in a complex environment. Among other things they present two kinds of decompositions of the problem: A distance-detail decomposition and an environment decomposition. Both help to reach a high abstraction level so that a pathfinding algorithm can be implemented to handle the problem. Burgess [Bur03a] works with terrain analysis, identifying avenues of approach, and deployment of forces which can be adjusted to react to new information (for instance enemy sightings). Both papers are written in cooperation with the US army. Although real-life tactical situations have some resemblance to the tactical situations occurring in RTS games, the previous work on tactical planning is difficult to reuse when trying to develop a method for representing tactics. This does however, not mean that some of the ideas in these papers cannot be reused when executing tactics within a *Tactical Planner*. The representation of tactics are however a different matter, and this representation must be both developer friendly and expressive enough to deal with very advanced tactics within any RTS game genre. The basic responsibilities of a tactic in RTS games is to define the following:

Units: The tactic must specify which unit types and how many of each are needed to successfully carrying out the tactic.

Formations: A tactic must define how different units should position themselves in relation to each other and in relation to the enemy's units. A formation could for instance place units with a lot of hitpoints in the front of the army, while ranged and weaker units could stand protected behind these. The tactic should also specify whether the formation should be kept at all times during battle, or if it should mainly be considered a good starting position before a battle.

Focus Fire: In most games, some units will have higher attack damage on certain kinds of units compared to others. A tactic should include rules for which units should focus on which type of enemy units. Furthermore, the tactic should define rules for how many should focus fire on a certain unit at a time and if enemy units withdrawn from battle should be pursued.

Unit Preserving: Tactics should also specify when to withdraw units from battle. This may vary for different unit types, as weak units should probably react faster when receiving damage. Unit preserving tactics furthermore includes the decision of where to send withdrawing units. This could be the main base or just without of enemy range and then back into the battle again.

Using Support: A tactic should specify how to use support units, and more specifically the spells or abilities they have. This includes decisions regarding who should receive buffs and debuffs as well as how area of effect spells or abilities should be used. Furthermore, as some spells or abilities have limited use, the tactic should define when and under what conditions these should be used.

Using Terrain: Several games includes terrain specific advantages during battles. This could be things such as using choke points, high ground etc. A tactic should dictate how to take advantage of different terrain features.

Creating a representation of tactics usable in many different RTS games is a difficult task. The goals of this representation is as follows:

Generic: The representation should be able to represent tactics in all of the supported RTS games.

Versatile: The representation should allow for a tactic to be easily adapted to suit a specific game.

A tactic consists of the six responsibilities mentioned earlier, and it is logically to represent it using these distinct areas. That is, a tactic consists of the unit types necessary, rules for how units should stand in formation, rules for focus fire, rules for anti-focus fire, rules for using support units and rules for using terrain advantages. The idea behind this tactic representation is to use small building blocks of rules to composite larger and more complex tactics. We will explain the idea of the representation through a typical standard tactic usable in most RTS games. The tactic is called a *Siege* tactic, and basically consists of having siege units attacking the enemy base, while these are protected by a number of other units. This has the advantage that it forces the enemy to come out of his base (thus leaving the advantage of any base defence) and attack the player, who has placed her units in an advantageous position to deal with this attack.

First, we will describe in general terms how each of a tactic's responsibilities could be defined for the *Siege* tactic. Then we will discuss an attempt to express the tactic in a simple generic way. This will not be a full discussion on the topic as it is much more advanced than the example presented, but it will serve as a proof of concept that a generic representation is possible. Finally, we will briefly discuss how this tactic could be used in different kind of RTS games. The following presents the details of the *Siege* tactic:

Units: This tactic requires as a minimum two types of units. First of all, siege units are needed to attack the enemy base from a distance, and secondly, a group of units are needed to protect these siege units. For simplicity, we will assume that the group of non-siege units are all melee units.

Formations: A formation in this tactic should ensure that the siege units are properly protected by the melee units. This means they should be positioned between the siege units and the enemy units or base.

Focus Fire: As such, there do not need to be any focus fire rules for the melee units as their primary focus will be to hold formation and thereby protect the siege units. The siege units on the other hand, should primarily focus on defensive structures in the enemy base, or buildings important for unit production.

Unit Preserving: To avoid hit and run attacks from the enemy killing wounded units, units should be sent back to the main base when they have lower than 20% of the maximum health.

Using Support: For this example, we will assume that there are no support units and that none of the other units have special spells/abilities.

Using Terrain: For simplicity, we assume that there will be no tactical terrain considerations when using this tactic.

Consider a simple approach for defining this tactic shown in Listing 5.1. This approach assumes that the AI knows about which units are considered siege units, melee units etc. and that it knows which buildings are defensive structures, unit production facilities etc. Each line in the listing describes a rule used with this tactic, and their meaning will be explained in the following. *Line 1* defines the tactic named *Siege*. *Line 2* and *line 3* defines which units are to be used with this tactics. The numbers indicate their group id, and as can be seen, group 1 consists of all siege units, while group 2 consists of all melee units. The AI can itself figure out exactly which units to place in each group as it knows which are considered melee units and which are considered siege units. These group id's can later be used to specify individual rules for each group. *Line 4* specifies the formation these units should be in. Units groups are listed with the first being closest to the enemy units or buildings. This means that in this case, the melee units are standing in the front, while the siege units are behind them. Furthermore, one can specify the importance of units keeping the formation during battle. In this case, the melee units should always keep formation in order to protect the siege units, while the siege units can move around freely (the melee units will still protect them while moving). *Line 5* adds a focus fire rule for group 1, the siege units. The rule simply states a prioritised list of units on which to fire upon first. In this simple example we have decided not to include things like how many units should focus fire at the same building at a time, but it could relatively easily be added at a later time. A similar rule could be added for each group in the tactic if necessary. The two final lines, *line 6* and *line 7* simply dictate a unit preserving rule for both group 1 and group 2. They state that a unit should try to run back to its own main base if it is down to only 20% of its original hitpoints. All of this assumes that the framework has internal operations that can handle the execution of the different rules. However, given the information in for instance the *add_focusfire_rule*, it should be relatively easy to provide these operations.

Listing 5.1: Tactic Template

```
1 Siege = Tactic()
2 Siege.add_unit_group(1, siege_units)
3 Siege.add_unit_group(2, melee_units)
4 Siege.formation([(2, strict), (1, loose)])
5 Siege.add_focusfire_rule(1, [defensive_structures, unit_production_facilities])
6 Siege.add_preserving_rule(1, (20, main_base))
7 Siege.add_preserving_rule(2, (20, main_base))
```

Even this simple approach to the *Siege* tactic can be useful in many different kind of RTS games. In *Starcraft* the tactic could be used with *marines* and *siege tanks*, in *Warcraft III* the tactic could be used with *grunts* and *demolishers* and in *Age of Mythology* the tactic could be used with *Axemen* and *catapults*.

The approach presented in this section is in no way a fully complete representation. Further work on this must be done in order to fully represent all kinds of tactics in RTS games. We hypothesise however, through the example described in this section, that it is possible to create a generic representation which will be usable to represent tactics possible in many different RTS games. The representation will not be tested in the prototype implementation, because of the idea not being fully developed and because of limitations of the test game. These limitations are presented in Chapter 7.2.2.

5.4.4 Base Building Templates

In this section we will present the idea of *base building templates*, for determining how the layout of a base should be. To our knowledge no related work exists dealing with this subject. What can be seen in games today is that there is either a fixed layout that is used, which is tweaked to fit the terrain, or the buildings is placed at random where there is room for them. First we will present the main idea, then an example of a base layout, and then an example of how this could be modelled in the templates.

To control the layout of the base in accordance to the strategy that is used, we have designed what we call *base building templates*. These will dictate how the base is constructed, taking into account the terrain, resources in the area, and the strategy used. Just like with the strategy tree the future development of the base must be expressed in the *base building template tree*.

On each strategy node in the strategy tree the user of the framework must specify what kind of base buildings templates she determines is the best for that strategy. The base building templates create the tree of how the base could evolve, themselves. In each of the nodes a parameter specifies what buildings that should be used for perimeter and what buildings that should be protected.

It should furthermore be possible to create functions that specify how to place buildings in special cases. Inside these functions a lot of different tools must be available to the user, such as influence maps[Sch04][Del01][6], complete information about the already placed buildings, and other map related information such as terrain and positions of resources. The influence maps give access to features where the developer can specify what weight different terrain types and resources should have as well as being able to define the propagation function, and tell how each layer in the map should be combined.

An example of a layout of a defensive base in Warcraft III can be seen in Figure 5.9. Here the buildings with high hitpoints are placed at the front, because they work as a perimeter. The *Town Hall* is placed close to the *Mine*, so that the distance that the workers have to go is not long, but it

[6] A further description of influence maps are given in the pre-master thesis [FKL05].

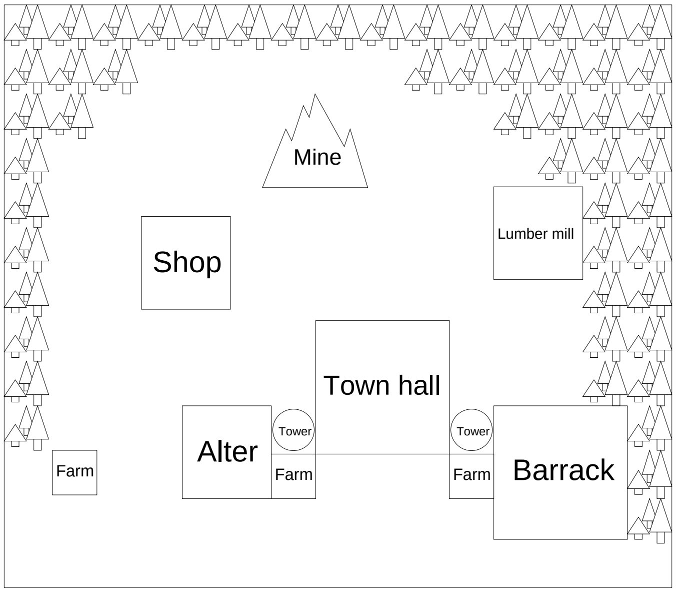


Figure 5.9: A defensive base layout in Warcraft III

is also placed at the perimeter, because it is a high hitpoint building. Two towers are placed right next to the *Town Hall* so that they can defend the building, but they are also centrally placed so that they can attack any units that try to attack any other buildings in the base. The *Tower* buildings do not have that may hitpoints so they are placed behind the base perimeter. The *Lumber Mill* is placed close to the resource that the workers harvest for it, and is placed behind the base barriers, because it has relatively low hitpoints, and the workers would be vulnerable when gathering wood. The *Alter* and *Barracks* are high hitpoint buildings, so they are used as barriers. *Farms* are also used as perimeter, not because they are strong, but because they are fast to build, cheap, and the price to hitpoint ratio makes it a cheap barrier building.

This example of how the layout of the base could be written in base building templates syntax is presented in Listing 5.2. The module using the templates must be able to find the best place for the centre of the base by using influence maps. Furthermore, it must be able, again with the use of influence maps, to find what perimeters that does not need to be defended, because of for instance terrain or resources that work as perimeter.

The module should find the best place to build harvesting buildings are, and create influence maps where the placement of these affect how the rest of the template is built. What buildings that are harvesting buildings is not necessary to specify in the template, because this is known from the technology tree. *Line 1* defines the base building template named *Defensive*. On *Line 2* it is assigned a name, to describe what kind of base layout it is.

The *town hall* is set to be the central building in the base on *Line 3*. *Line 4* defines that there should only be one opening in the base. On *Line 5* it is specified that the building types that are added to `protect buildings` should be placed within the base, behind the barriers. The add `buildorder` to the template on *Line 6* and *7* is used as a guide to what buildings to build first. If none is specified, the order will be found while playing the game. A base building template node must only contain the order of those buildings that have been added additionally from the last node. The building types in `barrier buildings` added on *Line 8* will be used for constructing the barriers. The type of `defensive buildings` that can actively defend the base is added on *Line 9*.

Listing 5.2: Base building template node

```
1 Defensive = Base_building_template()
2 Defensive.set_name("Single␣opening␣defensive␣layout")
3 Defensive.set_central_building(town_hall)
4 Defensive.set_number_of_openings( 1 )
5 Defensive.add_protect_buildings([town_hall, lumber_mill, shop, tower])
6 Defensive.add_buildorder([town_hall, farm, farm, alter, barrack, lumber_mill,
7      farm, tower, tower, shop])
8 Defensive.add_barrier_buildings([barrack, alter, farm, town_hall])
9 Defensive.add_defensive_buildings([tower])
```

5.4.5 Summary

This section has presented four RTS specific concepts specifically designed to create AIs in this genre. We presented the idea of strategy trees, as a data structure built for representing strategies in RTS games. Then we discussed the issue of pathfinding, and presented an optimised way of doing this in an RTS game environment. A preliminary design of a representation of tactics in RTS games were also presented, along with the idea of base building templates, used for defining optimal building placement for the AI. Only strategy trees and our pathfinding idea will be tested in the prototype implementation.

Chapter 6

Framework Design

This chapter will focus on the design of the AI framework. An illustration of the overall design can be seen in Figure 6.1. The framework architecture is built under the assumption that it is completely separated from the GDF and that the framework is in complete control of all AI actions. This means that the architecture must handle all AI activity. Internally in the framework, an event system decides which modules are executed and each module in the framework is configured through a script.

We will start by presenting the architecture of the framework, by discussing how one can transform the human model presented in Chapter 3 into a cognitive framework architecture. With each module we will present its responsibilities, how it communicates with other modules and how it fulfils one or more of the framework capabilities defined in Section 5.1. Afterwards, we will present how knowledge is represented in the framework. This includes a discussion of how knowledge may be represented in each of the knowledge bases defined in the human model. Furthermore, we will discuss how the different representations of knowledge dictates the organisation of knowledge in the framework. We will then discuss where and how hot spots

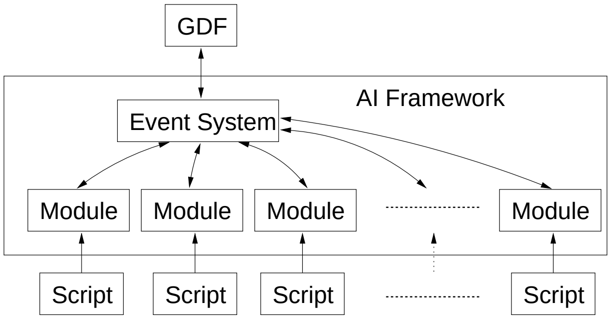


Figure 6.1: Overall design of the framework

are represented, and hereby explain how we allow framework instances to differ. Finally, the last section will present the event system controlling the framework and explain how a scripting language is used with the framework.

We have decided not to include any cooperation features in the design, because it would bring unwanted complexity to a design that is already more than capable of proving whether the idea of an RTS framework is a good idea. This means that the *Cooperation* task discussed in Chapter 3, will not be considered when designing the architecture of the framework.

6.1 Framework Architecture

When designing the cognitive architecture of the framework, it is natural to look at the human model once again. Using the human model as a starting point, many of these tasks can be reused, some must be separated into several modules, while others can simply be in-cooperated in other modules. We will start by presenting these changes and then present the overall architecture along with a description of each module. This description will include a presentation of each module's specific responsibility as well as a reference to how they fulfil one or more of the framework capabilities presented in Section 5.1.

6.1.1 Cognitive Architecture

Compared to the human model in Chapter 3, several changes has been made. First of all, the *Scouting* task has become a part of the *Strategic Planning* task, because it is a relatively small task compared to others and because it is so closely linked with *Strategic Planning*. The *Micromanagement* task has been divided into two modules: *Tactical Planning* and a *Reactive Module*. The latter being a new module to handle urgent actions such as withdrawing units from battle because of focus fire. The reason for this division is that micromanagement is so closely linked to tactical planning that these cannot be handled separately. However, there are still some actions that should be carried out instantly, which is what the *Reactive Module* is intended to handle.

The two tasks *Opponent Modelling* and *Reasoning* has been divided into two new modules that better represents what their responsibilities are: *Probabilistic Reasoning* and *Pattern Recognition*. The *Probabilistic Reasoning* module takes care of all calculations concerning the opponent's strategy as well as maintaining belief knowledge in the *Opponent Model*. The *Pattern Recognition* module has two responsibilities: Updating the *Opponent Model* with new information and recognising strategies and tactics used by the opponent. Finally, two new modules have been added to ease communication between the AI framework and the GDF. These are a *Percept Interpreter* module and an *Action Planner* module. The *Percept Interpreter* module is

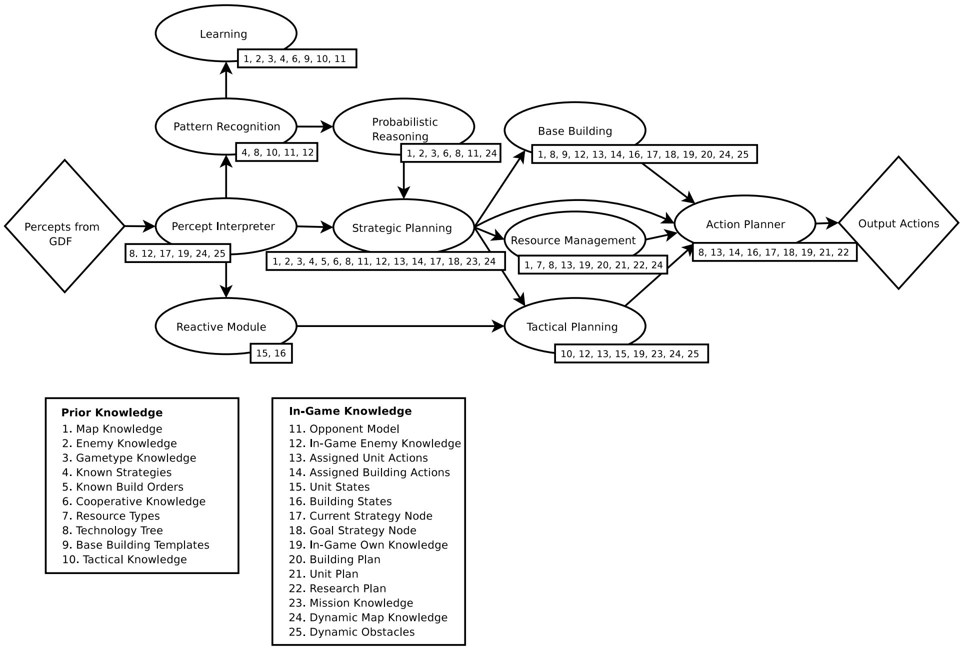


Figure 6.2: The cognitive architecture of the framework

added to handle all input from the GDF and is responsible for updating all the appropriate knowledge bases in the AI framework. The *Action Planner* on the other hand, is responsible for handling all output from the AI framework to the GDF. Furthermore, it is responsible for deciding in which order actions are to be executed and how to prioritise actions according to the resources available.

Figure 6.2 (a larger version can be seen in Appendix K.2) shows the cognitive architecture of our AI framework. Circles represents framework modules and arrows represents data flow from one module to another. The diamonds represents communication with the GDF. Each module has a small box with numbers attached to it, which shows which knowledge bases the module uses. The numbers correspond to the prior knowledge and in-game knowledge tables at the bottom of the figure. Note that some of the knowledge bases have not yet been introduced, as this will first happen in Section 6.2, but for easy referencing, a short description of all of them can be found in Appendix C.

6.1.2 Modules

The following will present the responsibility of each module and relationship with other modules:

Percept Interpreter: Percepts can take the form of simply being the game state at each decision cycle, or a direct message sent to the AI such as "Your base is under attack" known from many RTS games. This module takes care of translating percepts to a form usable for *Pattern Recognition* methods as well as providing the *Reactive Module* with the necessary data, including information about damaged units and native AI events. Furthermore, this module takes care of updating several of the in-game knowledge bases: *In-Game Enemy Knowledge*, *Current Strategy Node*, *In-Game Own Knowledge*, *Dynamic Map Knowledge*, and *Dynamic Obstacles*. The module corresponds to a part of the *Perception and Situation Assessment* capability presented in Section 5.1. Further details on this module can be found in Appendix B.1.

Reactive Module: As explained earlier, the *Micromanagement* task has been removed and replaced by a *Reactive Module*. This module takes care of all low level unit reactions and it handles all native AI[1] events. The degree of reactiveness should be left to the AI designer as its importance is game specific and likewise it should be user-specific how to handle different native AI events. The module will monitor how much damage units and buildings are dealt over time, and make sure that the *Unit State* and *Building State* in-game knowledge bases are updated. A detailed discussion of how this module is designed can be found in Appendix B.2.

Pattern Recognition: The main responsibilities of the *Pattern Recognition* module have already been mentioned: Updating the *Opponent Model* and recognising strategies and tactics. Updating the *Opponent Model* includes keeping track of enemy units disappearing into fog of war and recognising tactics used. To recognise tactics, the user must specify special recognising methods based on *Tactical Knowledge*. Recognising strategies on the other hand, can be handled by the framework, as this is basically a matter of matching a strategy node with the strategy tree defined in *Known Strategies*. This module will furthermore during a game, keep track of a strategy tree describing which stages the opponent's strategy has gone through, which will be a big help later for the *Learning* module, when adding new strategies to the AI's repertoire. According to the capabilities described in Section 5.1, the *Pattern Recognition* module takes care of two capabilities: *Recognition and Categorisation* and partly *Prediction and Monitoring*. Details describing the design of this module can be found in Appendix B.3.

[1] The native AI is the built-in reactive AI on each unit in a game. When a unit for instance is attacked and then tries to find and attack the enemy attacking it, it is the native AI reacting. This is often a problem in RTS games, because a player can lure parts of an army away taking advantage of this unit's native AI.

Learning: The architecture supports two types of learning: Knowledge acquisition and knowledge refinement. Knowledge acquisition happens when the AI should learn strategies, tactics or base building templates. Adding a new strategy to strategy trees is a relatively simple operation as the data structure makes it easy to do so. Learning new tactics or base building templates is a more complicated process, and is described in further detail, along with the rest of the *Learning* module, in Appendix B.9. The *Learning* module must furthermore evaluate the success of strategies, tactics and base building templates and modify the strategy tree knowledge bases, *Tactical Knowledge* and *Base Building Templates* accordingly. Finally, it must keep track of different opponents by updating their corresponding strategy tree in *Enemy Knowledge*. The framework in-cooperates so-called *lazy learning* [LL02] as it is most sensible to reason about the development and strategies of a game after the game has ended. Furthermore, some learning methods take a lot of CPU time, which is unwanted during game as less time can then be dedicated to finding the right strategies and actions. This module takes care of the *Remembering, Reflection and Learning* capability discussed in Section 5.1.

Probabilistic Reasoning: The main tasks of this module is determining the opponent's strategy and figuring out potential follow-up strategies. Determining the opponent's strategy is a matter of comparing the *Opponent Model* with strategy nodes in the different strategy trees. Different optimisation methods can be used to do this as is described in Appendix B.4. Finding follow-up strategies is a relatively simple task, as strategy trees has direct support for this operation. Besides these two tasks, the module must take care of updating the *Opponent Model* with new belief information about the opponent's strategy and determining important variables that may give away the opponent's strategy given that the module finds that there are more than one possible. The *Probabilistic Reasoning* module makes use of primarily *Inductive Reasoning* by going from specific observations about the enemy to more general beliefs about her current and upcoming strategy. This module takes care of the *Reasoning and Belief Maintenance* capabilities mentioned in Section 5.1.

Strategic Planning: The *Strategic Planning* module takes on the task of choosing an overall strategy for the AI. It can do this partly by using different knowledge bases, and partly by using the *Reasoning* module to provide reliable information on which to base decisions. The module must go through two phases: First, an overall strategy must be selected and second, units must be assigned different tasks to properly execute the strategy. Besides this, the module is also responsible of scouting and planning where to expand if the strategy dictates this. A further discussion of each of the tasks the *Strategic Planning* module must take care of, can be found in Appendix B.5. The module provides means for the *Decision Making and Choice* and the *Perception and Situation Assessment* capabilities presented in Section 5.1.

Tactical Planning: This module is essentially responsible for all unit actions that are not directly related to a resource gathering or base building activity. This can be further divided into two parts: Unit movement and unit engagement. Unit movement includes pathfinding and avoiding walking through enemy armies. This area will partly be covered by the pathfinding idea mentioned in Section 5.4.2. Unit engagement includes all micromanagement details such as focus fire, using support units and withdrawing damaged units from battle. To do this, the module will rely on the representation of tactics presented in Section 5.4.3. The module is furthermore responsible for deciding when to withdraw an army from a losing battle, for keeping armies in formation while moving, and for making detailed terrain analysis of the battlefield. More details on the design of the *Tactical Planning* module can be found in Appendix B.6. This module handles two of the cognitive framework capabilities presented in Section 5.1: *Decision Making and Choice* and *Perception and Situation Assessment*.

Base Building: The *Base Building* module is responsible for planning base building placement and determining building priorities. Given a strategy from the *Strategic Planning* module containing buildings to be built, the *Base Building* module must prioritise which to build first and where to build them. The latter must consider optimal placement of all the buildings that must be built, and must thus both include a certain measure of anticipation as well as a terrain analysis of the area surrounding the main base. Furthermore, the module must take action when the *Building State* in-game knowledge base shows that a building is in critical health and repairing is necessary. Details on the complete design of this module can be found in Appendix B.8. This module primarily focuses on the *Problem Solving and Planning* capability defined in section 5.1

Resource Management: This module has four different responsibilities. Given a strategy from the *Strategic Planning* module it must determine resource requirements and determine a resource gathering plan that allows for the strategy to be executed as fast as possible. One factor in this determination is a resource analysis, which considers which and how many resources are left at different locations. This way, the *Resource Management* module will decide if an expansion is necessary. Afterwards, the module must assign all workers to the appropriate resources. The final task of the *Resource Management* module is to ensure that resource gathering happens at a optimal rate. This includes not only optimal pathfinding, but also having an optimal number of workers harvesting the different resources. The details of the design of this module can be found in Appendix B.7. The module is primarily contributing to the *Problem Solving and Planning* capability presented in Section 5.1.

Action Planner: The *Action Planner* module is responsible for scheduling actions and communicating the chosen actions to the GDF. Scheduling of actions are important when the AI player do not have enough resources to carry out all the wanted actions. It is this way also responsible for delaying resource spending if the AI is required to carry out an expensive action later in the game. The module is furthermore responsible for producing two plans: *Unit Plan* and *Research Plan*. The *Unit Plan* contains information about which units are to be built next, and the *Research Plan* contains information about which research upgrades to purchase, and when this should happen. The *Action Planner* is the only module in the framework, which is allowed to output actions to the GDF. Details on the design of this module can be found in Appendix B.10. The *Action Planner* handles the *Execution and Action* capability presented in Section 5.1

6.2 Representation of Knowledge

To discuss in detail how each knowledge base should be represented in the framework, some of the knowledge bases defined in the human model must be divided into smaller knowledge bases. We will start by introducing the new knowledge bases and then move on to discuss how each knowledge base is represented. The reader is once again referred to the illustration of the cognitive architecture of the framework in Appendix K.2 to see the role of each knowledge base in the architecture.

6.2.1 Division of Knowledge Bases

Only one of the prior knowledge bases must be divided into smaller knowledge bases and this is *Game Specific Knowledge*. This knowledge base is divided into the following:

Resource Types: This knowledge base defines what kind of resources are available in the game.

Technology Tree: This knowledge base defines game specific building dependencies, unit dependencies and research dependencies, as well as resource cost for everything in the tree. Furthermore, it includes knowledge about what actions each unit or building is capable of.

Base Building Templates: Contains templates for structuring base building. These templates also contain a prioritised list of buildings to build first for each building plan.

Tactical Knowledge: A knowledge base describing all tactics possible in a certain game. These are essentially also present in the *Known Strategies* knowledge base, but is here hidden within the different strategy nodes. This knowledge base is basically for easy referencing the different kinds of tactics.

All of the in-game knowledge bases have been divided into smaller knowledge bases as well to get a clearer overview of what each of them consists of:

In-Game Enemy Knowledge:

- **Opponent Model:** Contains information about the current strategy of the enemy, including a strategy tree and current node information for the enemy. It also specifies beliefs about the number of units and buildings the enemy has. The beliefs are only valid if the attribute in question have not been scouted, and they are only there to represent what the AI currently thinks the opponent is doing. All updates includes a time stamp, which allow the AI to give less importance to variables not updated for a long time.
- **In-Game Enemy Knowledge:** Contains the position of each enemy unit currently visible on the map and knowledge about where certain units have been seen earlier (So the AI does not forget enemy units when they enter fog of war)

Unit and Building Information:

- **Assigned Unit Actions:** Information about each controlled unit and the current action assigned to it.
- **Assigned Building Actions:** Information about each controlled building and the current action assigned to it.
- **Unit State:** Contains a collection of all controlled units and the state each of them are in.
- **Building State:** Contains a collection of all controlled buildings and the state each of them are in.

Own Strategy:

- **Current Strategy Node:** Maintains the current strategy node for the AI player.

Goal Strategy Node: Describes the goal strategy node.

In-Game Own Knowledge: Contains the position and current status of all friendly units and buildings.

Building Plan: Contains the current building plan for the AI's base.

Unit Plan: Contains information about which units to build and in what order.

Research Plan: Contains information about which research upgrades to purchase and in what order.

Mission Knowledge: Contains information about different missions that should be executed in accordance with the current strategy. Each mission is noted along with the goal of the mission and the units assigned to perform it.

In-Game Map Knowledge:

Dynamic Map Knowledge: Includes dynamic elements such as resource locations and amounts. Will differ a lot depending on the game in question.

Dynamic Obstacles: Contains the position of all obstacles currently in view that are able to move from one game tick to another.

6.2.2 Data Representation

One of the most central aspects of a cognitive architecture is the way it represents knowledge. An architecture can choose to use a single, uniform encoding of knowledge, because of its simplicity and elegance and because it is easier to provide learning or reflection to only one type of data structure. The architecture can also provide a mixture of knowledge representations, because limiting the framework to only one type can in some cases force an awkward or inappropriate use of the framework. However, offering several different representations can bring unwanted complexity to the framework. Most frameworks therefore limit themselves to only a few different types. A common distinction between the choice of representation is whether it is declarative or procedural [LL02]. A *declarative representation* of knowledge allows manipulation by cognitive mechanisms independent of its content. *Procedural representations* on the other hand, represents knowledge as a way to accomplish some task. Another distinction is between *skill knowledge* and *conceptual knowledge*. While *skill knowledge* typically describes sequences of actions to achieve a certain goal, *conceptual knowledge* deals with objects and situations rather than the actions that manipulate them [LL02].

Many of the knowledge bases in our framework use trivial data structures and their underlying representation are not interesting, because the user will never be required to be aware of these representations. In the following we will emphasise the use of strategy trees and strategy tree nodes, as these are in many ways the foundation on which several modules work on. The use of strategy trees suggests a very organised hierarchy of knowledge, which means that knowledge pieces reference each other and have a relation to each other [LL02] [FKL05]. Strategy trees are furthermore a *declarative representation* and focuses on *conceptual knowledge* rather than *skill knowledge.* We hypothesise that strategy trees are a sufficient representation to represent all kind of strategies in all kinds of RTS games. This is not a data structure the user of the framework will be able to change, because this would complicate the internal methods in the framework working on strategy trees. Furthermore, because of strategy trees being able to represent all kinds of strategies, it provides the user with a relatively simple representation of strategies, which also gives access to learning methods as discussed in Section 5.4.1. First, all knowledge bases using strategy trees will be presented and their use of them will be explained:

Known Strategies: This knowledge base is a strategy tree containing all possible strategies available in the game in question.

Enemy Knowledge: This is a strategy tree containing all the strategies a certain opponent has done over several played games. Furthermore, it keeps track of how many times a certain strategy has been selected and thereby it will be possible to detect if the opponent has specific strategic or tactical tendencies.

Game Type Knowledge: This knowledge base contains several strategy trees depending on the number of different game types in a certain game. Each strategy tree contains different strategies and probabilities based on the game type.

Map Knowledge: This knowledge base includes strategies for each map in the game. Depending on the map, strategies and their likelihood of success will change, and a strategy tree for each map in the game represents this fact. The knowledge base makes it possible to learn map specific strategies.

Three in-game knowledge bases furthermore makes use of the strategy tree structure by using strategy tree nodes as their representation form:

Opponent Model: An opponent model consists of two things: A strategy node representing the opponent's current strategy node and a strategy tree path showing the steps the opponent went through to get to the current strategy node.

Current Strategy Node: A strategy node representing the AI player's own current strategy.

Target Strategy Node: A strategy node representing the AI player's goal strategy. This will always be a strategy node from one of the strategy trees in the other knowledge bases.

Two other knowledge bases deserve special mentioning at this point, as their representation are not obvious:

Base Building Templates: Base building templates follow the representation presented and discussed in Section 5.4.4.

Tactical Knowledge: The representation of tactics follow the representation presented and discussed in Section 5.4.3.

These two representations both focuses on a more *procedural representation* compared to strategy trees. They are concerned not only with representing a tactic or template, but also on how to execute the tactic or use the template. Having these two additional representations in the framework, means that the framework will have a total of three representations that the user should be aware of. This mixture of knowledge representations is acceptable in this framework, because each representation covers very different areas. At the same time, they can be used in connection with each other, as a strategy tree node can contain both tactics or base building templates used with the particular strategy.

6.3 Framework Versatility

When discussing a framework's versatility, one often talks of hot spots vs. frozen spots in the framework. Hot spots are the parts of the framework that differ from one instance of the framework to another, while frozen spots are the part of the framework that never changes from one instance to another [ML01]. We have decided to provide hot spots in two forms: Through configurable scripts for each module and through module extensions. By providing these two methods we both allow the framework to be used by the novice user and the advanced user. Novice users can change simple variables in scripts for each module that change the behaviour of the AI to the most common behaviours. Advanced users on the other hand, can extend entire modules or just single methods, to suit the needs of the particular game or AI in question.

Making easy configurable scripts to use for novice programmers is not an easy task. One must consider this from the beginning of the design of the framework, and make sure that the configurable variables allow for the necessary tuning, as well as being adaptable enough to suit all the different kinds of RTS games supported by the framework. Our framework will have scripts for all prior knowledge bases to allow for simple configuration of both the game in question, but also of what the AI should know before starting a game. Furthermore, each module will have its own configuration script, where module specific attributes can be set and easily configured depending on the wanted type of AI. All configuration scripts will be defined in a scripting language, and it will thereby require a minimum of programming knowledge to specify different variables. Furthermore, most of the scripts will be specified in a way such that not even knowledge of the scripting language will be required. However, even though not much programming knowledge is necessary, the user must still understand all variables that can be defined. It is for instance not possible to define strategies without understanding our representation of them through strategy trees. Easy script configuration is important for achieving the *shift of workload* design goal discussed in Section 4.1.

For the advanced users, the framework should allow for changing some of the internal details of the framework. In some cases, an RTS game can introduce an unusual feature that the AI should include in its consideration when deciding upon an action in a certain module. To support this, the framework will allow for extensions of all modules and nearly all methods in each module. The only methods not extendable will be the ones handling events, but these will as such not contain any other functionality than calling other extendable methods depending on the type of events. Only the event system itself cannot be extended, but it can, however, be configured through a configuration script where for instance priorities can be specified. Extensions of modules or methods in the framework must be written in C++, because of performance concerns. Having a module written in the scripting language will require far too much data transfer between the scripting language and C++, and this is too resource demanding to be used in a real-time system [PP04]. However, it should be possible for novice users to extend modules or methods in the scripting language in order to create prototype methods. This will enable AI designers to experiment with different methods, before having an experienced programmer implement the function in C++, because of the performance concerns. Performance issues will be further discussed in section 7.3.

6.4 Framework Control

This section will present the backbone of the framework. First it will be presented how the design techniques have been used, then the main system that controls all the modules will be described, and arguments are given to why it is built in this way.

6.4.1 Using the Design Techniques

While it has already been introduced how the RTS specific concepts are used in the design in Section 6.1 and Section 6.2, this section will focus on how to use event based systems and scripting languages.

Using events as a backbone for the activation and intercommunication between modules makes it possible to modify the architecture and even add more modules to the design, fairly easily, at a later point. Each module can send events to the event manager, which then activates the modules that have been assigned to handle that event. If later users would like to extend the framework with additional modules, they would just have to create the module and assign it to handle a certain event. If it is a new type of event, this can also be added and the system and the event trigger can be added, where the new module should be triggered.

Using the scripting language to configure each of the modules makes it unnecessary to recompile the code each time a small configuration is done, and makes it faster to tweak the framework for the variables that are configured there. Writing the prior knowledge bases in the scripting language furthermore makes it possible for a designer to configure these knowledge bases. This is good, because the prior knowledge bases are the part of the framework that have the most significant impact on the behaviour of the AI.

6.4.2 Event System

The main execution of the framework is controlled by an event manager. It must be possible to assign modules to be run when certain events happen. Each module must handle all the different types of events it can be sent, so the actual handling of each event is on each module. To simplify the event manager, it will just queue the modules that must be executed when an event is sent. Each module has a priority that the event manager uses to prioritise in what order the modules are executed. The event manager is accessible to any module at any time. After sending an event to the event manager, the event manager takes care of the rest.

The event manager must also be able to send events on its own at specified intervals. This should work like a timer. When assigned, it must be possible to specify at what interval the event type should be run.

6.4.3 Constructing the Architecture

Figure 6.3 shows what happens, when changing the influence in the framework architecture into events that activate each module at specific conditions.

Each module can be thought of as an object, and each of them can send events to the event manager. Further details about the event system can be found in Section 6.4.2. Each module is activated by sending events to it, with the exception of the *Game State Interface*, which is described in

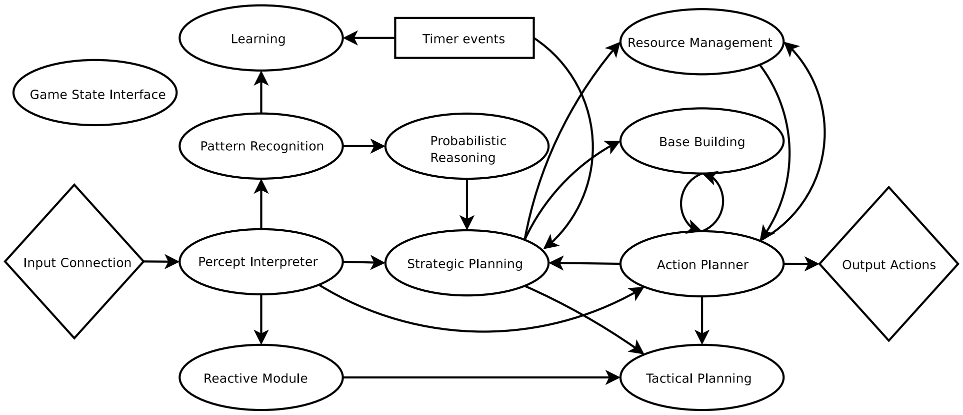


Figure 6.3: Event design

Section 6.4.4. The *Timer events* box represents the part of the event system that can send events with a certain frequency. The diamonds represent the interface to the GDF.

The entire framework is activated when an event is sent from the *Input Connection* to the *Percept Interpreter*. This event contains information about the changes that have happened since last *game tick*[2]. A short description about what events each module can send is given below. It only contains the modules that send events:

***Percept Interpreter*:** The *Percept Interpreter* notifies the *Pattern Recognition* module when new knowledge of the enemy is received. The *Strategic Planning* module is activated when the AI player has built new units, or one of the AI's units have been killed. Finally, the *Reactive module* module is activated if one of the AI player's own unit's hitpoints have changed. The *Action Planner* is also activated so that the units actions can be executed in the end of each time tick.

***Reactive Module*:** When a reaction from the *Tactical Planning* module is necessary an event is sent.

***Pattern Recognition*:** An event is sent to the *Probabilistic Reasoning* module when there are significant updates to the opponent model.

***Probabilistic Reasoning*:** An event is sent to the *Strategic Planning* module notifying that the *Probabilistic Reasoning* module has attempted

[2] A game tick is the discrete time steps that an RTS game is divided into.

to reason about the opponent's actions, and thus have new information available.

***Strategic Planning*:** If units have been assigned to either be used for gathering resources, constructing buildings, or scouting, the *Resource Manager*, *Base Building*, or *Tactical Planning* modules are sent an event accordingly. Furthermore, if there are any changes to *Mission Knowledge*, an event is sent to the *Tactical Planner*.

***Resource Manager*:** When more workers are required to gather resources, an event is sent to the *Action Planner* requesting this.

***Base Building*:** When the position of a building has been determined, the *Action Planner* is requested to give permission to construct the building.

***Action Planner*:** An event is sent to *Output Actions* each tick, telling what actions to do. When any unit or building action is complete, it sends an event notifying this to either *Strategic Planning*, *Resource Manager*, *Base Building*, or *Tactical Planning*.

***Timer Events*:** Events are sent to the *Learning* module at some interval to make it reason about the game being played regularly. The *Strategic Planning* module is sent an event to make it scout after a certain amount of time, and at some interval to make it evaluate the current situation of the AI situation. These intervals are specified by the user.

6.4.4 Game State Interface

This module is an almost direct connection to the game state, and should make it possible for developers using the framework to access information that has not already been put into knowledge bases. This is data that does not have any effect in how the AI should react, like static information about the map or time information. In new games there are always some new features or information that can be used in some way that this more general framework cannot comprehend. This interface gives a chance to get game dependent information into the framework. This module should be accessible any time and place in the framework.

The game state interface should give access to the following things:

Game Tick: It must be possible to get the timer counter, making it possible to know how much time has gone since the game started.

Tile Type: It must be possible to get the type of a specific tile, so it is possible to check if a unit can walk there.

Map Size: The map size is also necessary, especially for pathfinding.

Additional Game Specific Information: Any other game specific information that could be useful can always be added by the user in the user extension of the *Game State Interface*.

6.5 Summary

This chapter presented a cognitive framework architecture based on the human model presented in Chapter 3. We defined framework modules based on the tasks in the human model, and defined their exact responsibilities in the framework architecture. The complete design of all these modules have not been presented in this chapter, as it would be too extensive, and the reader is instead referred to Appendix B for the internal design of each framework module. In Section 6.2 we presented how knowledge in the framework is organised, and explained which knowledge bases make use of non-trivial data structures. We then proceeded to discuss framework versatility, and more specifically how AI developers can vary instances of the framework from each other. The final section, Section 6.4, focused on presenting the event system controlling the framework and also discussed how the framework is to be connected to the GDF used.

Part III

Proof of Concept

Chapter 7

Implementation

This chapter will describe the prototype implementation of the AI framework. We will start by describing the contents of our prototype implementation, and explain why each of the implemented features are important to prove the merits of the framework idea. Then we will specify certain implementation specific choices such as the game development framework used and chosen programming languages. A discussion of the trade-off between usability and performance is then presented. This discussion will focus on which parts of the framework can be specified by designers and which parts must be specified by programmers. Next we will present a new module, which connects the AI framework to the GDF. Finally we will introduce the reader to some of the problems encountered throughout the implementation.

7.1 Proof of Concept

We will through this implementation try to prove that the design goals presented in Section 5.1 will be fulfilled and that some of the key ideas in the framework are applicable for real use. This section will discuss which elements of the framework are necessary to create a running version and which elements are essential for proving the merits of the idea. First, the goals of the implementation will be outlined:

Reuse: The implementation serves to show how much a generic RTS AI framework can reuse.

AI Quality: Through the implementation, we assess how one can improve the quality of AI with an AI framework.

Developer Friendly: The implementation is also an experiment to see how developer friendly the framework can be made, and how much of the development of AI can be left in the hands of inexperienced programmers.

RTS Specific Concepts: Through the implementation, we will be able to test two of the RTS specific ideas presented in Section 5.4: Strategy trees and pathfinding.

Potential Problems: The implementation will also serve to identify potential problems in the design. The includes problems with connecting the AI framework to several different GDFs as well as identifying potential bottlenecks in the overall architecture of the framework. Furthermore, the implementation will give an idea of any performance problems.

We have decided to focus on two of the ideas presented in Section 5.4: Strategy Trees and Pathfinding. Both are extremely important elements of the framework, and essential for the framework to work properly. Strategy trees are the very foundation on which strategic decisions will be made, and are used by four framework modules: *Strategic Planning*, *Probabilistic Reasoning*, *Pattern Recognition*, and *Learning*. Furthermore, several of the in-game knowledge bases rely on the structure of the strategy tree node as well. Pathfinding is equally important as it is required to make units move, and hence three modules require its presence: *Strategic Planning*, *Tactical Planning*, *Resource Management* and *Base Building*. The *Tactics* representation presented in Section 5.4.3 will not be part of the implementation, as this idea is not yet fully developed. Furthermore, to really focus on tactics in an RTS game, a game with a complex unit composition would be required, including support units, and a unit system with armour types and attack types[1]. The test game, which is described in Section 7.2.2, does not support such a complex unit composition. The idea of base building templates presented in Section 5.4.4 will not be tested in the implementation either, because the selected test game only contains three types of buildings and placement of buildings is hence of minimal strategic importance.

The modular design of the framework allows for a clear distinction of the responsibility of each module, but it also means that a prototype implementation will be required to implement almost all modules for the framework to work in even a simple RTS game. The two modules that handle communication with the GDF, the *Percept Interpreter* and *Action Planner* modules, will be mandatory. In even simple RTS games, the AI must gather resources (*Resource Manager*), it must build a base (*Base Building*), it must control its army efficiently (*Tactical Planning*) and it must choose strategies and send the army to the right coordinates (*Strategic Planning*). For the *Strategic Planning* module to work properly however, it requires input from the *Probabilistic Reasoning* module, which in turn relies on a updated opponent model, ensured by the *Pattern Recognition* module. We can, however, limit the implementation of the *Pattern Recognition* module to just being able to

[1] Having different armour types for units allows different attack types to do more or less damage against a certain type of armour.

Module Name:	Implementation
Percept Interpreter	Complete implementation
Reactive Module	Complete implementation
Pattern Recognition	Updating of the opponent model
Learning	Not implemented
Probabilistic Reasoning	Complete implementation
Strategic Planning	Complete implementation, but simplified scouting and execution of strategies
Tactical Planning	Pathfinding, simplified micromanagement and situation assessment
Base Building	Simplified building placement
Resource Management	Resource gathering
Action Planner	Communicating actions to GDF

Figure 7.1: Implementation details

update the opponent model. This has the consequence that all attempts to recognise tactics or new strategies are not implemented and hence there is no information for the *Learning* module to work on. The *Learning* module would also be very complex in terms of developing methods for recognising tactics and base building templates and in terms of deciding how to control learning, so that the AI will not learn the wrong things. Furthermore, the only module that other modules do not rely on directly, is the *Learning* module. We have therefore decided to not include learning as part of the implementation. Table 7.1 presents how much of the different AI modules will be implemented. The simplified micromanagement of the *Tactical Planning* module means that we have implemented simple rules for focus fire and unit preserving, but nothing as advanced as discussed during the tactics representation in Section 5.4.3. For a complete reference of the design details that has been left out, the reader is referred to Appendix B. Besides the framework modules, the event system described in Section 6.4 will also be implemented.

7.2 Implementation Specific Choices

This section will discuss three implementation specific choices: The GDF used to test the AI framework, the game used to test the AI and the language selected to be the scripting language used to configure the framework.

7.2.1 Game Development Framework

There are only two possible choices to use as GDFs [FKL05]: ORTS [ORT05] and Stratagus [Str]. As mentioned in Section 5.1.2 there are problems in using both of them. ORTS lacks stability and documentation, while Stratagus requires changing the internal code of the GDF to allow the AI framework to handle low level AI actions [FKL05]. We have chosen to use ORTS as GDF for mainly two reasons:

- Changing the internal mechanisms of Stratagus is considered a far greater task than accepting the documentation of ORTS. This does, however, mean that a lot of time must be spend studying ORTS source code to make up for the limited documentation.
- Stability of ORTS is improving and because of an upcoming AI tournament [BASC05] organised by AIIDE [AII], where ORTS will be used as underlying platform, one can expect a certain level of stability. This tournament is open for all AI researchers and includes three different types of games: A resource gathering game, a tank combat game and a simplified version of a real RTS game.

7.2.2 Test Game

To test instances of the AI framework, one must select a game sufficiently complex to show important AI capabilities. The choice of ORTS as GDF however, limits the amount of choices available. As ORTS does not currently include a standard game, the choice is among three different games used for the AIIDE tournament [BASC05]. Only one of these is a real RTS game, which includes activities such as base building, resource management, opponent modelling, tactical planning, and strategic planning. The game is, a very simplified version of commercial RTS games, but it will be sufficient to test the AIs created with this prototype framework implementation. A short description of game details are listed in the following:

Game Type: The game is played as a 1on1 game between two AI players.

Unit Types: Three different unit types are included: Workers, marines and tanks.

Building Types: Three different building types are included: Control Centers, Barracks and Factories.

Resources: The game includes four different resource clusters randomly placed around the map. Furthermore, a resource cluster is placed close to both player's starting position.

Map: The map is a 64x64 tile randomly generated map with two terrain types: Ground and cliffs. Ground tiles are passable while cliff tiles are not.

Objective: The objective of the game is to destroy the opponent player's buildings.

More details on the game specification including technology tree, possible unit actions etc. can be found on the ORTS tournament page [Ort].

7.2.3 Scripting Language

The choice of a scripting language to use with the AI framework relies on several different factors. The main ones being:

- The language must be easy to use to support novice programmers and designers, while still being expressive enough to write complex behaviour in relatively few lines of code.
- The chosen language should make it possible to do rapid prototyping by allowing for adjustments of scripts without having to re-compile the entire framework.
- The language should be easily embedded into framework native code.

In our pre-master thesis [FKL05] we have analysed the following languages for their ability to handle the role as scripting language for the AI framework: Lua [Lua], Python [Pyta], Perl [Per], Tcl [Tcl], LISP [Lis] and Java [Jav].

Only two languages were able to fulfil our requirements: Lua and Python. Comparing the two, Python has direct support for objects and it is possible to pass entire objects from native C++ code to scripting code. This means that the programmer does not have to work with a stack or some converter tool only capable of passing simple types, which is the case with Lua. Because of this, we have chosen Python as being the best suited scripting language for the AI framework.

7.3 Trade-offs between Usability and Performance

In this section we will discuss what parts of the AI framework that should be created in the scripting language, and which should be implemented in the native language C++. In this discussion we will outline the advantages and disadvantages of creating the framework to be easy to configure. Afterwards a short description of how to configure the scripts in the framework will be given. Finally, it will be discussed how much actually can be created and modified with the scripting language.

7.3.1 Scripted Parts

The framework is designed so that it should be possible even for inexperienced programmers to use the framework if it has been connected to a specific GDF. The novice user should then be able to create very different AIs just by editing the scripts that configure the framework instances.

Being able to configure each module in the framework gives the advantage that there are some variables that can be modified depending on the connected game. In other cases it can be variables that change the behaviour of the modules, or thresholds identifying when and how the module should react. The only added resource use is at load time, which does not affect the performance of the framework. If functions that are run during the game are created in the scripting language, it has to be ensured that there is not too big a movement of data between the scripting language and the native language, because as stated by Phelps *et al.* [PP04], this part of integrating a scripting language or any other integrated language is the most resource demanding.

All of the prior knowledge bases are created in the scripting language. These are game and AI dependent, and are essentially the part of the framework that have the highest impact on AI behaviour. Some of the knowledge bases are only game specific. That means that if they have been configured to a specific game it is not necessary to change them, unless the game itself is changed, which often happens during the final balancing of the game.

7.3.2 Configuring the Framework

The framework is configured through scripts. The script files are saved in predefined folders, and in each of these script files a description can be found of what it does and what each variable configures in the framework. An example of this is the configuration of the *Pattern Recognition* module:

Listing 7.1: Script configuration of Pattern Recognition module

```
# This script defines variables for the Pattern Recognition module

### - Opponent Model - ###

# The following describes how often the AI will re-consider its
# strategy. Each unit or building will have two values defined for
# it. The first value indicates how much a certain attribute in the
# opponent model must change before the AI should re-consider its
# strategy. The second value describes how much a certain attribute's
# percentage part of the opponent model must change before the AI
# should re-consider its strategy.

worker = [10, 20]
marine = [10, 15]
tank = [8, 10]
controlCenter = [12, 22]
```

```
barracks = [7, 15]
factory = [4, 11]
```

As can be seen in Listing 7.1, it is a very simple syntax. Variables are simply assigned values. All the variables that can be set are listed in the script, and the user just have to fill in the numbers. This will further be discussed in the evaluation in Section 8.1.

To configure the *technology tree* that is used many places in the framework, there is a directory where each type of unit and building is defined in a separate file. The framework will dynamically load each of these files and add each of them as an element to the technology tree.

7.3.3 Scripting Limitations

There is no doubt that even though the computers of today have become much faster, there is still a need for optimising for performance, especially when dealing with games. That can be seen just by looking at the requirements of some newer games. The buyers still demand that the graphics become more and more realistic and that requires more and more processing time. Even if the AI is given more processing time, it will always be better to be able to take more things into account, so the better the performance of the AI code, the more things can be taken into account. But with the limited time for developing the AI as stated in Section 2.1.2, rapid prototyping is required, and as stated by Ousterhout [Ous98] and shown numerous time according to a lot of the cooperations using Python [Pytb], scripting is well suited for this purpose.

The central parts of the framework, which should make sure that the execution speed is high should however not be implemented in a scripting language. Even though scripts can be created to do the same things and some tweaking could make them faster, the execution speed that the scripting language can perform at, would simply not be enough.

7.4 Implementation Specific Details

This section will introduce a number of implementation specific details.

7.4.1 GDF Communication Architecture

The GDF communication architecture has an impact on how to integrate the AI into the GDF. The ORTS GDF uses a server/client architecture, where traditional games often use peer-to-peer communication[BF05]. This can be seen in Figure 7.2. The server/client architecture in ORTS makes it possible to hide information from each of the clients, so that it is not possible to cheat with full map knowledge[Bur02]. In the ORTS architecture each of the AIs in the game are connected to the server like any other client

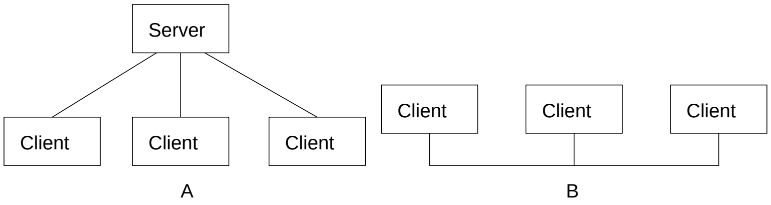


Figure 7.2: Server/client(A) and peer-to-peer(B) architecture

program[BF04a]. In the peer-to-peer architecture all AIs is run on each client to save bandwidth, but this takes up a lot more processing on each client.

In ORTS, each client sends its actions to the server every eighth of a second, and the server will then respond with the new updates in the game universe[UB06]. Because the AI has to send its actions to the server, and because the reaction time of the opponent is important, the framework has to be to some extend real-time.

7.4.2 GDF Connection

In the implementation we have decided to make the sub-module *Interface GDF*(described in Appendix B.10.6), from the *Action Planner*, into a separate module called *Connection* module. This will make a cleaner separation of the AI framework and the GDF. Furthermore the *Input Connection* and the *Output Actions* part described in Section 6.4, are also combined into this module.

The implementation of the *Connection* module that connects the AI framework with the GDF is one of the larger tasks that have to be implemented by experienced programmers. This module should get the percepts from the GDF, and input these into the AI framework. When the actions have been found by the AI framework, this module should then translate these into actions that can be understood by the GDF. The *Connection* interface must be implemented, and this contains two functions: *read*() and *write*().

When creating the *read*() function, the data that should be passed on to the *Percept Interpreter* must be extracted from the GDF. In the implementation that connects to the ORTS framework, this part reads, with the use of the client interface, the data that is transmitted from the ORTS server. The ORTS framework has an example of how to connect to the server, and this is used as a guideline for the implementation. The user must also implement the *Percept Interpreter*, which updates and maintains some of the in-game knowledge bases. The task of implementing this module requires some knowledge of how the knowledge bases are constructed.

The *write*() function gets a list of AI actions as input. It is then the responsibility of this function to take each of these actions and do what is equivalent in the GDF. Each type of AI action must be handled. Even though the AI framework should make sure that a certain action is possible to do, it might happen that something has not been taken into consideration, like trying to attack a unit that is already dead. Therefore, we have added a knowledge base where return values from the GDF can be stored. These return values are identified by the unit or building that should perform the action. The return values are then used in the AI framework to tell that the action was not possible, and it should try to find another action.

7.5 Implementation Problems

The following section will introduce the reader to the problems that were encountered during the development of the prototype of the framework. The problems concerning the GDF were mostly expected but the extend to which they affected the implementation was not. The first section will go through the expected problems with using the ORTS GDF and also describe how the unexpected side effects affected the implementation. The next section will describe some of the problems encountered when connecting the prototype to the ORTS GDF and finally the last section will present the current status of the prototype implementation.

7.5.1 GDF

This section will introduce some of the experiences we have made throughout the implementation concerning using ORTS as GDF. This section will mainly focus on the three problems identified in Section 5.1.2:

- Documentation
- Stability
- Full Control of AI

To summarise we discovered that ORTS lacked documentation and stability, but allowed full control over the AI. Stratagus that was the other candidate for GDF did have a limited documentation and because it has been used by developers for a couple of years, it is relatively well tested. Stratagus did, however, not allow full control of the AI. Through an analysis in the pre-master thesis [FKL05], we estimated that ORTS would be the GDF that was best suited.

Unfortunately the implementation has revealed a number of problems that we will now present and discuss. These problem have ultimately meant that precious time has been spent on tasks that could otherwise have been avoided.

Documentation

This section will emphasise the problems we have found that are related to the documentation, or rather the lack of this.

Presentation of the Environment: ORTS has until recently not included any information at all on the characteristics of the environment. Basic knowledge such as map size, map topology and cell movement cost have not been available to the users of the framework through any kind of documentation. The only way to gain any knowledge in this area is through hard study of the GDF source code. Even if any answer was found we could in most cases not be completely certain that this really was the right answer.

Reference Manual: The interface of all framework modules is available through a Doxygen documentation [dox]. This is as such an elegant solution for this type of problem, however, the documentation in the Doxygen has been added gradually through the development. This has the consequence that we had to guess how various modules worked and infer input on a number of functions through use of these in the sample AI, provided by default in the GDF.

Tutorials: The use of tutorials is a well-known technique to introduce new users to a framework. If tutorials had been available modules and connections could have been made more smoothly and the general learning curve for learning to use the framework could have been lowered considerably.

Limited Example Code: As already mentioned earlier, ORTS includes a sample AI in which many basic tasks are introduced. In order to be effective, however, this example code must be more extensive than it is the case. The sample AI essentially only moves randomly around and attacks an enemy when it gets within range. Areas such as mining, base building are not handled at all. Coupled with tutorials a broad scaled array of sample code can be a powerful tool, but a limited amount of sample code can raise as many questions as it answers.

7.5.2 Stability

The ORTS development deadline has been a concern from the very beginning of this project. Originally ORTS was meant to be in such a state that AIs could be integrated into it at the start of 2006. This deadline has been delayed a number of times till the interface definition was finally locked in late February. This has meant that the implementation of the framework in this project has been done in parallel to development on the ORTS GDF.

This has had the consequence that numerous times there have been compile errors in the ORTS code that had to be corrected before it could be used.

The GDF itself is not very developer friendly. The server will crash if it receives any invalid input. In general very few errors are handled.

7.5.3 Pathfinder

One of the key elements that have suffered from lack of documentation is the pathfinder. In order to even be able to start the implementation specific design of a pathfinder the developer must know things like map width, map height, which terrain types there are, movement cost of different terrain types, the map topology in general and how to extract unit and obstacle positions. All these factors and more had to be found through intensive study of source code of the GDF's internal modules. The development was not made easier by the fact that very little information exist about the "Blueprint" language in which game configurations are defined. This has had the consequence that it has been impossible to set up a "sandbox" environment in which to test and develop the pathfinder. Instead one of the predefined games had to be used. These games feature random generated maps that vary from game to game. This complicated development as data such as unit and obstacle positions cannot be transfered from test to test. Indeed the size of the game itself complicates matters. Instead of for instance working in a 32*32 cell environment in which a path can be easily verified, the games work in a 1024*1024 cell environment that severely complicates verification.

7.5.4 AI Framework

This section will summarise some of the problems that were encountered during the implementation of the AI framework.

There were mainly problems with the interaction with the ORTS GDF when having to make units perform certain actions, because this was not documented anywhere. The sample AI that came along with the GDF was the only thing that could give an idea about how to make units perform actions, and the only actions that was performed in this example code was the move action. Every other type of interaction with the GDF were more or less trial and error. Currently there are the following problems with the interaction with the ORTS GDF: Constructing buildings, harvesting resources, and attacking other units. The method of making a unit perform any of these actions is by activating the corresponding script function on the unit object. The problem is currently that when parsing the assumed parameters, the ORTS server crashes. The assumption about what parameters that is passed to the object is based on reading the script code. We have basically not been able to figure out what parameters should be passed to these functions, as there are no examples of how to do this, and it has not been possible to extract this information from the ORTS code. This has only left a trial and error approach, which has not been successful and hence these actions have not been implemented.

7.5.5 Implementation Status

This section is presented to give the reader an overview of the status of the prototype implementation, before we in Chapter 8 will evaluate it. At this point, all the features mentioned in Table 7.1 have been implemented. However, as discussed throughout this section, we have experienced numerous problems with interacting with the GDF. This has resulted in that we are not able to properly execute the actions we want. Furthermore, the JIT part of our pathfinding idea presented in Section 5.4.2 has not been included, because it has been an idea in continues development throughout the project and would require extra design consideration before being applicable in the implementation. This does not affect the overall test of the pathfinder. However, for an actual use in a framework the JIT technique will have a significant impact as the response time will be severely reduced [BMS04] and the computation will be distributed over several game ticks.

7.5.6 Conclusion

All in all we must conclude that ORTS is simply not yet mature enough for the intended purpose. In fact to our knowledge there are currently no RTS GDF that lives up to the requirements presented in Section 5.1.2. An AI framework such as the one built in this project would indeed have been easier to realise if ORTS for example had another couple of years to mature.

Chapter 8

Evaluation

In this chapter we will evaluate the implementation presented in Chapter 7. Throughout this discussion, we will identify strengths and weaknesses of the framework, and point out areas that require further research. The first part of this chapter will be divided into six sections, each covering an area important to be evaluated in order to determine the success of a possible full implementation of our framework. Combined, these areas will serve as success criteria for evaluating an AI framework for game development. The following shortly explains the focus of each section:

Configurable: The first area stems from the design goal in Section 4.1, which states that the framework must be able to shift the workload of AI development from programmers to AI designers. This section will primarily deal with subjects concerning the usability of the framework for inexperienced programmers. More specifically, we will evaluate how easy it is to configure knowledge bases and framework modules as well as discuss how to connect the AI framework to different GDFs.

Versatility: This area discuss an area of extreme importance for frameworks in general, namely its versatility [FS97]. We will in this section evaluate the different ways of varying instances of the framework from each other. This will be done by taking a closer look at how prior knowledge bases and module configuration can be specified to create different types of AI's for the different genres of RTS games defined in Section 2.4.2.

Extendibility: A third area which frameworks rely on is their extendibility [FS97]. As some games require special kinds of features, the AI framework should be relatively easily extended to deal with this. This section will describe how to extend the framework by example, and then discuss this method of adding new features to the framework.

Performance Testing: The fourth area will focus on performance testing and scalability, as this is important in real-time systems. This section will present performance tests on the prototype implementation, which will determine possible bottlenecks in the design. Furthermore, these tests will determine how much performance is used on the prototype implementation and discuss whether a fully implemented AI framework is a realistic possibility performance-wise.

AI Improvements: This area will focus on the AI improvements the framework is able to provide, which was also one of the design goals described in Section 4.1. This section will outline areas that the prototype implementation handles, which most commercial RTS games do not.

RTS Specific Concepts: The sixth and final area will focus on an evaluation of the RTS specific concepts presented in Section 5.4. This section will evaluate the two RTS specific ideas we have chosen to implement in the prototype of the framework: Strategy trees and pathfinding.

Following this, we will reflect upon the transition from design to implementation and discuss the development model used throughout this project. Finally, this chapter will end with a section that summarises important details discovered through the evaluation, and discusses potential problems and areas that require further work.

8.1 Configurability

This section will present a number of ways to vary framework instances from each other through configurable Python files. Appendix B presents the design of each module, including a specification of what can be configured on each of them. First, we will show how to specify units and buildings, and thereby build a technology tree, for the game in question. Then we will show how strategies are specified, and more specifically, how the user builds a strategy tree for the AI. Furthermore, as each module can be configured as well, we will pick an example module and show which and how different variables can be configured. Following each example on how to configure a certain part of the framework, we will briefly discuss advantages and disadvantages in this way of configuring the framework. Afterwards, we will shortly describe what the focus of AI developers should be, depending on the RTS genre the AI is being made for. Finally, the section will explain how the AI framework and the GDF are connected to each other and discuss problems in this approach, as well as potential solutions.

8.1.1 Configuration of Technology Tree

Four different things must normally be specified for a technology tree: Unit types, building types, research types and the dependencies between these.

For our prototype implementation, we can ignore research types as this is not a part of the game used with this implementation. A technology tree in the framework is specified by a unit or building specification including any preconditions there might be to this unit or building. For this prototype implementation there are a Python file for each unit or building specification. An example of this can be seen in Listing 8.1, which defines a marine type. The attributes defined for this unit are all specific to the test game used with the prototype implementation.

Listing 8.1: Specification of a marine type

```
name = "marine"
type = "Unit"
preconditions = ["barrack"]
hitpoints = 100
attack_max = 50
attack_min = 30
ground_attack_range = 8
movement_speed = 3
sight_range = 6
actions = ["move", "attack", "stop"]
minerals = 100
built_by = "barrack"
build_time = 100
supply_cost = 1
```

The code defines several standard attributes for a unit type. Furthermore, it defines the type of actions available for this unit and its place in the technology tree through the *preconditions* variable. The framework should attempt to include all standard attributes for units and buildings, such as the ones in Listing 8.1, but sometimes a game will require more than these. To add new attributes, a developer must do two things. First, the attribute must be added as an attribute of the particular unit or building type in the framework. This is not a difficult task, and can be done by simply copying how other attributes have already been created. Secondly, the developer must specify exactly how to use this new variable. If for instance the user have included an *armor type* attribute, which defines how much damage a unit takes from different kind of units, the *Tactical Planning* module should be modified to use this information in battle. Extensions such as these are discussed in further detail in Section 8.3. Depending on the new attribute added, the developer may want to extend several modules and methods to achieve the desired effect, and hence the complexity of such an extension varies. There is no way for inexperienced programmers to add such new attributes without the help of C++ programmers. Furthermore, the programmers implementing the feature must have detailed knowledge of the internal architecture of the framework. This must be provided through the framework documentation.

Documentation is a critical issue when building frameworks [FHLS97]. Several methods for providing good documentation with frameworks have been presented in literature and it has been identified as an essential factor in how well users are able to reuse software [BKM00]. One method is to divide software documentation into two broad categories: User documentation and internal documentation [Øst99]. Here, user documentation refers to reference documentation and introduction material such as tutorials, guidelines, cookbooks etc. Internal documentation on the other hand, refers to all kinds of documentation that a user may need to maintain and further develop the software. The framework described in this project should include both kind of documentation, as it must be used by both users of the framework and developers wanting to extend the framework.

As can be seen in Listing 8.1, although the code is written in Python, it does not really look like code. All units and buildings are specified in this manner, and the specification of all of these for the game used with our implementation can be found in Appendix J.1. Basically, everyone could easily create new unit or building types by just looking at previous examples of this. No programmers need to be involved in this process.

8.1.2 Configuration of Strategy Trees

Strategy trees are built in a way similar to the technology tree in the previous section. Each strategy node is defined separately with a unique name and information about parent strategies and counter strategies. The user must specify one strategy as the *starting_point* strategy, which is used as the root of the tree. In this particular game, the user starts with 6 workers and 1 control center. The code in listing 8.2 defines a fast tech strategy (explained in Appendix A.5) for this particular game.

Listing 8.2: Example of a strategy tree node defined for a fast tech strategy

```
fast_tech = {
"name" : "Fast␣tech",
"precondition" : "Starting␣Point",
"follow_up_strategies" : ["Mass␣tanks"],
"counters" : ["Fast␣expand"],
"percentage_use" : 30,
"time" : 500,
"purpose" : "step",
"expansions" : 0,
"controlCenter" : 1,
"barracks" : 1,
"factory" : 1,
"worker" : 8,
"marine" : 0,
"tank" : 5
}
```

The code defines all the attributes for strategy tree nodes discussed in Section 5.4.1. Furthermore, a strategy tree with all strategies can be built through the variables *precondition* (parents in the tree), *follow_up_strategies* (children in the tree) and *counters* (counter nodes). The *percentage_use* variable indicates how often this strategy should be used compared to other strategies at the same level in the tree. Finally, the *purpose* variable indicates what the AI should do when reaching this particular state. It can basically either be *step*, indicating that this strategy is only a stepping stone towards following strategies, or *attack*, indicating that the AI should attack at this point in the strategy. The entire definition of strategies and the strategy tree, the *Known Strategies* knowledge base, can be found in Appendix J.7.

It is possible to change or add attributes for strategy tree nodes, but C++ programming knowledge is required. For this brief example, assume that the developer wants to change the *purpose* attribute to contain a *defend* option. To do this, the developer must first define internally in the framework that this attribute can be a *defend* type. Afterwards, the framework must be told how to use this new type. In this case, it must be used when the AI reaches the state dictated by a strategy tree node having the *purpose* attribute set to *defend*. Checking the *purpose* attribute is already done within the *Strategic Planning* module in the *Evaluation* sub-module described in Appendix B.5.9, and the *defend* option can easily be added here. Defining what should be done when reaching this state, can be done in two ways: Either the developer uses functions already defined within the *Strategic Planning* module or she defines entirely new functions that dictate the behaviour of the *defend* type. Both ways require detailed knowledge of C++ and the internal parts of the framework. This must be obtained through the framework's documentation as also discussed in the previous section.

The code in Listing 8.2 is relatively easy for even non-programmers to write. Even though each strategy is actually a Python *dictionary*, the user does not need to be aware of this. With just a single example and an explanation of each attribute in the strategy node, an AI designer can easily define strategies for the game. The disadvantage of this approach is that the designer herself needs to keep track of strategy names, follow-up strategies, parents nodes etc. While this is manageable in simple games with a small strategy tree, it becomes very hard to keep track of when designing large and complex strategy trees. To overcome problems such as these, one possibility would be to let the designer, design strategy trees in a graphical user interface, where strategies and their relation to each other are more obvious.

8.1.3 Configuration of Framework Modules

For each framework module, there is a corresponding Python configuration file. In these, all game or AI specific variables can be set. As an example, consider a sub-part of the script used to configure the *Probabilistic Reasoning* module in Listing 8.3.

Listing 8.3: Configuration script for the Probabilistic Reasoning module

```
# This script defines variables for the Probabilistic Reasoning module

# The following describes the strategic importance of different attributes
# in a game. Values must be between 0 and 1, with 1 being the maximum
# strategic importance and 0 being no strategic importance at all.

controlCenter = 0.2
barracks = 0.2
factory = 0.2
worker = 0.5
marine = 1
tank = 1

# Maximum node deviation: The following variable describes how to compare
# two strategy tree nodes. It defines the percentage deviation that an
# attribute in the two nodes may deviate from each other and still be
# considered equal.

max_deviation_percentage_of_total = 10

# Determination of important variables. How much should variables in
# possible strategies deviate before being considered important to
# determine the final choice of strategy.

importance_buildings = 30
importance_units = 20
```

As with the scripts mentioned in the previous sections, the only thing the user needs to do is to define variables. To ease understanding of the scripts, each script includes comments on exactly what each variable means. However, it is still very hard for new users of the framework to understand some of the variables in the scripts without understanding the architecture of the module, which the script configures. The code in Listing 8.3 is for instance much easier to understand when the user has read the internal design of the module presented in Appendix B.4. The only way to solve this problem is through the documentation of the framework, which has already been discussed in Section 8.1.1.

A developer may want to add new variables to the module configuration script. As an example, consider that a developer wants to add a *aggressiveness* variable to the configuration script for the *Strategic Planning* module. This variable should define how aggressive the AI should be, and how willing it should be to choose an aggressive strategy. Two steps must be completed to add this new variable. First, the constructor of the *Strategic Planning* module must be modified to extract the new variable from the Python script. Listing 8.4 shows an extract of the constructor where this happens. This step includes creating a private *aggressiveness* variable on the *Strategic Planning* module class.

Listing 8.4: Extracting a variable from the Strategic Planning configuration script

```
Python_interpreter* py_interpreter = Python_interpreter::instance();
py_interpreter->run_file("ortsai/module_conf/strategicplanner.py");
object ns = py_interpreter->get_namespace();
aggressiveness = extract<int>(ns[("aggressiveness")]);
```

The second step will be to dictate where the variable is to be used. In this case, it will probably be in the sub-module of the *Strategic Planning* module, which deals with selecting a new strategy: The *Find New Strategy* sub-module described in Appendix B.5.7. Exactly how to extend methods in the framework is discussed further in Section 8.3. Both steps require C++ programming knowledge, and can hence not be done by AI designers alone.

8.1.4 Configuration of AIs in Different RTS Genres

This section will discuss how the framework can be configured to create AIs in the four different RTS genres presented in Section 2.4.2. We will present each genre in turn, and discuss what typically will be the focus when building AIs for that particular genre:

The Command & Conquer Genre: An AI in this genre should typically have less emphasis on the reasoning part of the framework, as counters only have little effect in these kind of games. The main focus is securing enough resources to mass units, while at the same time stopping the opponent from doing the same. The *Resource Management* and *Tactical Planning* modules are the primary modules for achieving this. The configuration of strategy trees should focus on the strategies in the tree rather than the counter nodes in the tree.

The Age of Empires Genre: This genre focuses on resource management with resources spread all around the map, and games tend to be a battle of control of these resources, hence making the *Resource Management* module essential. Furthermore, games in the genre often focuses on counters as well, making both the *Probabilistic Reasoning* and *Strategic Planning* modules very important to configure to perform the best possible way in the game in question. Strategy trees should be created with emphasis on both strategies and counters. Units have relatively few hitpoints, which makes micromanagement difficult, and hence the developer can put less work into configuring the *Tactical Planning* module. However, this module must still take care of things like formations, unit deployment, use of support etc., which are also important for AIs in this genre.

The Starcraft Genre: The key areas in this genre are areas like strategic variation, build orders and good placement of defensive structures. This effectively means that the *Strategic Planning* and *Base Building* modules are of great importance in these kind of games. Games in this genre will also benefit greatly from adding new states to the *States* sub-module of the *Strategic Planning* module presented in Appendix B.5.10. This is because there is so much emphasis on the execution of strategies in this genre.

The Warcraft Genre: This genre is characterised by having relatively high hitpoint units and buildings, which means that micromanagement and hence the *Tactical Planning* module has much more effect than in the other genres. The newer games of this genre also include a focus on counters, and because of this, both the configuration of strategy trees and the *Strategic Planning* module should be the focus of the developer.

Section 8.3.3 will discuss how the framework can be adapted to work with games that do not exactly follow the definition of RTS games presented in Section 1.1.

8.1.5 Configuration of Interaction with GDF

Configuring the *Connection* modules interface between the GDF and the AI framework is one of the larger tasks, and it is difficult to implement because it requires extensive knowledge of both the AI framework and the GDF. This evaluation will only consider connecting the framework with the ORTS GDF, as it has not been possible to test other GDFs, because of the problems with open source GDFs described in Section 5.1.2.

In Listing 8.5, the part of the *Connection* module that handles the reading from the ORTS GDF is shown. This excerpt checks for updates in the game state, as can be seen on *Line 4*. If anything is received, a container object is created on *Line 5*, a pointer to the game state is inserted on *Line 7*, and then on *Line 8*, a pointer to the changes that have happened in this game tick is inserted. Then the module will send an event telling that the game state has been updated, as seen on *Line 11* and *12*. This event contains a pointer to the container class where the data is stored, which the *Percept Interpreter* requires.

Listing 8.5: Read function from connection module

```
1 void Orts_connection::read () {
2   // looks for server messages
3   // if one or more arrived, send event
4   if ( gsm->recv_view() ) {
5         Game_change* data = new Game_change();
6
```

```
        data->game = game;
        data->changes = changes;

        if ( data->game ){
          AI_event* event = new AI_event ( AI_event::UPDATE_GAME_STATE, data );
          Event_manager::instance()->send_event( event );
        }
  }
}
```

Most of the complexity is not here in the read function, but is instead in the percept interpreter that transfers the data from the game state into framework knowledge bases. The *read*() function only makes sure that the data is accessible. Depending on the framework, this is a fairly straightforward task. In ORTS, the AI opponents act as separate clients, which means that if a client is already implemented, this code can be reused in the *read*() function for getting the necessary data.

In Listing 8.6, an excerpt of the user implemented *Percept Interpreter* is seen. This user implemented module is the largest and most time consuming task to implement for developers, the complex parts are hidden in the four functions called in this excerpt(*line 9-12*). It requires knowledge about the framework and the GDF, because it is here where all the data received from the GDF is translated into something that the framework can handle. Each of the knowledge bases that contain data that can change from game tick to game tick is updated here. However, when this rather large task of implementing has been done for the GDF, then it does not have to be changed anymore, unless new attributes are added to the game.

Listing 8.6: User implemented Percept Interpreter

```
1  void User_percept_interpreter::user_run(AI_event* event) {
2
3    Orts_connection::Game_change* data =
4               (Orts_connection::Game_change*)event->get_data();
5    this->current_game_state =  data->game;
6    this->changes = data->changes;
7    this->id = ((Game*)current_game_state)->get_client_player();
8
9    add_new_objects();
10   update_changed_objects();
11   remove_dead_objects();
12   remove_vanished_objects();
13 }
```

The function presented in Listing 8.6 takes the event that was sent from the *Connection* module, and extracts the data from it. The pointer it contains is first casted to the type that is in the event, as shown on *Line 3* and *4*. Then each of the pointers contained in this class is assigned to pointers in the local class. Afterwards four functions are called, each taking the data out of the newly assigned pointers and updating different knowledge bases. These functions are the complex part of the *Percept Interpreter* that translate the data from the GDF into knowledge in the knowledge bases

Listing 8.7 presents an excerpt of the *write*() function in the *Connection* module. This function is one of the largest tasks for the developers to implement, because it has to wrap all of the actions that are generated in the framework, into something that the GDF can understand. The *write*() function is given a list of actions, and all these actions should then be carried out in the game environment.

Listing 8.7: ORTS connection module

```
void Orts_connection::write ( AI_action::listtype* actions) {
  for(AI_action::listtype::const_iterator it=actions->begin();it!=actions->end();it++){
    GameObj* obj = id_obj_map->id_to_obj [ (* it)->id ];
    if ( obj && ! obj->is_dead() ){
      switch ( (* it)->type ) {
        // .... more cases
        case AI_action::ATTACK:
        // WARNING starting new scope
        {
          AttackAI_action* action = static_cast<AttackAI_action*>(* it);

          Vector<sint4> args;
          args.push_back ( action->enemy );
          rv->returnvalue[action->id] =
                obj->component("weapon")->set_action("attack",args);
        }
        break;
        // ... more cases
        default:
        cerr << "Invalid action give to Orts_connection" << endl;
      }
    }
  }
  delete actions;
  gsm->send_actions();
}
```

As can be seen in the code examples listed above, all the code is in C++, and this sets some boundaries for how developer friendly it is to implement. Furthermore, the programmer implementing this should have at least some understanding and knowledge of the structure and architecture of both frameworks, in order to translate data from one to the other in a reasonable way. The actions that are received as input to the *write* function are encapsulated in a data structure that the user also has to know, but this structure is very simple, and can be used as shown in Listing 8.7. This example is an excerpt that only handles the attack action, but all other actions should be handled as seen in this function. The list containing the AI actions can be iterated through like any other *Standard Template Library*[SL94] list. The most primitive action, from which all others are derived, contains a type enumerator, and an object id. The type enumerator can be used to assure the type of the class, and then a downward cast can be made safely, so the more specific data in the class can be accessed. As shown in the example, the *AttackAI_action* contains the ID of the unit that should be attacked. This ID is in the enemy variable, as seen on *Line 13.* With this information, the action can be performed in the GDF, like the attack action set on the object show on *Line 14* and *15.* Then on *Line 24*, the action list is deleted to clean up, and on *Line 25*, the GDF is asked to send the actions. In ORTS this means that the actions just assigned are sent to the server.

To improve usability for the *Connection* module a graphical user interface could be created to assist the creation of the module. This would most of all be similar to an *Integrated Development Environment*, because the only way to connect the two frameworks will be by creating a custom wrapper interface. However, the graphical user interface can only assist as an understanding aid of what has to be created, like having auto completion, checklists and descriptions of knowledge bases and their content. For instance for the *Percept Interpreter*, the developer could be aided by giving a checklist of the knowledge bases that have to be updated, also containing descriptions of the knowledge bases.

8.2 Versatility

This section will deal with two issues of versatility: Framework versatility and AI versatility. First, we will discuss the versatility of the framework by discussing if the prototype implementation has proven that it is possible to build an AI framework that is independent of the GDF to which it is connected. Secondly, we will discuss different ways of varying framework instances from each other and evaluate whether it is possible to create all kinds of AI for RTS games using the framework.

8.2.1 Framework Versatility

One of the goals of the framework was to make it independent of the game development framework. This would allow the AI framework to be connected to any GDF. The prototype implementation of the framework has only been connected to one GDF: ORTS. This means that the framework has as such not been tested in this area. However, one can make some general observations about the versatility of the framework based on the prototype implementation. The prototype implementation has successfully separated AI code and game development framework code by keeping all ORTS specific details out of all modules but two: The *Percept Interpreter* and the *Connection* module described in Section 7.4.2. All other framework modules are independent of the GDF. These two GDF specific modules must solve specific tasks, which the rest of the AI framework relies on. The *Connection* module must control the direct communication with the GDF as described in Section 7.4.2, and the *Percept Interpreter* must extract the information required for different knowledge bases, as defined in Appendix B.1.

The two modules, *Connection* and *Percept Interpreter*, contain 259 and 607 lines of C++ code respectively, in the implementation connecting the AI framework to the ORTS GDF and the code is not complex. Coding these modules requires the developer to have extensive knowledge of how the GDF operates and how the game state is accessed, to obtain the necessary information. However, as the programmers developing the AI are typically also involved in the game creation process, it is safe to assume that they also have knowledge of the GDF being used.

There are two requirements that a GDF must fulfil to be used with this AI framework:

1. The GDF must support giving full control of all AI actions to the AI framework.

2. The GDF must support retrieving the necessary data, described in Appendix B.1, for the AI framework to update in-game knowledge bases.

It is our understanding that the first requirement is fulfilled by most GDFs, but this cannot be studied, as most game development companies will not share information about their GDF. However, it would from a design perspective, not make any sense to have the two too closely linked. The second requirement should be fulfilled by most GDFs as well. The information required by the AI framework is essential for creating strong AIs and without a way to extract this from the GDF, it would not be possible to create the AI. It is unlikely that a GDF does not support operations required for creating AIs.

8.2.2 AI Versatility

There are two ways of varying framework instances from each other in the prototype implementation of the framework: Through the strategies that an AI knows, and through the configuration of different AI modules. This section will illustrate both ways of creating different kind of AIs and discuss whether this is sufficient to represent any kind of AI a designer may want to create. In the full implementation of the framework, the user would be able to also change AI behaviour through both tactics and base building templates, which were discussed in Section 5.4.3 and Section 5.4.4 respectively.

Strategy Trees

An AI built using this framework will never follow strategies not present in its strategy tree. This way, an AI designer has complete control over what the

AI will try to do during a game. If only one strategy is present in the strategy tree, the AI will only do this strategy. If only one counter is present for a certain strategy, the AI designer will know for sure that this is the strategy the AI will choose when faced with a certain strategy from the opponent. By letting strategy trees define the AI's strategic knowledge, we enable the AI designer to create AIs with very specialised behaviour. The above is however, only the case when there is no learning included in the framework. According to our definition of what a strategy should contain (Appendix A.1), strategy trees allow a designer to customise an AI to perform any strategy.

To illustrate how to use strategy trees to define different kind of AIs, consider Figure 8.1 and Figure 8.2. Both figures show a strategy tree for an AI in the game used for the prototype implementation. Figure 8.1 shows a strategy with three possible starting strategies, and a follow-up strategy for each starting strategy. For simplicity, no counter nodes are shown on this figure. Figure 8.2 shows a strategy tree for an AI containing only one of the starting strategies shown in Figure 8.1. The AI shown in Figure 8.1 will be able to choose between three strategies, and as seen on the edges, it must choose the upper branch 50% of the times. The AI shown in Figure 8.2 on the other hand, will always do the same strategy. By defining strategy trees this way, an AI designer can control the possibilities an AI will have during a game, and thus create exactly the kind of AI wanted for a particular game or situation. The addition of tactics and base building templates to strategy tree nodes in the complete implementation of the framework, will further increase the possibilities an AI designer will have to customise the AI.

Module Configuration

In terms of module configuration, each module typically provides three different types of configuration:

Game Specific Variables: These variables allow framework instances to be designed to suit a specific game. This is for instance definitions of which units should be considered workers, which buildings should be considered farms etc.

AI Balancing Variables: These variables are as such also game specific variables, but deals more specifically with the AI of a particular game. They help balance internal AI calculations by defining balancing variables. This could for instance be the variables in Listing 8.3 on *line 24* and *line 25*, which define the strategic importance of different types of units/buildings in the game.

AI Behaviour Variables: These variables define different behaviour attributes for an AI instance of the framework. Behaviour attributes are variables that define how an AI should reason and react to things

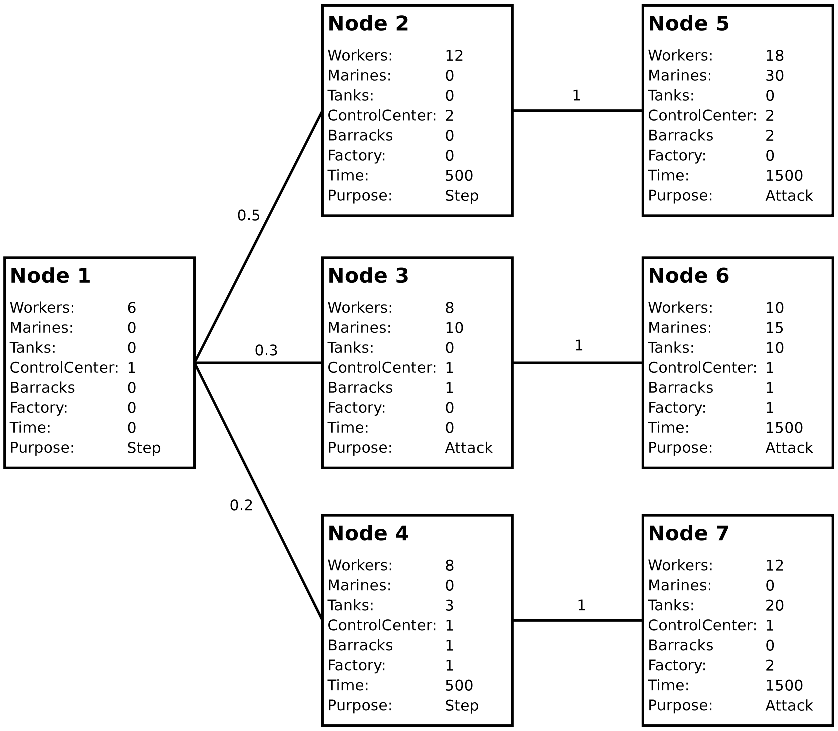


Figure 8.1: Strategy tree for an AI in the test game

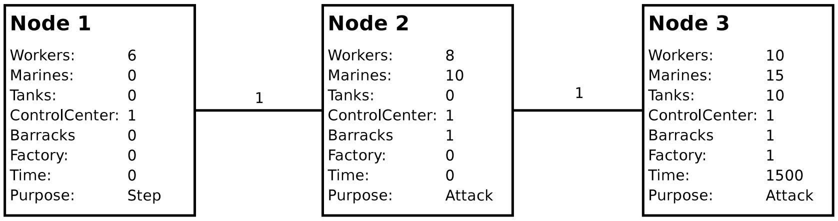


Figure 8.2: Specialised strategy tree for an AI in the test game

seen in the environment. This could for instance be the variable in Listing 8.3 on *line 18*, which defines when two strategy nodes should be considered the same. Depending on the variable, the AI will take more possible opponent strategies into consideration and change its behaviour accordingly.

Combined, these three types of variables allow for adapting the framework to any kind of game and any kind of AI. By having all game specific variables in scripts outside of the framework, the internal code of the framework is kept generic and independent of the game in question. Furthermore, the advantage of having these variables defined in scripts is that a user can change them without having to re-compile the entire framework. As discussed in Section 8.1.3, developers can add their own configuration variables to further customise the AI.

8.3 Extendibility

This section will give an example of how to extend a framework module, and then discuss and evaluate the method of doing this. Afterwards, we will give an example of how entirely new modules can be added to the framework and in the last section we will discuss different framework limitations.

8.3.1 Methods and Module Extensions

As explained in Section 6.3, the framework allows all modules to be extended. A user can choose to simply extend a single method in a module or to extend the entire module, including all its extendible methods. For this example we will only extend a single method, as this will be enough to provide the reader with the basic idea.

Imagine that an AI developer comes up with a new idea for how scouting should be carried out in a certain game. The basic scouting provided with the framework may have turned out to be insufficient for the game in question. In other words, the AI developer wants to replace the scouting method in the framework with a new method. This requires an extension of the *Strategic Planning* module, where scouting is handled. Consider a subset of the functions available in this module in Listing 8.8. These functions represent just some of the responsibilities of the *Strategic Planning* module which can be extended, and these are explained in detail in Appendix B.5. These extendible methods are often referred to as *hook methods* in framework literature [ML01].

Listing 8.8: Sub-set of the extendible functions in the Strategic Planning module

```
virtual bool sufficient_knowledge();
virtual void determine_scouting_mission();
virtual bool change_strategy();
virtual void find_counter_percentages();
virtual void find_new_strategy();
virtual void determine_expansions();
virtual void evaluate_situation();
virtual void execute_state();
```

In this case the user may want to change both when scouting is necessary (the *sufficient_knowledge()* function), and how scouting is actually performed (the *determine_scouting_mission()* function). Two things must be done to achieve this: Extending the module and informing the event manager to use this extended module. Extending the module is a relatively simple task, and is shown in Listing 8.9. The module will then use the new extended methods when these are present, and otherwise use the default methods the *Strategic Planning* module provides. The user does not need to worry about when to call the different functions, as this is handled internally in the framework.

Listing 8.9: Extension of the Strategic Planning module

```
class Extended_strategic_planner : public Strategic_Planner {
public:
        bool sufficient_knowledge() { ... };
        void determine_scouting_mission() { ... };
};
```

Following this, the event manager must be informed to use the new *Extended Strategic Planning* module instead. The place where the *Strategic Planning* module is assigned to the event manager, it must be changed to use the *Extended Strategic Planning* module instead. An excerpt is shown in Listing 8.10. Shown on *line 1*, the *Extended Strategic Planning* module is assigned to the `sp` variable instead of the normal *Strategic Planning* module. The pointer can still be of the derived class, because they have the same interface. The argument given to the constructor are the priority of the module along with the required knowledge bases. Then on *line 4* the module is given a meaningful name, and the module is assigned to the events it should handle as normal. *Line 1* and *line 4* are the only lines needed to be changed to inform the event manager to use the *Extended Strategic Planning* module.

Listing 8.10: Assignment of events to the Extended Strategic Planning module

```
1  Strategic_planner* sp = new Extended_strategic_planner ( 4, gtk, mk, ek, csn,
2    om, tsn, ks, kbo, ck, dmk, igek, mik, aua, igok );
3
4  sp->name = "Extended␣strategic␣planner";
5
6  event_mng->assign_module_to_event_type ( sp, AI_event::PRR_TRIGGER_SP );
7  event_mng->assign_module_to_event_type ( sp, AI_event::START_STRATEGY );
8  event_mng->assign_module_to_event_type ( sp, AI_event::AUA_NEWUNITS );
9  event_mng->assign_module_to_event_type ( sp, AI_event::AUA_DEADUNITS );
10 event_mng->assign_module_to_event_type ( sp, AI_event::SP_MOVE_END );
```

```
event_mng->assign_module_to_event_type ( sp, AI_event::RETREAT );
```

All framework modules can be extended this way. As explained in this section, extending modules and methods are relatively easy. One will need C++ programming knowledge to actually implement the extended methods, but the programmer can do this without knowledge of how other methods in the module works. We have identified two problems with this approach:

1. The user must know the architecture of the framework fairly well to know which methods to change to obtain a certain effect, and the user must likewise know which knowledge bases provide the different types of knowledge, and how to access them.

2. The framework restricts the user in the way the internal architecture of a module is designed. If the user wants to change the internal architecture of the module, she must basically implement the entire module from scratch, and personally make sure that all events are properly handled, and sent from the module.

These are typical problems when dealing with frameworks [FS97] [MBF99] and there are no way around them. Frameworks will include the architecture behind the solution, and the benefit of this increased code reuse is greater than the cost, as most AIs built using the framework will not require changing the internal architecture of any modules. Furthermore, it is unrealistic to expect to be able to extend methods in a module without understanding the basic architecture of that module.

8.3.2 Adding New Modules

Although this architecture is built to deal with all games included in the RTS genres defined in Section 2.4.2, some games may contain special features that users of the framework want to add a new module to handle. The following will describe how to do this, and what developers must take into consideration when modifying the framework in this way. The first step will be to create the new module, which is derived from the *Module* interface, with the appropriate knowledge bases, assign it a name, and then assign the event types that the module should handle. This is shown in Listing 8.11.

Listing 8.11: Creating a new module in the framework

```
New_module module = new New_module( /* Priority and Knowledge bases */ );
module->name = "New␣Module␣Name";
event_mng->assign_module_to_event_type ( module, AI_event::TYPE_1 );
event_mng->assign_module_to_event_type ( module, AI_event::TYPE_2 );
```

Afterwards, the actual module must be created. It must implement the module interface that includes a *run()* function, which must handle all the different event types this module can be sent. Listing 8.12 shows how this is done.

Listing 8.12: Mandatory run function in the new module

```
void module::run(AI_event* event)
{
    switch (event->type) {
        case AI_event::TYPE_1:
            // Handle event
            break;
        case AI_event::TYPE_2:
            // Handle event
            break;
        default:
            cerr << "Module␣failed␣to␣handle␣event" << endl;
        }
}
```

All that is left now is to add the functionality required to handle the different event types. However, adding a new module does nothing if it is never sent any events. This means that developers must also identify when events are to be sent, and add this to existing modules and methods in the framework. All modules that are to send events to the new module must be extended as explained in Section 8.3.1.

Now that the module is created and events are sent to it, the developer must consider how the module should affect other modules in the framework. There are basically two ways of doing this:

Through Knowledge Bases: By modifying shared in-game knowledge bases, the new module can change the foundation on which other modules work on, and through this, influence their behaviour. This must be done with great care, as in-game knowledge bases are often shared between several modules, and changing them may cause unexpected consequences. The developer must have extensive knowledge of the internal parts of the framework to make such modifications safely.

Through Events: The developer can also choose to create new events sent from the new module, which existing modules must handle. This is an approach that requires more work, but is safer as no unexpected side effects can occur. The new event types must be added to the event manager and each module receiving a new event type must be extended. However, the developer is left in more direct control of exactly how to handle different things from the new module. It still requires some knowledge of the internal framework, but less than influencing other modules through knowledge bases.

Which of the two methods the developer should choose, depends on the type of influence the new module should have on other modules and on the developers' understanding of the internal parts of the framework. No matter which method is chosen, adding new modules to the framework is the most difficult way to extend the framework. We hypothesise that if the framework is used for the intended games, the RTS genres defined in Section 2.4.2, developers will seldom find themselves in a situation where it is necessary to extend the framework this way. However, should special circumstances arise, it is possible to do, provided the developer understands the internal mechanisms of the framework.

8.3.3 Framework Limitations

The AI framework is built to handle the games falling into the category of RTS games explained in Section 1.1. However, many newer RTS games have introduced special features that make it deviate a little from traditional RTS games. The methods presented in this section have explained how developers can extend the framework in various ways to cope with these new requirements.

As an example, consider The Lord of the Rings: Battle for Middle-Earth which is widely considered an RTS game. However, this game differs from traditional RTS games by the way it handles resource gathering. Instead of having workers running to and from resources spread around the map, this game relies on one universal resource that the player acquires by building farms and slaughterhouses in predefined positions on the map. In fact, all buildings must be built at predefined locations. These two features have significant impact on how an AI should play the game. All internal operations within the *Base Building* and *Resource Management* modules in the AI framework would basically be useless and unable to cope with these kind of changes. For the AI framework to be useful in this game, both of these modules must be extended and completely re-implemented to suit the specific demands of the game. At this point, developers must seriously consider whether the benefits provided in other AI areas by the framework is enough to justify modifying the framework to such an extend.

In general, when several framework modules must be completely changed, developers must consider the trade-offs between the benefits of using the framework compared to the learning curve required to be able to modify the framework. If basic structures such as strategy trees and tactics cannot be used in the game in question, it is probably not an advantage to use the framework. The benefits of using frameworks in general, disappear when users have to change too much of the internal architecture.

8.4 Performance Testing

This section will show how the performance of the framework is measured, to test if the AI framework meets the real-time performance constraint, presented in Section 7.4.1. First a description of how the tests are created will be presented, and following this, the result of the tests. Then a discussion of the results will be given, and finally a discussion of how the framework scales in a complete implementation is presented.

8.4.1 Performance Test Construction

To test the performance of the framework and framework modules, code is added to the event manager, which tells how much time is spent in each module, and after a complete game tick. The numbers that the time test can give will represent; The actual time that is spent in the module, the time the operating system has spent on behalf of the application, and the complete time that is spent. Because the operating system will change between processes while the program is executed, the most realistic result will be the complete time spent, because the operating system will always do this when running a game. Only the most necessary programs will be run on the machine while the test is performed to minimise the factor of other programs taking processing time. This means that only the ORTS server and two instances of the prototype are run.

The following four tests will be performed:

Game Tick Performance: After each game tick the time used is recorded. This will show if the framework is fast enough to be executed the number of times each second that is required by the GDF. In ORTS this is eight times per second, which means that the framework will have 0.125 seconds to execute each game tick. When the framework is not performing any actions, this should be considerably less, in the area of 0.02 seconds. No processing time should be used, because no decisions or actions are made. If the framework uses too much processing time, there would not be enough processing time for the actual game. A graph can be drawn comparing processing time over game ticks. This will show if there is an increase in processing further into the game.

Module Performance: This test will show which modules use the most processing time. After each module is run, the time passed is recorded. An average of each module is then made. This is presented in a list, showing each of the modules and their average use of processing time per game tick. This test can also be used in the actual use of the framework, as an indicator of what modules could be optimised to get better performance.

Module Game Tick Performance: This is a combination of the two previous tests, where each module's processing time over game ticks is plotted in a graph. This shows what modules are used in different parts of the game. Some modules are used more in the start of the game, while others are used more during battle. This test can give an idea whether a module like for example the *Tactical Planning* module is fast enough, when the AI comes into a large battle against an enemy. There are no time requirements to the specific modules, just as long as the total time of all modules in a game tick is less then 0.125 seconds.

Pathfinding Percentage: Pathfinding is the most time consuming part of an AI, so a test is performed to see how much of the total time in the AI that is spent in the pathfinder. This is done by recording how much time is spent inside the pathfinder and comparing this with the total AI framework processing time. This is done over several game ticks, so it is possible to get a meaningful and general result.

The test is performed on an Intel®Pentium®III Mobile 800MHz, running Linux 2.6.16 and the code for the framework is compiled with g++ 4.0.3.

8.4.2 Performance Test Results

The result of each of the four tests can be seen in the this section. The tests are performed with the use of a timer that can tell how many hundredth of a second that have passed since the program was started. The time spend in a function is then found by subtracting the time before and after this function is called. This does however mean that the precision of the measurement is limited to a hundredth of a second. The values in the graphs and tables below are therefore all measured in hundredth of a second that functions use of processing time.

During testing it was discovered that the pathfinder implementation had performance problems. This is most likely because of the implementation of the algorithm, so in the first three performance tests, a simplified version of the pathfinder is used. It is simplified in the way that it moves on larger cells instead of on unit coordinates, and does not take obstacles into account. This means that it is possible for units to walk into each other and get stuck. This simplification was necessary to be fast enough to respond the ORTS server within a reasonable time, otherwise it would not perform any actions.

Game Tick Performance Test

Figure 8.4.2 presents the first 77 game ticks. It can be seen that there is used a lot of processing time in the first game tick. This makes sense because this is where a lot of the data that is sent from the ORTS server is inserted into the knowledge bases. Furthermore a lot of decisions are made, for instance what strategy to follow. The second peak at the graph is at the third game tick. This is the first time that the pathfinder is used, because the workers are assigned to gather resources, and a path is found for each of them. After this nothing happens until the scouts have to find a path, at game tick 31.

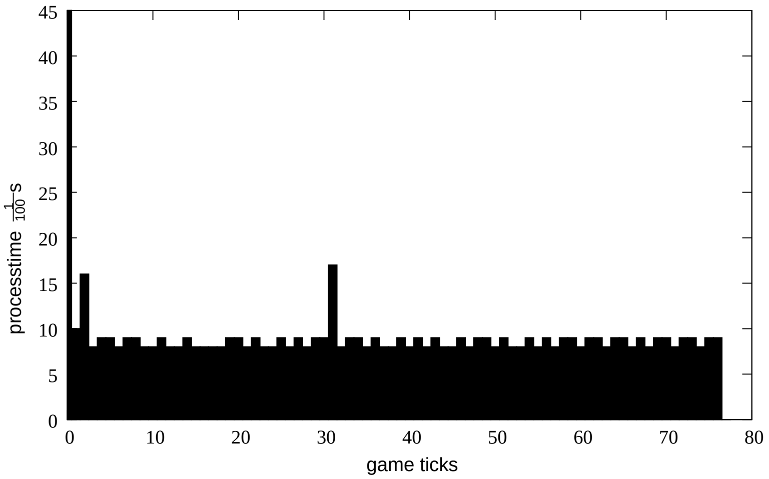


Figure 8.3: Game tick performance test

In general it is observed that the processing of each game tick takes around 0.07 or 0.08 seconds, when nothing of importance is happening. Considering that there does not happen anything, and the framework still uses 0.07 or 0.08 seconds there is something that has to be optimised. The framework has to be a lot faster to be usable in a real game.

Module Performance Test

Table 8.4.2 presents the result of the individual module performance test, and are the average of 20 games run for 960 game ticks, meaning two minutes. There is an uncertainty of this test, because of the already mentioned millisecond limitation. This part is even worse here, because after each module is run its millisecond count is recorded, and the next time this module is run, the count from this is just added to the first value. This means that if a module runs for less then a millisecond at each run then it will never be recorded as being run. This was the case for some of the modules. However, in average the time was barely one, so this is the number recorded.

This table shows that too much time is used in the *Tactical Planner*, and that there have to used some time to optimise what is being done in this module. To get more detailed information about what takes all the processing time in this module can be done by using a profiler like the one used in this project, the *GNU gprof* profiler[gpr]. Here it was seen that it was the pathfinder that used all the processing time, even though the pathfinder has been simplified. This might mean that the pathfinder is called too many times, and should be distributed out over some more game ticks. The *Percept Interpreter* uses 0.53 seconds, which is reasonable enough, considering it is

Module name:	Processing time:	%
Action Planner	51	1.00%
Base Building	1	0.02%
GDF Connection	18	0.35%
Percept Interpreter	53	1.04%
Probabilistic Reasoning	1	0.02%
Reactive Module	1	0.02%
Resource Manager	7	0.14%
Strategic Planner	1	0.02%
Tactical Planner	4944	97.38%

Figure 8.4: Module performance test

over two minutes, because it takes all the percepts from the GDF and puts it into knowledge bases. The *Action Planner* uses almost as much as the *Percept Interpreter*, which would make sense because it is also run at every game tick, collecting all the actions and sending them along to the *GDF Connection* module.

Module game tick performance

An extract of a complete performance test log can be found in Appendix H.1. This has not been plotted to a graph, because it is too difficult to illustrate. In the log some of the first game ticks are presented. Here it is possible to see what was also identified earlier, that the *Percept Interpreter* uses a lot of processing time to get all the information first given from the ORTS server. The second game tick is fairly standard, the only module that uses processing time is in *Tactical Planning* because it is using the pathfinder. In the third game tick we can see that the *Resource Manager* is more active. This is because this is where it identifies what resources it should go and harvest. After this the only module that uses any processing time is the *Tactical Planning*. Then in game tick 32 something is happening again. This is where scouting is started, so the *Strategic Planning* makes the *Tactical Planning* send out a scout. This makes the *Tactical Planning* uses a bit more processing time.

It is identified through this test that there is something that has to be done in the *Tactical Planning*, because it always uses a lot of processing time, even when it is not supposed too.

Pathfinding Performance Test

As already mentioned it was discovered that the pathfinder was not fast enough to be used in the first three performance tests, which meant that a simplified version was used. In this test the complete implementation of the pathfinder will be tested. We know that it is not fast enough, so the performance of the pathfinder compared to the rest of the framework will not be considered, as the results at this point would be meaningless.

In the test of the pathfinder, a unit is made to find a path to a position that are five clusters away, meaning 80 cells. This sort of movement should be no problem for a pathfinder. The calculation of the passable path takes less than 0.01 seconds, and the calculation of the path takes 0.96 seconds. Considering that there are only 0.125 seconds of processing time available at each game tick, this is not fast enough. If the JIT functionality is implemented the calculation of the path will be distributed over more game ticks. This would mean that there would no longer be such significant peaks as seen in Figure 8.4.2.

8.4.3 Performance Test Discussion

The performance test is not representative for what would happen in a complete AI framework, but it gives an idea if it is possible to create a framework that is fast enough to meet the time requirements. The modules and features that have been completed, which are presented in Section 7.1, have been tested to perform within the time constraints, with the exception of the pathfinder and the *Tactical Planning* module. Because the pathfinder is too slow to be executed in one game tick it has to be optimised, it has to be possible get at least a small path within one game tick. Furthermore to minimise the use of the pathfinder, it could be made to pathfind for groups of units, instead of doing it for each individual unit. The *Tactical Planning* module has to be optimised so that it does not use so much processing time at every game tick.

The event manager that controls all the modules have not been tested for performance, because the number of events that are sent within a game tick will never be large, so a stress test of how many events it can handle would be meaningless. To distribute the execution of modules even more, the event manager could be modified so that it does not only consider what game tick it is in, but also how much time there have been used. Then it takes its current module execution list and save it for the next game tick, and sends the actions that have already been found. This distribution of the processing time will make sure that more processing time can be used in each module, but the AI would be slower to react.

The test setups that have been presented here could be used, along with the complete AI framework, to identify some of the same problems that have been identified in these tests. A GUI can be created to automatically create these tests. This would then present the graphs and tables that would help the developers with the identification of bottlenecks.

8.4.4 Scalability

Scalability is important if the framework is to be used in a real game. In a real game there are a lot more units, tactics, and strategies than in the simplified game used for the performance test of this prototype. The question is whether the framework in a complete implementation can cope with the complexity of a real game, and if the framework data representation is efficient enough to be useful. In a real game there can of course be a lot more units than has been the case in this test scenario. This is in most cases not a problem, because the only parts that should be dependent of how many units there are in the framework is where units are deleted when they are dead and where they are added when they are constructed. The only place, where the number of units is an issue, is pathfinding. This could be handled by, instead of pathfinding for each individual unit, pathfinding for a group at a time. The event system will not have any issues with the increased complexity. This will only start to be a problem if more modules are added, because the number of events sent around in the framework, is not dependent on the number of units, only on the number of instructions that each module have to inform each other about. This can of course be dependent on the number of units that have to perform actions at a given time, but this could be handled with some optimisations. When dealing with performance the data representation is very important, because this is often what can tell if an application has potential to scale to a larger solution [CLR90]. The framework uses three types of non-trivial data structures as presented in Section 5.4: Strategy trees, tactics representation, and base building templates.

The strategy trees have no problem with scalability, because even if the tree becomes very large, it is for the most times only the nodes that are in the closest relation (meaning its parents and children) with the node that is currently being worked with that is considered. Each of the strategy tree nodes contains references to the children and the parent, which makes it possible to get the nodes in the closest relation in constant time. When searching for what strategy the enemy is using, it is no longer possible to access the data in constant time, as the tree has to be traversed from the root node. This is most likely not that big of a performance problem, because the worst case scenario is that it has to search from the root to the top of the tree. This will only be a problem if the degree of the internal nodes of the tree are low and the tree thus deeper than wide. The worst case scenario have complexity $O(n)$ and best case is $O(log(n))$, depending on how the tree is defined.

The tactics that are relevant for a certain strategy are listed in each strategy tree node, in a limited sized list. This means that even if there exist a large amount of different tactics in a game, there are limits to how many are accessible at one time, because only the tactics in the strategy tree node are considered. This way the complexity is in the hands of the designer, when she choose the amount of tactics in a certain strategy.

The base building templates are organised in a tree as explained in Section 5.4.4, and in the strategy tree nodes there are references to what base building templates that fits the strategy best. This means that the base building template tree nodes are accessible in constant time, even if the base building template tree grows extremely large. This is because there are direct links to the nodes used, and because when the tree is used it is only the closest relatives that are accessed.

Based on the above reasoning, we hypothesise that there should not be to many problems with the scalability of the framework. Even if the framework knowledge bases become huge, which would be the case for a real game, the representation of knowledge in trees is effective because it is only the relatives that have to be accessed. Furthermore, the only cases where any of the trees have to be searched is when looking for the enemy's strategy, and this search is not performed on each game tick. The issue of pathfinding can be solved with a change. When moving large number of units, the units will move in a group and only one pathfinding will be done for the entire group.

8.5 AI Improvements

This section will discuss how the framework improves the quality of AI in RTS games. We will use the test model described in Appendix D to test the prototype implementation. This model was also used to test existing AIs in games. The results can be seen in Appendix E. The test results are shown in Appendix F, which also shows the previous test on AIs for comparison, as well as what we hypothesise that a complete implementation will be able to handle. Areas only partially marked in the test table indicate features that are present in the prototype of the AI framework, but have not yet been testing with a GDF. Note that we will not consider the *Cooperation* tests with the complete implementation as this is not part of the design presented in this report. We will first present how the prototype implementation handles the different tasks marked in the test table, and afterwards discuss how a complete implementation will be able to handle the remaining areas.

8.5.1 Prototype Implementation

The prototype was created to be able to handle 11 of the test scenarios in the test model. However, because of the implementation problems discussed in Section 7.5, it has only been possible to successfully test six of these areas. To overcome implementation problems, the test game was simplified in a few areas compared to the game described in Section 7.2.2. We have removed cliffs, dynamic obstacles and manually placed different units on the map to better test different scenarios. Compared to the test we made on

Strategy Chosen:	Number of times	Percentage
Fast Tech	7	23.33%
Fast Expand	8	26.66%
Marines	15	50%

Figure 8.5: Chosen start strategies

the commercial RTS games, being able to handle 11 of the test scenarios is a good result considering this is just a prototype of the framework. The best AI among the commercial RTS games was able to handle 17 scenarios. This is, however, including all the *Cooperation* areas, which we have not included in the design of our framework. We will begin by demonstrating how the six areas fully marked are able to handle their corresponding test scenario. These areas will among other things, demonstrate how strategy trees allow for strategic variation and counters during a game. Furthermore, it will show how relative simple scouting can make an important difference for AIs in RTS games. Throughout the test, the strategy tree used by the AI is the one shown in Appendix K.3. For illustration purposes, counter nodes are not depicted, but just noted as an attribute of each strategy tree node. The six areas and test results are discussed below:

Using Counters: The prototype implementation handles counters by utilising two framework modules and strategy trees. The strategy trees for the test game define counters to each strategy the AI knows. During a game, the *Probabilistic Reasoning* module will attempt to discover which strategy the opponent is using, by reasoning about the *Opponent Model*. Because of the *Probabilistic Reasoning* module, the *Strategic Planning* module is aware of which strategy the opponent is most likely using, and can then look it up in the strategy tree and find its direct counter strategy. Given the strategy tree, everything else is handled internally in the framework.

This area was tested through the game logs shown in Appendix G. These logs are extracts of complete logs with only necessary information included. As an example, the log in Listing G.1 shows how the AI sees the opponent in game tick 61. At this point, it cannot see anything else than what the AI started with, and the only potential strategy that matches what it sees, is the *Fast Expand* strategy which is the only strategy it knows only consisting of worker units. As the counter to the *Fast Expand* strategy is the *Marines* strategy, this is the strategy chosen. At game tick 89, additional information is discovered. The list of potential strategies shows how much the current opponent model differs from each of the potential strategies' corresponding strategy tree node. Here, the *Mixed* strategy seems most likely and it is therefore the *Mass Tanks* strategy that has the greatest chance of countering the enemy. Note that in these examples, we have out-commented the code that determines how much a counter may deviate from the current state of the AI before being applicable. This means that the AI would not necessarily follow the proposed counter strategy. However, these examples demonstrates how the AI is capable of recognising the enemy's strategy and selecting the right counter for it, based on knowledge from the strategy tree.

Strategic Variation in one Game: As a direct consequence of the AI being able to use counters, it is also capable of changing its strategy during a game.

Strategic Variation Game to Game: Because of strategy trees' ability to represent several options at any given state in the game, the AI will choose its strategy based on probabilities given for each possible strategy. Given the strategy tree used for the AI, the AI should choose a *Marines* strategy 50% of the time, a *Fast Tech* strategy 30% and a *Fast Expand* strategy 20% of the time.

This area was tested by letting the AI play the same map 30 times in a row, and then observing which strategies it decided to use. The results can be seen in Table 8.5.1. The results show the *Marines* strategy being picked 50% percent of the time as expected, and the other two around 25%. This shows how the AI varies it strategic choice from game to game.

Does It Scout At All: A timer ensures that the *Strategic Planning* module sends a scout in the beginning of the game and then afterwards with regular intervals. In the full implementation, the *Strategic Planning* module should base its decision to scout on whether it had sufficient information about the enemy.

Scouting is demonstrated in all game logs in Chapter G. In game tick 31, the timer ensures that a scout is sent, and when the scout finds the enemy base, this is reflected in the opponent model.

Using The Acquired Information: The *Probabilistic Reasoning* module uses the information obtained from scouting, which is in the *Opponent Model*, to reason about the opponent's choice of strategy.

As already discussed in regards to the AI's ability to counter the opponent's strategy, the AI uses information gained from scouting to determine the opponent's potential strategies.

Sensible Unit Used for Scouting: In the prototype implementation, the user dictates which units to use for scouting and the *Strategic Planning* module chooses an appropriate unit of this type to scout.

As seen in game tick 31 in all game logs in Chapter G, the AI always chooses a worker to scout. This is dictated by the module configuration script for the *Strategic Planning* module.

One important conclusion can be drawn from the AI improvements the prototype implementation of the framework provides. The internal framework representation of strategies, strategy trees, enable AI designers to easily create AIs that both counter and use information about the enemy to predict her strategy. There is only one example of an AI in a commercial RTS game being able to do this, which is *Age of Mythology*.

Five other areas have also been implemented in the prototype implementation, but as explained earlier, we have not been able to test them because of the problems discussed in Section 7.5. The five areas are listed below, along with an explanation of how they are handled in the design of the framework and prototype specific details.

Measure Own Str. vs Enemy Str.: The AI will compare its strength to the enemy in two sub-modules: The *Evaluation* sub-module in the *Strategic Planning* module described in Appendix B.5.9 and in the *Evaluation* sub-module in the *Tactical Planning* module described in Appendix B.6.3. In the *Strategic Planning* module the evaluation decides if the AI should engage the enemy or not, and in the *Tactical Planning* module the evaluation decides if the AI should retreat from a battle.

Saving Hurt Units: The sub-module *Unit Deployment* in the *Tactical Planning* module described in Appendix B.6.7, takes care of saving hurt units. As there is no healing in the game, units are simply withdrawn from the front line, and then returned to battle. In the prototype implementation, the functionality is explicitly defined into the sub-module, but in the complete implementation, this functionality should stem from the tactics the AI designers have designed before the game.

Focus Fire: The sub-module *Targeter* in the *Tactical Planning* module described in Appendix B.6.9, take care of focus firing. This functionality is also explicitly defined in the module, and should be replaced by tactics defined by the AI designer in the complete implementation.

Spending Available Resources: The sub-module *Unit Planner* in the *Action Planning* module described in Appendix B.10.3 will make sure that resources are constantly spent. If the goal strategy node is already reached in terms of the number of units wanted, the module will simply keep producing the units in the goal strategy node, while maintaining the percentage unit distribution of the node.

Scouting Enemy: After the first scouting, where the AI finds the location of the enemy base, the *Strategic Planning* module will make sure that the regular scouts will always scout the enemy base.

8.5.2 Complete Implementation

This section will discuss why we hypothesise that the complete implementation of the framework can handle all the areas marked in the table in Appendix F. The design of the framework has been focused on being able to solve all of the tests in the test model. The focus of the test model is to test different areas of the human model, and this evaluation thus assumes that the human model is correct and the fulfilment of the human model is the goal for the AI. We will go through each of the areas not already handled by the prototype implementation, as these are also handled in the complete implementation. For each we will present how we expect the area to be handled, and refer to the part of the design that handles that particular area.

Exploiting Weak Spots: The *States* sub-module of the *Strategic Planning* module described in B.5.10 handles this area. The user will be required to specify what defines strong and weak points.

Reasonable Expansions: Because of strategy trees, the AI will always expand at the right times, that is, at the time where the strategy dictates it. Furthermore, the *Expands* sub-module of the *Strategic Planning* module described in Appendix B.5.8, ensures that the expansion is placed at a sensible spot.

Using Map: The tactics representation discussed in Section 5.4.3, includes rules for how to use the map terrain to the AI's advantage. Furthermore, the *Evaluation* sub-module in the *Tactical Planning* module discussed in Appendix B.6.3 will react upon a potentially bad battle position and act accordingly.

Good Buildorder: The prior knowledge base *Known Build Orders* should contain optimal build orders for achieving certain strategies in the fastest possible way. Using these, will enable the AI to successfully handle this test.

Using Formations: The tactics representation presented in Section 5.4.3 includes rules for formations, and the sub-module *Formations* in the *Tactical Planning* module described in Appendix B.6.5 will use these to deploy formations when moving during the game.

Map Considered When Moving: Handling this, is primarily a task for the *Path Planner* sub-module in the *Tactical Planning* module described in Appendix B.6.10. This module will use the pathfinding method presented in Section 5.4.2.

Using Tactical manoeuvres: The tactics representation discussed in Section 5.4.3 will define rules for making tactical manoeuvres. The *Tactical Planning* module must use these rules to actually execute the tactic in battle.

Staying in Control of Units: This test primarily deals with the native AI on each unit. To handle this area, we have designed a *Handle Native AI Event* sub-module specifically suited for this in the *Reactive Module*. This sub-module is described in further detail in Appendix B.2.6.

Counter Focus: By using the tactics representation discussed in Section 5.4.3, a user will be able to define rules for which units should focus on which enemy units. The *Targeter* sub-module in the *Tactical Planning* module described in Appendix B.6.9 will take care of executing the rules defined by the user.

Using Support: The tactics representation presented in Section 5.4.3 also defines rules for using support units and their spells/abilities. The *Support* sub-module in the *Tactical Planning* module described in Appendix B.6.8 is responsible for acting upon the rules defined in a tactic.

Predicting Resource Needs: The *Determine Resource Requirements* sub-module in the *Resource Management* module described in Appendix B.7.3 is responsible for predicting resource needs. It will use the plans for units, buildings and future research produced by other modules to make its prediction. By predicting resource needs, it can assign more workers to gather a certain resource before it should be used.

Flexible Resource Gathering: As a consequence of the AI's ability to predict resource needs, it will also use this knowledge to determine how many workers should be assigned to harvest each kind of resource. This all happens in the *Worker Planner* sub-module within the *Resource Management* module described in Appendix B.7.5.

Good Placement of Def. Buildings: A user of the framework has the ability to define how to place buildings in the base through the base building templates described in Section 5.4.4. The execution of base building templates is handled by the *Building Manager* sub-module in the *Base Building* module described in Appendix B.8.4.

Good Placement of Hrv. Buildings: The placement of harvesting buildings is also handled by the *Building Manager* sub-module of the *Base Building* module, which works on base building templates.

Sensible Base: The user is responsible for defining how buildings should be placed through base building templates, and the *Building Manager* sub-module is responsible for actually placing these buildings.

Scouting Map: The *Scouting* sub-module of the *Strategic Planning* module described in Appendix B.5.4, is responsible for the AI scouting the map in sensible places.

Scouting at Sensible Times: To scout at sensible times, the AI relies on the *Sufficient Enemy Knowledge* sub-module in the *Strategic Planning* module described in Appendix B.5.3. This sub-module will base its decisions of whether enough is known about the enemy, on information from the *Probabilistic Reasoning* module.

Learning: Learning is handled by the *Learning* module described in Appendix B.9. While learning new strategies is handled by the methods described in Section 5.4.1, learning new tactics and base building templates still needs some work.

Although not tested, the complete design of the framework should be able to handle all of the areas included in the test model.

8.6 RTS Specific Concepts

This section will discuss the two RTS specific ideas implemented in the prototype of the implementation: Strategy trees and pathfinding. For each, we will evaluate the success of the idea of the implementation in the prototype, and discuss the effect it would have in a complete implementation of the framework.

8.6.1 Strategy Trees

Strategy trees have been the foundation on which the *Probabilistic Reasoning* module and partly the *Strategic Planning* module have worked on in the prototype implementation. In both cases, the data structure and the ideas presented throughout the discussion of strategy trees in Section 5.4.1, have shown to work as intended, as discussed throughout this chapter. Section 8.1 showed how strategy trees could easily be specified, Section 8.2 showed how strategy trees could easily be configured to achieve different kind of AIs, and Section 8.5 showed how strategy trees have successfully helped in creating improvements in the AI. The game logs in Listing G.3 and Listing G.4 in the appendix furthermore demonstrate how the AI is able to follow a given strategy by building the required units. The following will list some of the most interesting advantages of using strategy trees:

Developer Friendly: The representation is straightforward and defines strategies and the relation between them in a simple manner, especially if depicted in a graphical user interface.

Versatile: Through strategy trees, an AI developer can create any kind of AI she wants, by simply adapting the strategic knowledge of the AI.

Built-in Operations: The data structure has through its representation natural support for finding counter strategies and follow-up strategies.

The only weakness of strategy trees in the prototype implementation, as mentioned in Section 8.1.2, is the lack of a graphical user interface. This would help provide a much needed overview when building large and complex strategy trees. There are also areas where the use of strategy trees can be improved compared to the prototype implementation. The following lists three areas which require further work:

Learning: One of the reasons for using strategy trees was the ability to easily add new strategies to a tree, and this way learn new strategies. Although adding the strategy itself is easy (as discussed in Section 5.4.1), learning new strategies includes other problems that must be solved as well. Two of them are the tasks of recognising that a new strategy is being used, and recognising new important game states.

Tactics and Base Building Templates: In a complete implementation of the framework, strategy tree nodes should contain both tactics and base building templates. These should support executing the strategy described by the node. A few key tasks regarding tactics require some further work however. First of all, the representation of tactics must be fully developed, and then a method to recognise these tactics must be composed. Secondly, a method making it possible to dictate when a certain tactic should be used during the execution of some strategy needs to be developed. This would also make it possible to dictate which tactics to use depending on the situation in the game, for instance whether the AI is attacking or defending.

Search Optimisation: When working with small strategy trees, like the ones for the game in this prototype implementation, the search through the strategy tree when finding matching nodes does not really matter. However, in more complex games, the strategy trees will consist of far more nodes, and when searching through this, optimised techniques should be used. There are many possibilities to guide a search for a matching node in a strategy tree. One way is to build a strategy tree for the opponent during a game, and this way guide the search in the tree, by only looking at nodes that are possible for the opponent to reach, given the strategy tree built for her. Another way could be

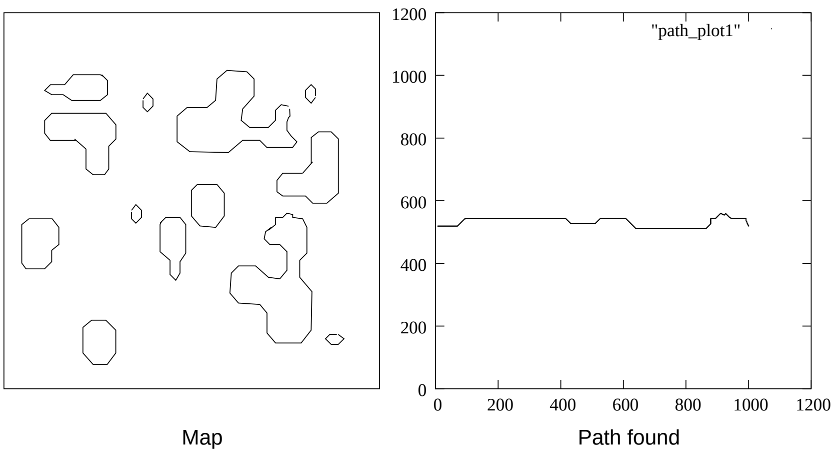


Figure 8.6: Path found in pathfinding test

to use the time variable on the strategy tree nodes, and only consider nodes that are within a certain time frame depending on the time since the game started. Finally, an optimised order of which attributes of strategy nodes are tested first for being close or equal to an attribute in another strategy node, could also result in a faster search.

8.6.2 Pathfinding

This section will first evaluate the pathfinder based on how well it finds the correct path (an optimal path) and how well it reduces the search space. Finally we will present some solutions to how the pathfinder can be improved.

Correctness

In order to verify that the pathfinder finds an optimal path we have tested it by making it find a path across a map with randomly placed obstacles. The pathfinder will start at the left side of the map and travel towards the right side. A test result can be seen in Figure 8.6 and additional test results in Appendix I. The tests show that the pathfinder indeed finds a reasonable path. We can furthermore conclude that the path is the optimal because of the algorithms used. A* will always find the optimal path [RN03]. The passable path that is responsible for restricting the search space is build upon A*. This means that the optimal path must be present within this restricted search space. Afterwards the optimal path itself is found within the passable path by using A*.

By basing both the passable path and the path itself on A* we conclude that the pathfinder always finds the optimal path.

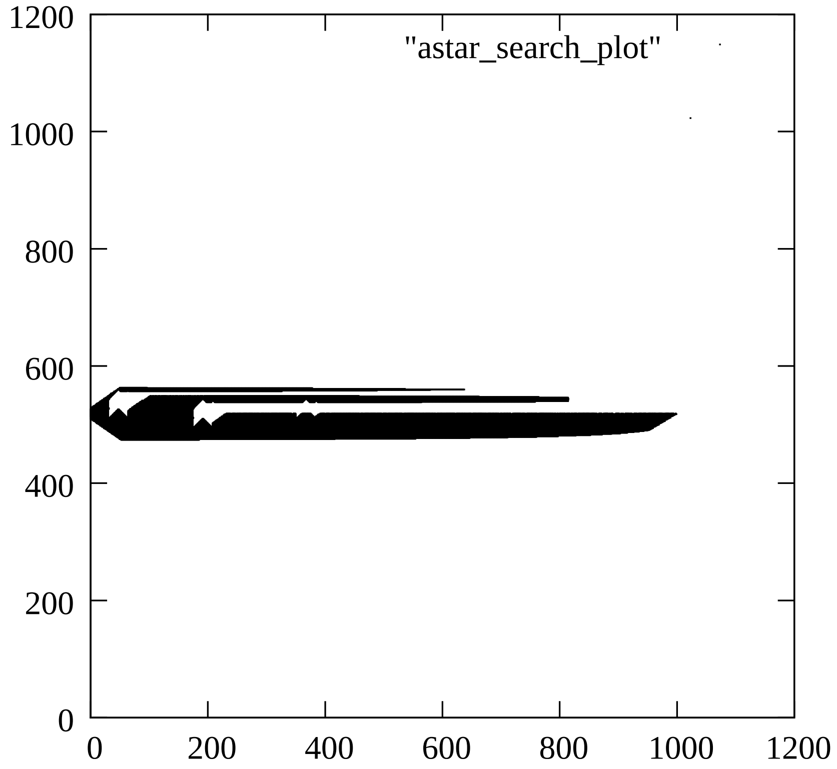


Figure 8.7: Search space explored by A*

Search Space

The test, to verify that the hierarchical JIT pathfinder reduces the search space, was carried out in the same type of environment as the previous test. Figure 8.7 shows the search space explored by A*, and the search space explored by the our pathfinder can be seen in Figure 8.8, when finding a path from the left side of the map to the right side of the map. A* explores 52289 cells, while our pathfinder explores 17152 cells. The search space can be further minimised by making the clusters contain a smaller amount of cells, but this will consequently mean a higher computation time to find the passable path. This means that finding the optimal solution is a balance between minimising the search space and the computation cost involved in doing so. The total length of the path and the amount and size of the obstacles on the map are also important factors that must be taken into account.

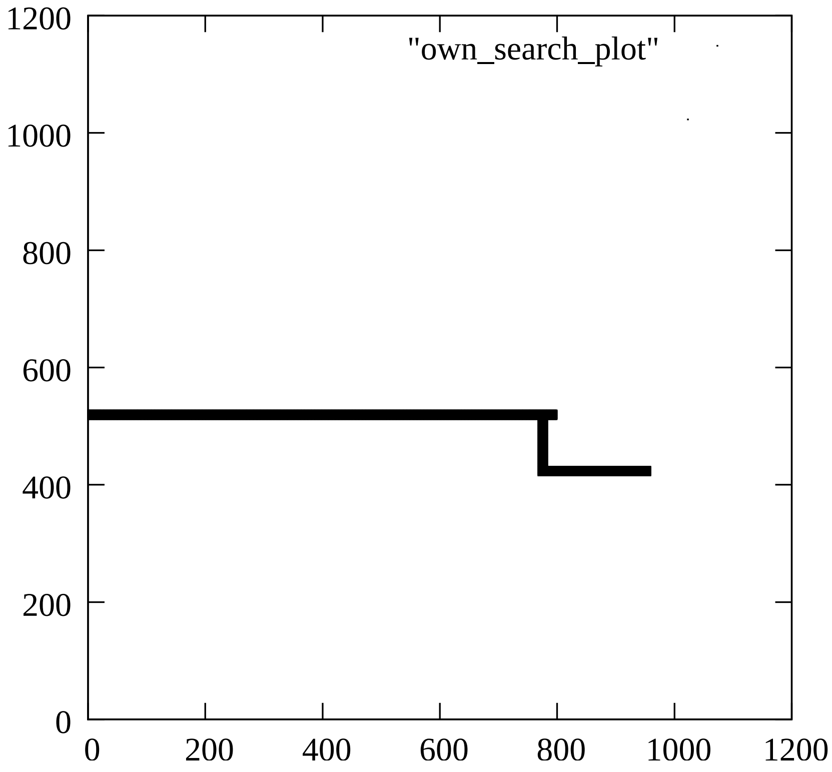


Figure 8.8: Search space explored by hierarchical JIT pathfinder

Improvements

We have identified two areas that must be improved in order for the pathfinder to get a reasonable performance. First the pathfinder must be run in a JIT fashion as was intended with the design. Second the data structure used in `open`[1] must be significantly reduced.

By simplifying the pathfinding to not include the JIT design means a trade-off between execution cost and development cost. A JIT pathfinder does not only affect the *Path Planner* in the *Tactical Planning* but also the *Action Planner*. Including such a functionality means some functionality must be shifted from the *Path Planner* to the *Action Planner*, which in itself is a small redesign of the framework.

Test have revealed that too much time is spent in the pathfinder maintaining `open`. Further investigation has shown that it specifically is the sheer size of the open elements that is the problem. Each element contains the path that currently has been travelled in order to reach the node in question. This is a list of coordinates unique to every single node in `open`. This list can be avoided and thus the total size of the open elements severely reduced by using back tracing.

8.7 Reflections

The following will reflect upon two things: The design presented in Part II and the development method of basing an AI framework on a model of how humans plays.

8.7.1 Design Reflection

The design of the framework presented in Part II and the design of each individual module presented in Appendix B have proven to be a big help throughout the implementation. We have been able to easily translate framework modules into C++ classes and sub-modules into functions. The design clearly separates different functionality into smaller manageable functions and furthermore clearly defines how the different functions work in relation to each other and the internal module architecture. Also, through the overall architecture of the framework, it has been easy to get an overview of where different knowledge bases have to be used.

In the prototype implementation, several small changes have been made compared to the design. As Section 7.1 explains, although the prototype implementation has had a focus on some areas compared to others, the implementation would still need to implement all modules except the *Learning* module. This means that some of the modules in the prototype implementation has not strictly followed the design, and only focused on making

[1] An important queue used in A*

a minimum implementation. This includes modules such as the *Resource Management* and *Base Building* modules. However, this has not affected the idea of the design of these modules.

The prototype implementation has also revealed some ideas for changing a few design details. One change induced by the prototype implementation is regarding how the *Action Planner* module is influenced by the *Strategic Planning*, *Tactical Planning*, *Resource Management* and *Base Building* modules. In the original design, the *Action Planner* was influenced by these modules by sending events. This has been changed to letting the *Action Planner* module be influenced by these modules through the shared in-game knowledge base *Assigned Unit Action*. The reason for this change is that with the original design, the *Action Planner* would be flooded with events of proposed unit actions. It is a much better choice to let modules change in the knowledge base, and then have the *Action Planner* iterate over the units in the knowledge base, when communicating actions to the GDF. This change is also reflected in the design of the *Action Planner* module described in Appendix B.10. Other issues of the implementation have concerned things like the exact data to send with events, the data structure used for different variables and knowledge bases, and how to construct developer friendly Python scripts. These were left as implementation details when designing the framework.

One of the concerns discussed throughout the chapter has also been the issue of providing solid documentation for the framework. As a start, the internal design of modules described in Appendix B provides both users and developers with a basic understanding of how things work in the framework. It specifies how framework instances can be varied from each other, and it defines the responsibilities of different sub-modules. The design is however, mostly suited for users of the framework that wish to extend or modify the framework in some way. Regular users will have more use of a user manual, which specifies exactly how each module and knowledge base can be configured to achieve different things. The more advanced users of the framework would be able to use the design explained in the Appendix, but would also benefit from tutorials on how to change the internal architecture of the framework.

8.7.2 Development Model Reflection

Using the human model as a basis for the framework does not mean that the framework can be used to all roles of an AI in RTS games. If for instance the RTS game is a simple game with only a few units and no countering system there is no need to bring up such an advanced solution as intended with this framework. The configuration will probably take longer than a simple scripting of the AI. Even in normal RTS games, the AI cannot be used for all aspects of the single player games. The human model will enable the AI to play like a human player in for instance a custom game, where the AI and the player will play against each other, but it will be less suited for campaigns, where the actions of the AI will be dictated by a story. One such example is the undead campaign in Warcraft III: Frozen Throne[warb], where the player will have to guide a hero through a dungeon filled with units controlled by the AI. These units have to act according to a story line, which is not covered by the human model. Luckily this is only a small part of the campaign and in some of the other parts, where the map dictates a playing style more like the one usually used in custom games, the AI is still well suited.

Handling cut scenes or scripted events in general can be done by adding an event module. This module must enable game designers to take direct control over anything in the GDF in order to get the wanted result. This is not normal human behaviour and is thus not covered by the human model. Consequently this module must be kept separate to the rest of the framework to maintain the correct abstraction. The module must also be kept separate from the rest of the framework, because it must have access to more than is allowed for the AI presented in our solution. While the event module is run, the rest of the framework must be stopped and then started again when the cut scene or event is over.

Furthermore the human model itself is based on our experience with the games shared with the experience of communities that are playing the games. The study of how a human player plays is a study of behaviour, psychology etc. that is not a part of traditional computer science. A more thorough study could probably enhance the model and lead to a better abstraction.

The current human model only models the tasks the human player solves when playing RTS games. When looking at the general areas of responsibilities the different modules have, it becomes apparent that the model might be generalised to model how a human player plays games in general. Some modules may have to be merged into more general areas and others have to be renamed in order to reflect the more general area of responsibility, but the model as such will remain the same. We will illustrate this through an example: When playing a FPS game like Quake, the player will basically go through the same process as modelled in the human model. She will still have to make an opponent model and compare this to what she know about the map, when deciding on a strategy. When meeting the opponent, she will have to use some tactics in order to engage in the right way, and if the enemy for instance throws a grenade after her, she will have to react instinctively. Resource management includes which weapons should be used, at which times, and which routes she must run in order to pick up various items before the opponent. Further work will probably reveal a human model that is structurally close to the one presented for RTS game, and able to model all game genres.

8.8 Summary

The first section of this chapter discussed how the framework could be configured to suit different demands from the developer. We demonstrated how AI designers were able to define technology trees, units, buildings and strategies, and how it is possible to vary AI behaviour through framework module configuration scripts in a simple manner. To further assist developers, we identified documentation and the inclusion of GUIs as possibilities for future work. However, if a developer wants to add new variables or new features, she will be required to extend modules and methods and this requires C++ programming knowledge. As most games, even within the RTS genres defined in Section 2.4.2, include some kind of special feature, the AI developer will often be required to extend the framework in order to support this. We hypothesise that the work required to do this will be minimum, as the framework already includes the most common attributes and features.

The test of the versatility of the framework was divided into two areas: Framework versatility and AI versatility. We concluded that framework versatility was difficult to evaluate, because the prototype implementation have only been connected to one GDF. However, the framework and GDF has been successfully separated, and the code required to connect the two is not very complex. In regards to AI versatility, we evaluated the two ways that developers can vary framework instances from each other: Through strategy trees and through module configuration. We verified that strategy trees can be specified to create any kind of strategic behaviour and that developers through module configurations are able to specify both game and AI specific variables that influence the AI's behaviour during a game. Both of these can be configured without having to re-compile the framework every time to see the effect, which helps provide easy balancing of variables and strategies.

Section 8.3 demonstrated how developers are able to extend modules and methods in the framework, as well as adding entirely new modules to the framework. Although it should be possible for designers to create prototype method extensions in Python, the task of extending the framework should mostly be left in the hands of C++ programmers, because of performance concerns. We concluded that modules and methods are quite easily extended with relatively little effort, but that adding entirely new modules is a more complicated task. It is possible to add new modules, but the developer must have extensive knowledge of the internal parts of the framework in order to ensure that no unexpected side effects occur. Once again, the issue of providing thorough documentation was found to be very important.

Our performance test in Section 8.4 showed that our prototype implementation suffers from performance problems. However, we have identified the problem as being the *Tactical Planning* module and more specifically our implementation of the pathfinding method described in Section 5.4.2. All other modules seems to be running at a reasonable performance. However, this does not say much as many of the modules are simplified versions or they are not called the appropriate number of times, because of the lack of effective pathfinding. The tests performed in this section have furthermore identified another important element to have included with the framework. As developers may also add and change modules, they will need a profiler to determine potential performance problems in their implementation.

In order to evaluate the AI improvements provided by both the prototype implementation and the complete implementation, we discussed their AI capabilities in relation to the test model described in Appendix D, which was also used to test commercial RTS games. This allowed us to compare our results with the current AI standard in the industry. We concluded that even the prototype implementation is able to handle areas not handled in many commercial RTS games today, such as counters, strategic variation and scouting. Furthermore, we discussed how each of the areas in the test model was handled by different framework sub-modules in the design of the framework, and hence also handled in a complete implementation.

The two RTS specific concepts used in the prototype implementation were evaluated in Section 8.6. Strategy trees have shown to be very useful in solving several of the scenarios in the test model, including making the AI able to use counters and strategic variation. They provide a developer friendly and versatile approach to defining strategies, and allow for several built-in operations useful in the internal parts of the framework. In a complete implementation, strategy tree nodes should come with both tactics and base building templates attached, and it should be used in the *Learning* module to learn new strategies. The pathfinder was shown to find the correct path and it also explored a smaller search space, as was expected.

Chapter 9

Discussion

This chapter will discuss if an AI framework for RTS games will be useful in the game development industry and present possible future work within this type of AI framework. To clarify the role of AI frameworks in the AI development of RTS games, we have contacted a number of game development companies to hear their views on the topic. A total list of the companies contacted can be found in Appendix L.1. The answers from these companies will be the topic of Section 9.1. In the end of this section, a number of industry demands for an AI framework will have been defined, and Section 9.2 will focus on discussing how the AI framework presented in this project, conforms to these requirements. Following this in Section 9.3, we will make a brief market analysis of RTS games and RTS game development companies, to get a clearer overview of the prospect of an AI framework for RTS games being used in the industry. This has not been presented earlier in the report, because it has been an enquiry done in parallel with the project. The enquiry was as such only done to establish whether the proposed AI framework would have any use in the industry. In section 9.4 other potential uses of the framework outside of game development industry will be presented. Finally in Section 9.5, we will discuss future work within the framework, and present the features that we believe will be the most challenging to design and implement.

9.1 Demand in Industry

This section will discuss the potential role of an RTS AI framework in the industry. To gain better insight into how AI development for RTS games is handled in the game development industry, we have contacted several RTS game development companies. These were sent a number of questions regarding AI development and the idea of a generic RTS AI framework, which can be seen in Appendix L.2. 4 out of 40 RTS game development companies responded and this section is based on their answers. There can be several reasons for why only four companies have responded to our enquiry:

- As most companies do not have a direct e-mail address used to contact the development team, we have been forced to use general information e-mail addresses to contact companies. This may have meant that some of our enquiries never reached developers who were qualified to answer our questions.
- One company responded that they simply did not have enough time to answer our enquiry at the given time. This may have been the case with other companies too.
- Finally, some of our questions are aimed at areas which companies may consider secret, and therefore do not wish to reveal to people outside the company.

The companies who answered our enquiry are: Oddlabs [Odd], Infinite Interactive [Inf], Inhuman Games [inh] and Fireglow Games [fir]. The following will list these companies, along with the games they have developed:

Oddlabs: Tribal Trouble (2004) [Tri].

Infinite Interactive: Warlords Battlecry II (2002) [wbia], Warlords Battlecry III (2004) [wbib] and Seven Kingdoms Conquest (in production) [sev].

Inhuman Games: Trash (2005) [tra].

Fireglow Games: Sudden Strike 3: Arms for Victory (2006) [sud].

Although these four companies are only representing about 10% of the total number of RTS game companies, which have produced RTS games within the last five years (shown in Appendix L.1), their answers should serve as an indication of different issues regarding AI development. Especially the developers from Infinite Interactive, who have produced several very popular RTS titles will be able to provide concrete answers of how AI development for RTS games is handled in the industry. We will divide this section into four different parts, each discussing an important element in AI development, and a final part discussing the merits of the idea of an RTS AI framework. The original answers to the questions asked in Appendix L.2 can be found in Appendix L.3.

9.1.1 Time Spent on AI Development

In general, the time spent on AI development varies from company to company. For games which were not under extreme time pressure, the development time seems to range from about 2000 to 5000 man hours. These numbers are rough estimates:

> *It's difficult to make a precise estimation, because vagueness of frames of which part of the game engine is AI and which is not.*
> - Max Dolmar, Fireglow Games

Max Dolmar also states that one of the most time consuming task of developing AI in RTS games is the issue of pathfinding.

9.1.2 Developers of AI

AIs in RTS games are primarily created by programmers with the help of the designers. Designers are in charge of the very high level part of the AI, including balancing, while programmers take care of the rest. There is however, a tendency to move towards having more of the AI that can be scripted by AI designers:

> *Programmers tend to do most of the AI development. Increasingly game designers with scripting ability are developing AI. Game designers tend to only control very high level aspects of AI.* - Mark Currie, Inhuman Games

Developers of *Warlords Battlecry* are for instance increasing their use of the scripting language Lua to get the designers more involved in the process of building the AI.

9.1.3 AI Development Tools

Two out of the four game development companies do not use any AI tools at all, and develops every AI for a new game from scratch. The developers of *Warlords Battlecry* uses a library built in-house as the foundation for the AI:

> *We have our own movement/pathing libraries on which everything is built. Everything apart from the movement and pathing is created from scratch on every game.* - Steve Fawkner, Infinite Interactive

The only company that uses an AI framework for development of the AI is the developers of *Sudden Strike*. They have built their own AI framework and uses this in connection with a third-party script system to create AIs for their games. Furthermore, it seems that the game development companies involved in making more than just one title, are better at using AI tools to reuse code from the AI in one game, to the AI in another game.

9.1.4 AI Integration with GDF

The separation of AI code from the GDF seems to vary greatly. Some companies have them completely separated, while others have them closely connected. It seems however, that companies with more than one shipped title, focus more on separating the two. The developers behind *Warlords Battlecry* keeps the two completely separated, but do include different functionality in the GDF for the AI to use:

> *They are kept completely separate. However, various functions of the engine have been added to help with AI, such as line-of-sight calculations.* - Steve Fawkner, Infinite Interactive

The AI framework used by the development team creating *Sudden Strike* has some modules that are totally independent of the GDF, while others are closely linked to the game and gameplay.

9.1.5 Generic RTS AI Framework

In general, all four companies are quite positive of the idea of a generic RTS AI framework. However, they all indicated that there have to be substantial advantages in using the framework, and that the framework has to deliver a number of advanced features especially important to AIs in RTS games. Otherwise, it would simply be too big a task to use and understand third-party software. The following lists a number of features that an AI framework for RTS games should contain facilities for, to be successful in saving AI developers a lot of time:

- Movement and Pathfinding
- Formations
- Influence Maps (e.g. for detection of danger)
- Threat Assessment
- Actions/Orders
- A State Machine of Actions of Individual Actors
- Grouping Mechanisms
- A Method for Tracking and Remembering Enemies
- Building and Production Hierarchies
- Resource Usage and Needs
- Managing and Prioritising Objectives

- Scripting System

Furthermore, the framework should be relatively easily connected to any kind of GDF. If an AI framework contained the above mentioned features, it would probably be used in the industry, as long as the quality was good and the prize affordable. One company, the developers of *Trash*, even went as far as guessing a possible prize on the product:

> *If your AI is the great, I think it could be sold. It would have to be extremely good and easy to integrate into any RTS game engine. If this was the case, perhaps you could charge $100k USD for it–if sold to big AAA studios*[1]. - Mark Currie, Inhuman Games

9.1.6 Summary

Only two of the four game development companies reuse their AI code, and one on them did this through very general AI libraries. The only company using an AI framework was the developers of *Sudden Strike*, but the scope of this is unknown. None of the companies take advantage of reuse to a degree comparable to the AI framework described in this project. Three of the companies have begun to focus on having designers more involved in the process of creating the AI, and they are using scripting languages to do this. There is a general consensus among the four companies that an RTS AI framework will be a good idea provided it lives up to a number of demands, making it possible to save a lot of time during development.

9.2 Conformance to Industry Demands

Section 9.1.5 listed a number of features an AI framework for RTS should support to save a lot of time during AI development. The complete design of our AI framework handles all of these areas. In the following we will list the framework modules that are involved in handling the different demands.

Strategic Planning: Threat assessments, grouping mechanisms and managing and prioritising objectives.

Tactical Planning: Movement and pathfinding, formations, influence maps, threat assessments, and managing and prioritising objectives.

Base Building: Building and production hierarchies.

Resource Management: Resource usage and needs.

Action Planner: Action/orders, and managing and prioritising objectives.

[1] AAA game development companies basically refers to companies producing large and expensive titles, which include a lot of PR/marketing.

Year:	RTS Games Published:
2002	20
2003	15
2004	27
2005	19
2006	20

Figure 9.1: RTS games published the last 5 years

Besides the demands handled by framework modules, some of the in-game knowledge bases also take care of a few of the demands. *Assigned Unit Action* and *Assigned Building Action* keeps track of which state the different units and buildings are in and *In-Game Enemy Knowledge* keeps track of enemy units and remember where they were last seen. Besides this, the framework as a whole, offers a scripting system, where all modules and knowledge bases can be configured. To summarise, our framework is designed to provide facilities for all the features listed in Section 9.1.5.

9.3 RTS Game Market Analysis

It is difficult to make an estimate of how many RTS games are produced every year, as there are no official records of this. Furthermore, the term *Real-Time Strategy* game is used to describe many different kinds of games and not only the ones being the focus of this project. Some would for instance characterise *Tetris* (1986) [tet] as being both a real-time and strategy game, and hence an RTS game. In this section we will only consider the RTS games that come into the category described in Section 1.1. We will use the popular game site Gamespot [gam] to identify published RTS games. Gamespot has records of any commercial RTS game relevant to this project. The number of RTS games published within the last five years can be seen in Table 9.3. The number includes both expansion packs and gold editions, as these often upgrade the AI compared to the original game. Furthermore, the number for 2006 is partially based on expected RTS game releases.

As Table 9.3 indicates, about 20 RTS games are released every year. Within the last five years, the production of these games have been handled by about 40 different RTS game development companies. It is difficult to estimate how many of these companies would have to buy the AI framework for it to be a worthwhile business. This depends on development cost, the prize of the framework and on the interest shown from game development companies. However, there are many potential buyers and if our answers from game development companies serve as any kind of indication, there may be up to 75% who would be able to benefit a lot from using the framework.

Furthermore, as many of the companies develop more than one title, the benefits of using the framework increase even more. The framework can be used on several titles, and this way increase code reuse. However, as observed in Section 9.1.3, it seems that game development companies involved in making more than one RTS game, often have their own AI tools to increase reuse and reduce development time.

9.4 Other Uses

The framework can also be used in other cases, where it is not the main AI development tool for an RTS game. One possibility is to use the framework for AI research. By providing a very modular framework, which includes standard implementations for each module, an AI researcher will be able to focus on a special area, probably a module, while letting all other modules be handled by the standard implementation. This provides new options for researchers wanting to focus on a certain aspect of AI. RTS games provide a platform for researching many fundamental aspects of AI [Bur03b]:

- Decision Making under Uncertainty
- Adversarial Real-Time Planning
- Reasoning
- Opponent Modelling
- Learning
- Resource Management
- Collaboration

Normally, researchers will have to build a test environment to use when testing their ideas, which is a time consuming task and not interesting from an AI research perspective. Using RTS games as a test environment combined with an AI framework, provides optimal conditions for AI researchers. They can focus on specialised areas, for instance learning, while leaving everything else to the framework and test their ideas in a complex environment.

9.5 Further Work

This section will discuss possible further work that can be done following the prototype implementation in this project. We will start by discussing what is needed to create a full implementation of the framework. Following that, we will discuss how the work done in this project can influence the creation of AI frameworks for other game types.

9.5.1 Complete Implementation

A complete implementation of the framework presented in this project will first and foremost require that all the remaining features presented in the design in Chapter 6 are implemented. A couple of areas require further work before being applicable in the framework. These are listed below:

Tactics: First, the concept of how to represent tactics presented in Section 5.4.3 must be expanded to deal with several other tactical issues such as using terrain, using support units and figuring out how to counter the opponent's tactics. Secondly, the framework must include methods for properly executing the different tactics defined by the user, through the rules defined in the tactics representation. Finally, work must be done in trying to develop a generic method for recognising tactics in all kinds of RTS games. This method should be based on the representation of tactics.

Cooperation: In this project we chose to disregard all cooperation features in the framework. This would not be an option in a full implementation, as cooperation has an important role in most RTS games. In the human model in Chapter 3, we defined cooperation as being a task influencing all other tasks and as being itself influenced by all other tasks. Translating this to the framework architecture, it would probably be a form of global module, dictating orders to other modules. More work has to be done in this area to ensure proper cooperation between both allied AIs and allied human players.

Graphical User Interface: Section 8.1 and Section 8.1.5 presented the idea that a GUI would be a big help in ensuring a developer friendly framework. This idea should be further developed, and developer friendly methods for the user to define strategy trees, technology tree, unit types etc. must be defined.

Documentation: The documentation for the prototype implementation is limited to the framework design descriptions in Appendix B and configuration examples in Python files along with comments on what each configuration variable means. However, frameworks are in general large and complex pieces of software and quite difficult to understand, and therefore proper documentation is required [FHLS97]. A lot of research has been done on documenting frameworks [BKM00]. Normally, one divides documentation of frameworks into three separate areas [Joh92]:

- The purpose of the framework.
- How to use the framework.
- The detailed design of the framework.

Work must be done in ensuring the right choice of documentation for a framework of this type, which includes the three areas mentioned above.

With all ideas fully developed, we can take a closer look at the features which were not implemented in the prototype implementation. In the following we will list some of the most interesting missing features, and present some of the challenges of a complete implementation.

Learning: The design of the *Learning* module is specified in Appendix B.9. Methods for learning new strategies have already been discussed throughout the discussion of strategy trees in Section 5.4.1. The biggest challenges when implementing the module will be to devise a method for learning new tactics and base building templates. Furthermore, a method is needed to properly revise strategies, tactics and base building templates.

Base Building Templates: The framework will have to use the ideas presented in Section 5.4.4. This includes letting users define their own templates, and defining internal methods in the framework that are able to use any given template. Work must also be done on how to build a few generic templates, which will work in any RTS game. These would serve as a standard way of handling base building.

Building Planner: This sub-module is part of the *Base Building* module, and is basically responsible for planning which buildings to build and when. It must given a strategy and base building template, plan where and what to build, taking the technology tree and the resources available into consideration. Further details of this sub-module can be found in Appendix B.8.5.

Unit Planner: This sub-module is part of the *Action Planner* and its responsibilities are similar to that of the *Building Planner*. It must consider all the same things, but here the module must create a plan for when and what units are to be built. The sub-module is discussed in further detail in Appendix B.10.3.

Research Planner: The *Action Planner* also contains this sub-module, which is responsible for deciding when and what research is to be purchased. Work must be done in developing generic methods able to deal with all the different kinds of research options available in different RTS games.

Action Scheduling: While *Building Planner*, *Unit Planner* and *Research Planner* all focus on each their area, the *Action Planner* must decide which of the suggested actions from each sub-module are to be executed first. The challenge of creating this sub-module is to devise a sensible way for developers to specify rules for how the module should prioritise the different actions. This sub-module is discussed in further detail in Appendix B.10.5.

Advanced Scouting: The prototype implementation has a very simple form of scouting. The complete implementation must both include features for scouting the right attributes of the enemy and for scouting the right places. The execution of scouting is as such not difficult, the task when creating this sub-module will be to let a developer define precise rules for when and what to scout. A further discussion of this can be found in Appendix B.5.4.

Advanced Execution of Strategies: While the prototype implementation has a very simplified execution of strategies, the complete implementation must feature the *States* sub-module explained in Section B.5.10. However, this method must be tested in terms of its ability to handle very specialised situations and a method for developers to easily define the execution of strategies must be devised.

Advanced Situation Assessment: Appendix B.5.9 presented a simple generic way of evaluating a situation. However, as also explained in that section, evaluating a situation is game specific, and a way for a developer to define how to do this will be necessary in most games. Work must be done in attempting to devise a way to do this without the developer being required to have knowledge of C++ programming.

In general, further work must be done in testing different aspects of the framework. The prototype implementation has only been connected to one GDF, and to really test the versatility of the framework, it must be tested with other GDFs as well. Furthermore, the GDF used for the prototype implementation is primarily used to create games in one of the RTS genres defined in Section 2.4.2. This means that the framework has not yet been tested in regards to its ability to create AIs for all the different genres. However, as discussed in the evaluation in Chapter 8, this should not be a problem.

9.5.2 AI Frameworks in General

This project also has uses outside the domain of AI frameworks in RTS games. We have through the project demonstrated how it is possible to reuse large parts of the AI for a specific game genre. We hypothesise that this is possible for more genres than just RTS games. This report can serve as a guideline of how to construct a framework specifically suited for one type of game. This includes building a human model of how a human will play the game as well as using this to create a framework architecture. This development model is discussed in more detail in Section 8.7.2. Furthermore, a number of examples have been described, of how to create special data structures particularly useful for both representing and learning domain specific knowledge in the game type being focused on.

Chapter 10

Conclusion

In this chapter we will conclude on the project and present the primary contributions made to academia and game development, described throughout this report. This project has presented the design of an AI framework for RTS games. We have based this work on our pre-master thesis [FKL05], which defined a model of how humans plays RTS games and suggested a preliminary design. This report presented a revised human model in Chapter 3, which defined the tasks an RTS game consists of and how these influence each other. We used this model as a starting point when we designed the framework architecture.

Throughout the report we have presented a number of design techniques used when designing the framework. We have demonstrated how these techniques can be used to enhance AI development in the game development industry as well as demonstrated a new area of application for these techniques for the academic world. Some of the known design techniques used include frameworks, scripting languages and event systems, but we also presented four new concepts specifically suited to create AIs in RTS games. These four techniques provides the foundation for the AI framework. In Section 5.4.1 we presented the idea of *strategy trees*, which is a data structure specifically suited to represent strategies in RTS games. Following this, we focused on *pathfinding*, which is an important element of any RTS game, and we presented a new method of doing this, optimised for working in an RTS game environment. The third RTS specific technique introduced was the notion of a representation of *tactics*. We presented a general approach to how this can be done in a way that AI designers can specify tactics specifically suited to the game they are working with. Finally, the fourth technique focused on what we chose to call *base building templates*. These were created to allow AI designers to specify how an AI should construct its base in a particular game using a certain strategy. Strategy trees, tactics and base building templates have all contributed to AI development within the RTS genre, by presenting new ways of representing AI specific data. They allow a developer friendly and generic representation, which can be reused in different RTS game genres. Furthermore, they allow developers to compose new kinds of strategies, tactics and base building templates, by combining small building blocks consisting of rules or strategies. Our pathfinding idea has not only shown a new way for AI developers to optimise pathfinding in their games by reducing the explored search space and distributing computations over several game ticks, but also contributed to the general academic research within this area, which has many applications outside RTS games.

The design of the AI framework was presented in Chapter 6, and followed the design goals outlined in Section 4.1. These included improving the AI, reducing development cost, creating a workload shift from programmers to designers and creating a structured overview of the development process of creating AIs for RTS games. The fulfilment of these design goals through our AI framework has contributed to the game development industry by presenting and implementing a design capable of achieving these goals. By using the human model defined in Chapter 3 as a foundation and by drawing upon knowledge of framework capabilities and characteristics, we created a cognitive architecture for the framework, described in Section 6.1. As for the non-trivial knowledge representation in the framework, we used the RTS specific concepts defined in Section 5.4: Strategy trees, tactics and base building templates. The detailed design in Appendix B furthermore specified sub-modules in each framework module, defining their responsibilities and proposed hot spots. Finally, in Section 6.3 we presented how a user should be able to vary framework instances from each other and in Section 6.4 we presented an event system designed to control our AI framework. The design of the framework has contributed to academia by combining three well-known design techniques in a new area of application in order to maximise reuse, secure a user friendly framework and create a clear separation of framework modules.

As a proof of concept, a prototype implementation of the framework was implemented and connected to the ORTS GDF. This GDF included a simple RTS game, which had all the necessary features required to test different AI capabilities. In order to allow inexperienced programmers to use the framework, we used the scripting language Python to configure framework modules and knowledge bases. The evaluation of the prototype implementation was presented in Chapter 8 and discussed six framework evaluation areas: Configurability, Versatility, Extendibility, Performance testing, AI improvements and test of RTS Specific Concepts. We concluded that an AI designer is able to configure strategies, technology trees and modules without much effort. However, if new attributes or features in a game are to be included, experienced C++ programmers must extend modules or methods in the framework. As long as the required AI for the game does not deviate too much from the internal framework architecture, this can be done without any problems. We hypothesise that this is the case with most RTS games within the four genres mentioned in Section 2.4.2. Our performance test revealed one major performance problem in the implementation of our pathfinding idea. However, we have identified the problem and presented a solution to overcome it in Section 8.6.2. The test of the AI created with the prototype implementation was presented in Section 8.5, and showed that the AI is able to both scout and vary its strategy by using counters. The ability to counter and vary its strategies is a direct consequence of using strategy trees as the representation of strategies in the framework. The prototype implementation has contributed to the game development industry by demonstrating to what degree an AI designer can develop AIs and how strategy trees in particular simplifies creating AIs with capabilities beyond those of current commercial RTS game AIs.

In Chapter 9, we presented a discussion of the potential use of the AI framework in the game development industry. We contacted several RTS game development companies, and set up demands that the framework had to fulfil to be useful in the industry. We then demonstrated how each of these demands set by the industry were fulfilled by our framework architecture. Furthermore, we analysed the market for an AI framework for RTS games, and concluded that there are many potential buyers and if our enquiry served as any kind of indication, most companies would be able to benefit from the use of our AI framework.

There are many possibilities for future work based on this project. The logical entailment following the prototype implementation would be to implement a complete implementation. This will require work in several different areas as also presented in Section 9.5. First of all, this report has contributed by identifying key areas which requires further work before they can be used in an actual implementation. This is the case with the tactics representation presented in Section 5.4.3 and the *Cooperation* task in the human model, which were not initially included in the design of the framework. Furthermore, throughout the evaluation in Chapter 8, two issues were found to be very important when users are to use the framework: Documentation and Graphical User Interfaces. Documentation is required for both designers and programmers to use the framework efficiently and this must be added for a complete implementation to be of any use. GUIs must be added to aid AI designers in designing strategy trees, technology tree, unit types etc. and for assisting programmers in creating modules. Both the documentation of frameworks and the creation of a developer friendly interface for the framework, are interesting areas from an academic research perspective. Finally, there are also still work to be done in regards to the *Learning* module. Learning strategies is well-defined through strategy trees, but work must be done on how to learn tactics and base building templates. Machine learning is as such a well-known discipline within machine intelligence research, however, the techniques described in our solution presents not only a new method of doing so, but also a new area of application.

The work done in this project may be used in other areas than developing complete AI solutions. One area may be within AI research. As stated in Section 9.4, our framework provides an opportunity for AI researchers to focus their attention on a certain area of RTS AI. RTS games are interesting for AI researchers, because they contain a number of interesting AI problems in well-defined environments, and our framework allows researchers to easily focus on one of them. This way we contribute to academia by providing a research platform for AI development.

The development method used in this project may also be used in future work. We hypothesise that the method of creating a human model for a certain game type, and then transforming this to a cognitive framework architecture is applicable in other game genres than the RTS genre. Further work may be done in both academia and within game development on developing a general human model, suitable to describe a human player in all kinds of game genres. Furthermore, a thorough study of the human model could probably enhance the model and lead to a better abstraction.

Chapter 11

Resume

It is a widely acknowledged fact that the AIs found in computer games are of a poor quality. Consequently this means that the gameplay that relies on the AIs also suffers. Players have found the answer to this problem by seeking challenges in playing against other players. Producers of Real-Time Strategy games have attempted to improve the quality of AI by allowing it to cheat, creating general and static solutions or by focusing on scripting the AI to perform a certain strategy as fast as possible. Neither of these methods have brought the AI to a standard, where it can resemble that of a human player. In order to enhance the gameplay in the game parts that rely on the performance of the AI, it will have to play with near human capabilities. We approach this problem by building a human model of how a human player plays and use this as a basis for a general framework for building AIs to Real-Time Strategy games. The preliminary work that served as a base for the model and the design of the framework was carried out in our pre-master thesis [FKL05].

Throughout the report we have presented a number of design techniques used when designing the framework. We have demonstrated how these techniques can be used to enhance AI development in the game development industry as well as demonstrated a new area of application for these techniques for the academic world. Some of the known design techniques used include frameworks, scripting languages and event systems, but we also presented four new concepts specifically suited to create AIs in RTS games. These four techniques provides the foundation for the AI framework. First we presented the idea of *strategy trees*, which is a data structure specifically suited to represent strategies in RTS games. Following this, we focused on *pathfinding*, which is an important element of any RTS game, and we presented a new method of doing this, optimised for working in an RTS game environment. The third RTS specific technique introduced was the notion of a representation of *tactics*. We presented a general approach to how this can be done in a way that AI designers can specify tactics specifically suited to the game they are working with. Finally, the fourth technique focused on what we chose to call *base building templates*. These were created to allow AI designers to specify how an AI should construct its base in a particular game using a certain strategy. Strategy trees, tactics and base building templates have all contributed to AI development within the RTS genre, by presenting new ways of representing AI specific data. They allow a developer friendly and generic representation, which can be reused in different RTS game genres. Furthermore, they allow developers to compose new kinds of strategies, tactics and base building templates, by combining small building blocks consisting of rules or strategies. Our pathfinding idea has not only shown a new way for AI developers to optimise pathfinding in their games, but also contributed to the general academic research within this area, which has many applications outside RTS games.

The design of the AI framework followed four design goals: Improving the AI, reducing development cost, creating a workload shift from programmers to designers and creating a structured overview of the development process of creating AIs for RTS games. The fulfilment of these design goals through our AI framework has contributed to the game development industry by presenting and implementing a design capable of achieving these goals. By using the human model as a foundation and by drawing upon knowledge of framework capabilities and characteristics, we created a cognitive architecture for the framework. As for the non-trivial knowledge representation in the framework, we used the RTS specific concepts mentioned earlier: Strategy trees, tactics and base building templates. Finally, we presented an event system designed to control our AI framework, and we presented how a user should be able to vary framework instances from each other. The design of the framework has contributed to academia by combining three well-known design techniques in a new area of application in order to maximise reuse, secure a user friendly framework and create a clear separation of framework modules.

As a proof of concept, a prototype implementation of the framework was implemented and connected to the ORTS game development framework. This game development framework included a simple RTS game, which had all the necessary features required to test different AI capabilities. In order to allow inexperienced programmers to use the framework, we used the scripting language Python to configure framework modules and knowledge bases. We then evaluated the prototype implementation in relation to six important areas: Configurability, Versatility, Extendibility, Performance testing, AI improvements and test of RTS Specific Concepts. Our performance test revealed one major performance problem in the implementation of our pathfinding idea. However, we have identified the problem and presented a solution to overcome it. The test of the AI created with the prototype implementation showed that the AI is able to both scout and vary its strategy by using counters. The ability to counter and vary its strategies is a direct consequence of using strategy trees as the representation of strategies in the framework. The prototype implementation has contributed to the game development industry by demonstrating to what degree an AI designer can develop AIs and how strategy trees in particular simplifies creating AIs with capabilities beyond those of current commercial RTS game AIs.

We also presented a discussion of the potential use of the AI framework in the game development industry. We contacted several RTS game development companies, and set up demands that the framework had to fulfil to be useful in the industry. We then demonstrated how each of these demands were fulfilled by our framework architecture. Furthermore, we analysed the market for an AI framework for RTS games, and concluded that there are many potential buyers and if our enquiry served as any kind of indication, most companies would be able to benefit from the use of our AI framework. We then presented a number of possible areas for future work. This included creating a complete implementation of the framework as well as further developing some of the ideas presented throughout this report. Furthermore, we hypothesised that the technique of building an AI framework architecture based on a human model is applicable in other domains than the RTS game genre.

Bibliography

[age] Age of Empires.
http://www.ageofempires.com/.

[AII] Artificial Intelligence and Interactive Digital Entertainment.
http://www.aiide.org/.

[aom] Age of Mythology.
http://www.microsoft.com/games/ageofmythology/.

[aox] Armies of Exigo.
http://www.aox.ea.com/.

[Ban] Black & White.
http://www.lionhead.com/.

[BASC05] Michael Buro, David W. Aha, Nathan Sturtevant, and Vincent Corruble.
Complex Video Game AI Competitions at AIIDE'2006. 2005.

[bat] Battle.net Homepage.
http://www.battle.net/.

[BF04a] Michael Buro and Timothy Furtak.
RTS Games and Real-Time AI Research. Behavior Representation in Modeling and Simulation Conference (BRIMS), 2004. In Proceedings.

[BF04b] Michael Buro and Timothy Furtak.
RTS Games as Test-Bed for Real-Time Research. Behaviour Representation in Modeling and Simulation Conference (BRIMS), 2004.

[BF05] Michael Buro and Timothy Furtak.
On the Development of a Free RTS Game Engine. 2005.

[BKM00] Greg Butler, Rudolf K. Keller, and Hafedh Mili.
A Framework for Framework Documentation. *ACM Comput. Surv.*, 32(1es):15, 2000.

[bli] Blizzard Entertainment.
http://www.blizzard.com/.

[BMS04] Adi Botea, Martin Muller, and Jonathan Schaeffer.
Near Optimal Hierarchical Path-Finding. *Journal of Game Development*, 2004.

[Bue98] Jesús Cerquides Bueno.
KDCOM: A Knowledge Discovery Component Framework. Master's thesis, Campus, UAB, Barcelona, Spain, 1998.

[Bur02] Michael Buro.
ORTS - A Hack-Free RTS Game Toolkit. October 2002.

[Bur03a] René Burgess.
Realistic Evaluation of Terrain by Intelligent Natural Agents. Master's thesis, Campus UAB, Barcelona, Spain, September 2003.

[Bur03b] Michael Buro.
Real-Time Strategy Games: A New AI Research Challenge. 2003.

[Bur04] Michael Buro.
Call for AI Research in RTS Games. AAAI-04 AI in Games Workshop, 2004. San Jose.

[CBS05] Michael Chung, Michael Buro, and Jonahthan Schaeffer.
Monte Carlo Planning in RTS Games. 2005.

[CCT89] N. V. Carlsen, N. J. Christensen, and H. A. Tucker.
An Event Language for Building User Interface Frameworks. In *UIST '89: Proceedings of the 2nd annual ACM SIGGRAPH symposium on User interface software and technology*, pages 133–139, New York, NY, USA, 1989. ACM Press.

[CLR90] Thomas H. Cormen, E. Leiserson, Charles, and Ronald L. Rivest.
Introduction to Algorithms. MIT Press, 1990. COR th 01:1 1.Ex.

[com] Command & Conquer.
http://westwood.ea.com/.

[cou] Half-Life: Counter-Strike.
http://www.counter-strike.net/.

[CSL] CSLI's Computational Learning Laboratory Homepage.
http://cll.stanford.edu/.

[dar] Dark Reign.
http://www.auran.com/games/darkreign/default.htm/.

[Daw] Bruce Dawson. GDC 2002: Game Scripting in Python.
http://www.gamasutra.com/features/20020821/dawson_pfv.htm.

[Del01] Mark Deloura.
Game Programming Gems 2.
Charles River Media, 2001.

[des] Descent 3.
http://www.descent3.com/.

[diga] Digital Drama Studios.
http://www.digitaldramastudios.com/.

[digb] Digitalmill.
http://www.dmill.com/.

[dJSR05] Steven de Jong, Pieter Spronck, and Nico Roos.
Requirements for Resource Management Game AI. International Joint Conferences on Artificial Intelligence, 2005. Workshop on Reasoning, Representation, and Learning in Computer Games.

[dox] Doxygen.
http://www.stack.nl/ dimitri/doxygen/.

[dun] Dune II.
http://duneii.com/.

[EE1] Empire Earth.
http://www.empireearth.com/.

[EE2] Empire Earth 2.
http://www.empireearth2.com/.

[FCGC02] Carlos J. Fernandez-Conde and Pedro A. Gonzalez-Calero.
Domain Analysis of Object-Oriented Frameworks in FrameDoc.
In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 27–33, New York, NY, USA, 2002. ACM Press.

[fea] F.E.A.R.
http://www.whatisfear.com/.

[Feu97] Alan R. Feuer.
MFC Programming.
Addison Wesley Professional, 1997.

[FFM03] Ludger Fiege, Felix Freiling, and Gero Muehl.
Modular Event-Based Systems. *Knowledge Engineering Review*, 17(4), 2003.

[FHLS97] Gary Froehlich, H. James Hoover, Ling Liu, and Paul Sorenson.
Hooking into Object-Oriented Application Frameworks. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 491–501, New York, NY, USA, 1997. ACM Press.

[fir] Fireglow Games.
http://www.fireglowgames.com/.

[FKL05] Kasper Frederiksen, Kasper Kristensen, and Anders Lauritsen.
Towards an AI Framework for RTS Games. Pre-master thesis, December 2005.

[FS97] Mohammed Fayad and Douglas Schmidt.
Object Oriented Application Framework. *Communications of the ACM*, 40(10), 1997.

[FSJ97] Mohammed E. Fayed, Douglas C. Schmidt, and Ralph E. Johnson.
Object-Oriented Application Frameworks: Problems & Perspectives. *Wiley, NY*, 1997.

[gam] Gamespot.
http://www.gamespot.com/.

[GHG04] Thore Graepel, Ralf Herbrich, and Julian Gold.
Learn to Fight. International Conference on Computer Games: Artificial Intelligence, Design and Education, 2004.

[god] God Games Definition.
http://en.wikipedia.org/wiki/God_game.

[gpr] GNU gprof.
http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html.

[Hal] Half-Life.
http://www.valvesoftware.com/.

[HF04] Stuart Hansen and Timothy Fossum.
Events Not Equal To GUIs. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 378–381, New York, NY, USA, 2004. ACM Press.

[HNOR88] Tor Hauge, Inger Nordgard, Dan Oscarsson, and Georg Raeder. Event-Driven User Interfaces Based on Quasi-Parallelism. In *UIST '88: Proceedings of the 1st annual ACM SIGGRAPH symposium on User Interface Software*, pages 66–76, New York, NY, USA, 1988. ACM Press.

[Hue] Robert Huebner. Adding Languages to Game Engines". http://www.gamasutra.com/features/19971003/huebner_01.htm.

[Inf] Infinite Interactive. http://www.infinite-interactive.com/.

[inh] Inhuman Games. http://inhumangames.com/.

[Jav] Java. http://www.java.sun.com/.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JG05] Joshua Jones and Ashok Goel. Knowledge Organization and Structional Credit Assigment. International Joint Conferences on Artificial Intelligence, 2005. Workshop on Reasoning, Representation, and Learning in Computer Games.

[Joh92] Ralph E. Johnson. Documenting Frameworks using Patterns. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 63–76, New York, NY, USA, 1992. ACM Press.

[Joh97] Ralph E. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, 40(10), 1997.

[jsw05] Creating a GUI with JFC/Swing. http://java.sun.com/docs/books/tutorial/uiswing/, 2005.

[kal] Kali.net. http://kali.net.

[KNYH05] Bharat Kondeti, Maheswar Nallacharu, Michael Youngblood, and Lawrence Holder. Interfacing the D'Artagnan Cognitive Architecture to the Urban

Terror First-Person Shooter Game. pages 55–60. International Joint Conferences on Artificial Intelligence, 2005. Workshop on Reasoning, Representation, and Learning in Computer Games.

[Lai01] John E. Laird.
Using a Computer Game to Develop Advanced AI. *Computer*, 34(7):70–75, 2001.

[Lai03] John E. Laird. The Soar 8 Tutorial.
http://sitemaker.umich.edu/soar/, 2003.

[Lew98] Scott M. Lewandowski.
Frameworks for Component-Based Client/Server Computing, 1998. S.M. Lewandowski, Frameworks for Component-Based Client/Server Computing, ACM Computing Surveys, Vol. 30, No. 1, Mar. 1998.

[Lis] Lisp.
http://www.clisp.org/.

[LL99] Michael van Lent and John Laird.
Developing an Artificial Intelligence Engine. pages 577–588, San Jose, CA, March 1999. Game Developers Conference.

[LL01] Michael van Lent and John Laird.
Human-Level AI's Killer Application. AAAI, 2001.

[LL02] Pat Langley and John E. Laird.
Cognitive Architectures: Research Issues and Challenges. 2002.

[lot] Lord of the Rings: Battle for Middle-Earth.
http://lotr.ea.com/.

[Lua] Lua.
http://www.lua.org/.

[MBF99] Michael Mattsson, Jan Bosch, and Mohamed E. Fayad.
Framework Integration Problems, Causes, Solutions. *Commun. ACM*, 42(10):80–87, 1999.

[ML01] Marcus Eduardo Markievicz and Carlos J.P. Lucena.
Object Oriented Framework Development. *Crossroads, ACM Press*, 7(4):3–9, 2001.

[moo] Moore's Law.
http://www.webopedia.com/TERM/M/Moores_Law.html.

[msr] Microsoft Research.
http://research.microsoft.com/.

[MSWT05] Richard Maclin, Jude Shavlik, Trevor Walker, and Lisa Torrey. Knowledge-Based Support-Vector Regression for Reinforcement Learning. International Joint Conferences on Artificial Intelligence, 2005. Workshop on Reasoning, Representation, and Learning in Computer Games.

[Nar02] Alexander Nareyek.
Intelligent Agents for Computer Games. Computers and Games, 2002.

[Odd] Oddlabs.
http://www.oddlabs.com/.

[Ort] Orts Game Specification.
http://www.cs.ualberta.ca/˜mburo/orts/AIIDE06/game3.

[ORT05] ORTS Homepage.
http://www.cs.ualberta.ca/˜mburo/orts/orts.html, 2005.

[Ous98] John K. Ousterhout.
Scripting: Higher-Level Programming for the 21st Century. *Computer*, 31(3):23–30, 1998.

[pac] Pac-man.
http://en.wikipedia.org/wiki/Pacman.

[Per] Perl.
http://www.perl.com/.

[pon] Pong.
http://en.wikipedia.org/wiki/PONG.

[PP04] Andrew M. Phelps and David M. Parks.
Fun and Games: Multi-Language Development. *Queue*, 1(10):46–56, 2004.

[Pyta] Python.
http://www.python.org/.

[Pytb] Python Success Stories.
http://www.python.org/about/success/.

[quaa] Quake.
http://www.idsoftware.com/games/quake/quake/.

[quab] Quake II.
http://www.idsoftware.com/games/quake/quake2/.

[red] Red Alert.
http://www.ea.com/official/cc/firstdecade/us/index.jsp/.

[RKD00] Douglas A. Reece, Matt Kraus, and Paul Dumanoir.
Tactical Movement Planning for Individual Combatants. 9th Conference on Computer Generated Forces and Behavioral Representation, 2000. In Proceedings.

[RN03] Stuart Russell and Peter Norvig.
Artificial Intelligence A Modern Approach.
Prentice Hall, 2003.

[rts] Definition of RTS Games.
http://en.wikipedia.org/wiki/Real-time_strategy.

[Saw02] Ben Sawyer.
Serious Games: Improving Public Policy through Game-based Learning and Simulation. Woodrow Wilson International Center for Scholars, 2002. Technical Report.

[SC95] Douglas C. Schmidt and James O. Coplien.
Pattern Languages of Program Design.
Addison-Wesley, 1995.

[Sch04] Brian Schwab.
AI Game Engine Programming.
Charles River Media, 2004.

[sev] Seven Kingdoms Conquest.
http://www.enlight.com/7kc/.

[SG86] Robert W. Scheifler and Jim Gettys.
The X Window System. *ACM Trans. Graph.*, 5(2):79–109, 1986.

[SL94] Alexander A. Stepanov and Meng Lee.
The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, 1994.

[Soa] Soar Homepage.
http://sitemaker.umich.edu/soar.

[sta] Starcraft.
http://www.blizzard.com/starcraft/.

[Str] Stratagus Homepage.
http://www.stratagus.sourgeforge.net/.

[sud] Sudden Strike 3: Arms for Victory.
http://www.suddenstrike.com/.

[SZ04] Katie Salen and Eric Zimmerman.
Rules of Play.
The MIT Press, 2004.

[Tcl] Tcl.
http://www.tcl.dk/.

[tet] Tetris.
http://en.wikipedia.org/wiki/Tetris.

[tib] Command & Conquer: Tiberian Sun.
http://www.ea.com/official/cc/firstdecade/us/tiberiansun.jsp.

[tra] Trash.
http://inhumangames.com/.

[Tri] Tribal Trouble.
http://tribaltrouble.com/.

[UB06] Tapani Utriainen and Michael Buro.
ORTS Competition: Getting Started.
http://www.cs.ualberta.ca/˜mburo/orts/AIIDE06/getting_started.pdf,
May 19 2006.

[UGJM05] Patric Ulam, Ashok Goel, Joshua Jones, and William Murdock.
Using Model-Based Reflection to Guide Reinforcement Learning. International Joint Conferences on Artificial Intelligence, 2005. Workshop on Reasoning, Representation, and Learning in Computer Games.

[wara] Warcraft II.
http://www.blizzard.com/war2bne/.

[warb] Warcraft III.
http://www.blizzard.com/war3/.

[warc] Warzone 2100.
http://en.wikipedia.org/wiki/Warzone_2100/.

[wbia] Warlords Battlecry II.
http://www.infinite-interactive.com/wbc2/.

[wbib] Warlords Battlecry III.
http://www.infinite-interactive.com/wbc3/.

[Øst99] Kasper Østerbye.
Minimalist Documentation of Frameworks, 1999.

Part IV

Appendix

Appendix A

Terms and Expressions

The purpose of this section is to introduce a number of terms and expressions that will be used throughout the report. The terms have been sorted into five areas: General concepts, buildings, units, special abilities, and strategies.

A.1 General Concepts

Strategy: A strategy in an RTS game can be considered as a number of general guidelines for how the player is going to play the game. This includes the number of different units and buildings to build as well as which research upgrades to purchase. It may also include specific tactics dictating how to carry out a certain part of the strategy. Finally, a strategy may also contain information about strong and weak points during the course of the game using the strategy. A strategy can often be considered of one of the types mentioned in Section A.5.

Tactic: A tactic consists of rules dictating how units should be controlled during a battle. The includes rules for formations, focus fire, unit preserving, how to use support units (including spells/abilities) and how to use the terrain on the map.

Technology Tree: Most RTS games have a certain order in which the different buildings and units can be build. For instance, a certain building may not be build, before another building has been build or a certain technology has been researched. This building, unit and research dependency is called the technology tree. The technology tree can be divided into a few major levels (in the tree depth), where any further advancement is depending on a single upgrade or research. These levels are called the technology levels or tiers. In *Age of Empires* these tiers are the different ages advanced through an upgrade at the town center, and for most factions in the Craft series[1] these are the three

[1] The Warcraft and Starcraft series

tiers advanced through the upgrade of the main building.

Build Order: At the start of most RTS games the player will start out with her main building and a few workers. In order to build any other units than workers, she will have to build unit production facilities, support buildings, and perhaps some other buildings, depending on which unit she wants to get. The order, in which she builds (may it be units or buildings), is called the *build order*. An *optimal build order* is one in which the goal is reached as fast as possible with the least amount of waste (time or resources).

Fog of War: At the heart of any RTS game is the fact that the player does not know what the opponent is doing. The idea is that you should only know what is happening in proximity of your buildings and within the sight of your units. The rest of the map is unknown. This concept is called Fog of War. Fog of War comes in two layers: Territory that is still unexplored and territory that is known, but not in sight.

Faction: Since the beginning of RTS games there has always been different factions, houses, races etc. In the single player campaigns these factions usually fight an epic war for the domination of the world or universe. Mostly these factions are unique though still with some similarities to the other factions in the game. In the Age of Empires genre the difference is mostly that one faction may have some units available that the other factions do not, while in the Craft series this difference is a little more pronounce. Starcraft is the first game with three totally different races. Protoss, Terran and the Zerg are so widely different that each demands a different play style.

Resource Node: Resource nodes are found in many different sizes and shapes depending on the game in question. The nodes in the Age of Empires genre are spread all over the map. The player may for instance gather wood from the forests, she may hunt wild game for food and mine various minerals from gold, stone or iron quarries. In Warcraft only two different resources are available, being gold and lumber. Lumber can be harvested from the forests and gold mined at the few gold mines. Starcraft is a bit different, because in this game the resources are found in clusters. This means that the minerals that are the main resource in this game are all found at a few spots on the map, but with several nodes close to each other. Mostly there will also be a gas vein at these sites. On many of the maps in the Craft series, the player will find either a gold mine, or a cluster of minerals and a gas vein close to her starting spot. This site is called the natural expansion site.

Attack Move: This is a command that dictates that the unit(s) should move to the location indicated by the cursor and attack any enemy object on the way.

Choke Point: A choke point is a narrow spot in the terrain, or perhaps between buildings in a base. The danger of moving through choke points is that only few units can move through at a time and if the enemy is waiting on the other side, she will only have to fight those few units at the time.

High Ground: When a battle is fought from high ground, it simply means that the player is at a elevated position on the map compared to the enemy. This often results in a certain advantage for the player, as the enemy will often deal less damage, because of fighting uphill. The advantage varies from game to game.

Upkeep: Upkeep usually describes a fee the player will have to pay in order to maintain her army. In Warcraft III upkeep is more like a penalty on the amount of gold harvested (not unlike a tax). If the player has more than a certain amount of units she will receive a penalty on the amount of resources received every time a worker brings in a sack of gold.

Focus Fire: Focus fire is a concept of extreme importance in most RTS games. It is simply a matter of having a number of units focusing their attack on a single enemy unit until this unit is killed. This is a lot better than having units firing at enemy units at random. Although the damage dealt is the same, it is a lot better to face five enemy units at full health than ten enemy units at 50% health, because the ten units would deal twice as much damage as the five units.

Hitpoints Hitpoints are a number of points units and buildings have describing their current state of health. When taking damage the number of hitpoints will decrease and when repaired or healed it will increase. The number of hitpoints can in most cases not exceed a certain maximum defined for each type of unit or building. If the number of hitpoints reach 0 the unit or building will be destroyed.

A.2 Buildings

Main Building: This building is usually at the root of the technology tree. In most games the player will start with a main building and a few workers only. She will be able to produce more workers at the main building and bring gathered resources to the main building in order to add them to her resource pool. As previously mentioned, the main building is also mostly the building, in which the player gains access to the next technology level. If lost the player will be unable to build any building or unit that required the achieved technology level. However, any building or technology that was built or researched before the main building was lost, is kept.

Unit Production Facility: The main building is as such a unit production facility but when this term is used throughout the rest of the report, it refers to buildings that may produce offensive or support units, basically anything else than workers.

Research Facility: In most RTS games the player will have access to one or several research facilities. These buildings have the soul purpose of upgrading units or make new technology available. They usually are not of much use after the technology or upgrades have been researched, unless they are an active part of the technology tree, meaning that this building has to be present in order to gain access to some units or other buildings.

Supply Structure: In many games the player has to provide some kind of control or food in order to support her army. Mostly this is done by building supply or something equivalent. In Warcraft III for instance the farms will allow units worth eight support points to be build (four footmen or eight workers).

Defensive Building: In all the games analysed the player has been able to build some kind of defensive building that will attack any enemy object. In most cases these defensive buildings are in the form of a tower. The soviet army in Red Alert has the tesla coils, Protoss from Starcraft has the photon cannon etc.

Expansion: In order to increase her income the player may decide to start gathering resources from several different resource nodes. In Warcraft, for instance, this would mean that the player would gather gold from several gold mines. To prevent the workers from walking all the way from the new mine (potentially a long distance away from the base), she builds a new main building close to the new gold mine. She may also add a few defensive buildings to protect this from harm. This is called an expansion.

A.3 Units

Melee: This unit is the close combat unit. It has a very limited range of its attack. In most cases it will have to stand right next to its target. Melee units are generally a bit hardier than other units, as they will have to get into the hottest spots of a battle. They may even just be used as a meat shield for the ranged and supporting units. Melee units with a special high amount of hitpoints are often referred to at 'Tanker' units.

Ranged: Ranged units tend to be a bit more frail than the melee units. In combination with melee units the player is able to deal more damage, than had she had melee units only (there is only enough space for a limited amount of melee units at the front-lines of the battle). In most cases the ranged units are also one of the only units able to attack air units.

Support: Support units are mostly specialised units that have some abilities that either strengthens the player's army or weakens the enemy's army. They may also have some limited ranged or melee attack, but it is not their primary function.

Siege: The player may be able to build units that act as artillery. These units have a slow rate of fire, but exceptional damage and range. Furthermore, they also tend to deal damage with an Area of Effect. In most cases these units are a bit fragile, but as they have superior range the player may protect them behind her army. In Warcraft, for instance, siege units also receive a bonus when attacking buildings.

Worker: In most games the worker is the most vital unit. This unit either builds various buildings, harvests resources or both.

Air: Flying units are available in most games. They are generally quite fast but fragile.

Hero: Early RTS games introduced the concept of a hero unit in the single player campaigns. Usually the story was build on the adventures of this hero unit, which had exceptional powers and got even better as the story progressed (but so too did the enemy). Warcraft III, for instance, has taken this concept to the next step and integrated the hero concept fully into the game. This means that the player and the AI are able to hire one or more heroes in any game.

Summoned: In some games support units, or perhaps a hero unit, may be able to summon creatures. These creatures will then serve the summoner, often for a limited period of time.

A.4 Special Abilities

Area of Effect: Spells come in many different sizes and shapes. In Age of Empires, the prophets are able to cast a spell that starts an earthquake, which affects an area on the map and damages all buildings in the area. This is called an Area of Effect spell or just AoE spell.

Buff: A buff is a positive spell that is cast on a friendly unit. In Warcraft an example of this could be the bloodlust spell that raises the attack speed of the affected unit.

Debuff: Contrary to the buff, the debuff is a negative spell cast on a unfriendly unit. In Starcraft this could be the optic flare spell that lowers the range of vision of any unfriendly unit.

A.5 Strategies

Scouting: In order to find out what the enemy is up to the player will have to send a unit to the area she wants to know about. This is called to scout the area. She might have to do this frequently throughout the game, as this knowledge may give her an advantage over her enemy.

Rush: The player may decide to try to surprise the enemy by attacking very early in the game. She can do this by building an early unit production facility and make a lot of cheap attack units. This is called to rush the enemy.

Tower: The defensive building introduced in the previous sections do not necessarily have to be used defensively. Offensive towers can be built just outside the enemy base, or perhaps even within the base, if she is busy elsewhere. This is risky as the towers are defenceless while being built, and as they are buildings they cannot be moved, if the enemy starts applying siege units.

Fast Tech: The opposite of rush is to fast tech (short for quickly climbing the technology tree). By skipping all the basic units, the player may try to climb the technology tree as fast as possible in order to reach some better units. This will leave her weak, while she is teching, but if successful, she will be a lot stronger than the enemy, if she "wasted" resources on the weaker basic units.

Mass: To mass is trying to kill the enemy by brute force. If the player has a better income, she may try to swarm over the enemy by building a lot of unit production facilities, and pump out units.

Harass: The player may decide to send out units to the enemy base or perhaps her expansion with the purpose of slowing down the resource gathering, kill off unprotected buildings and otherwise do harm, while the enemy army is away. By doing this hit and run tactic, the player can slow down the enemy production and force the enemy to pay attention to the harassment. This leaves herself free to pursue other matters.

Appendix B

Module Design

The following chapter will present a detailed design of modules in the framework. Each module will be presented in turn. The design description will start out by showing the internal structure of the module. Then the responsibilities of the module will be listed. Finally each sub-module will be presented in the same fashion.

B.1 Percept Interpreter

This module extracts information from the game state, and updates internal knowledge bases in the framework. It must be implemented by the AI designer to obtain the following information:

Reactive Module: Current hitpoints for all units and buildings.

Reactive Module: Native AI events.

In-Game Own Knowledge: Own unit and building positions and important attributes for these.

Game State Interface: Map terrain information.

Current Strategy Node: Units, buildings, research, expansions, resources, and current position in tech tree.

In-Game Enemy Knowledge: Enemy unit and building positions as well as important attributes for these. Furthermore, last known position of vanished units are noted here.

Dynamic Map Knowledge: Resource locations and amounts. (AI designer specifies game or map specific objects)

Assigned Unit Action: All friendly units and their currently assigned action.

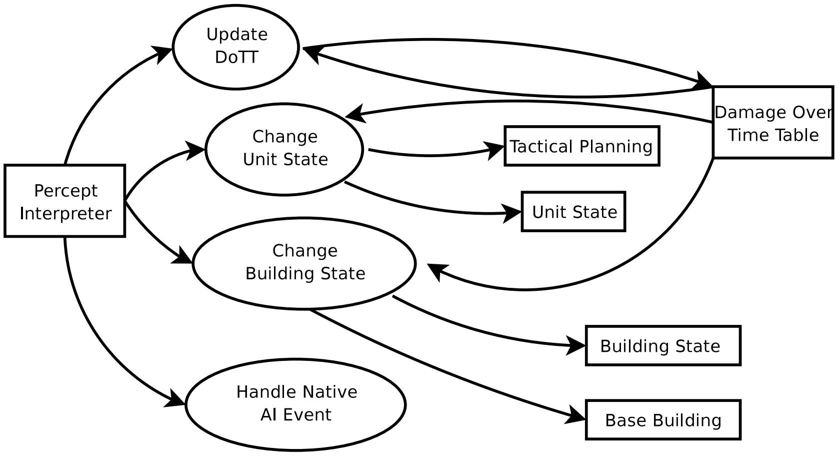


Figure B.1: Strategy Tree for the Example

B.2 Reactive Module

The purpose of the *Reactive Module* is to monitor and react on high DPS or change of building/unit states as well as handling native AI events. The structure of this module can be seen in Figure B.1

B.2.1 Responsibilities

Monitor DPS: Units and buildings that are damaged will be added to the *Damage Over Time Table* so that their condition and the damage they receive over time can be monitored

High DPS warning: When a unit or a building is exposed to a DPS exceeding a certain predefined value, the module that has to handle it must be advised and action must be taken

Change Building and Unit states: The amount of hitpoints the various buildings and units have define their state. If the amount of hit point change so that the building or unit changes its state it might mean that the unit or building should be handled differently than previously

Handle Native AI: In order to override the built-in native AI and replace it with better decisions all native AI events must be handed to the *Tactical Planning*.

B.2.2 Structure Overview

To meet the specifications of this module the structure has as such been split into three parts: One dealing with updating the *Damage Over Time Table* (*Update DotT*), another monitoring building and unit states and DPS(*Change Building State* and *Change Unit State*), and finally the third part handling the native AI events(*Handle Native AI event*). The modules do not interact but rather handle each their sub-task.

B.2.3 Update DotT

This module will work on the *Damage over time Table* (DotT). Whenever a unit or a building owned by the AI changes its amount of hitpoints, it will be monitored by the *Damage over time Table*.

In games like Starcraft and Age of Empires it does not really pay to try to remove units from the line of fire as the units have a relatively low amount of hitpoints. This means that this module could altogether be ignored. In other games like Warcraft the units have a higher amount of hitpoints and it is possible to heal these so in this case it makes a lot more sense to preserve them.

Responsibilities

Update Damage Over Time Table: Each unit that recently has had a change in its amount of hitpoints will have a list in the *Damage over time Table*. If no such exists, the list will have to be added. This module will record the current amount of hitpoints of each unit in the table at a predefined interval.

Damage Over Time Table Maintenance: When a list has not changed for a while - the unit has not changed its amount of hit points for a while, the list for this unit must be removed from the table.

Hot Spots

Update Interval: The user of the framework will have to define how often the amount of hitpoints should be updated

List Spaces: The lists in the *Damage over time Table* are FIFO lists of a user defined size. The size will have to fit with the update interval and the amount of time that DPS will be monitored.

Standby Time: The user will also have to define how long a list should be maintained if the hit point value does not change.

Standard Implementation

All operations on the table will be handled by default in the framework, as will the maintenance of the table.

B.2.4 Change Building State

Change Building State is responsible for monitoring the building state of all buildings. Given a rule set the *Change Building State* will for each building check which category the current amount of hit point is in. If the new amount of hitpoints means that the building will change into a different state it will notify the *Base Building* module and set the right state in the in game knowledge base: *Building State*. Furthermore this module will also calculate the DPS done to each building when these are getting damaged. It does this by adding the collected data and divide it by the number of data multiplied with the period of time the data was collected. To get an idea as to how serious this damage is, the resulting number could be divided by the maximum number of hitpoints the building has. This will yield the percentage of building health lost per second.

Most games feature the possibility to repair buildings, the way it is done, however, varies from game to game. In Command and Conquer the repair is done by the player - it is an ability that she can activate on buildings. This means that buildings are repairing at a constant speed whereas buildings that are repaired by workers as seen in Warcraft and Age of Empires are repaired by a speed defined by the number of workers repairing it.

Responsibilities

Change Building State: If a building has a change of hit points that means the building will change state, this module must set the right state in the *Building State* knowledge base and notify the *Base Building* module.

Calculate Damage Per Second: Given the lists in the *Damage over time Table* the module will have to calculate the damage the monitored buildings have received per second during the monitored time.

Issue Damage Per Second Warning: When the calculated damage per second exceeds a certain amount the *Base Building* module must be warned.

Hot spots

Building State Rule Set: This rule set will define the intervals of the different states.

Standard Implementation

The standard implementation will handle all the responsibilities of the module by default.

B.2.5 Change Unit State

Essentially this module has the same responsibility as the *Change Building State* module but unlike that module this will monitor the unit state and DPS done to units. In some games this module should be empty as the units are not worth saving or unit health is less important that the management of resources and production.

As mentioned earlier units in different games have different 'values'. In Starcraft they will in most cases be sacrificed in order to get the job done while it is imperative to preserve units in Warcraft. Furthermore the unit's role also plays an important part in defining when a unit should be removed from the line of fire. 'Tanker' units will have to be at a relatively low amount of hitpoints while support units in most cases have to be removed as soon as they are dealt damage.

Responsibilities

Change Unit State: A change in hitpoints that means that the unit will enter a different state must be handled by setting the right state in the *Unit State* module and notify *Tactical Planning* so that the new unit state can be taken into account.

Calculate Damage Per Second: Given the lists in the *Damage over time Table* the module will have to calculate the damage the monitored buildings have received per second during the monitored time.

Issue Damage Per Second Warning: If the calculated damage per second exceeds a certain amount the *Tactical Planning* has to be notified.

Hot Spots

Unit State Rule Set: This rule set will also define the intervals of the different states.

Standard Implementation

This module will be implemented by default.

B.2.6 Handle Native AI Event

Every time a unit that is not currently controlled by either the *Resource Management* module, the *Base Building* module or the *Tactical Planning* module is done an action upon, a normal game would handle this by some reactive action. This is, for instance, the case when a human player attacks an AI controlled unit that is standing alone. In most cases this enemy unit would follow and attack the human controlled unit even though it means to engage the entire enemy army. This and many other unfortunate events can be handled by simply passing the information to the *Tactical Planning* so that a well-considered action can be ordered. In order to do this the *Handle Native AI Event* module will have to receive all such events and the reactive part of the AI will have to be disabled. The event will have to contain the type of event and the unit/building in question.

In many games it is an unfortunate fact that the player can lure parts of the enemy army away from the rest by shooting on one of the enemy units and run away. The enemy unit that has been hit will then run after the one that shot it and perhaps pull part of the enemy army with it. In order to avoid this all reactive decisions must be disabled and handled by the proper modules.

Responsibilities

Override Native AI: All events that previously were handled by native AI will now be sent to the *Handle Native AI event* module.

Redirect Native AI events: Depending on the type of the native AI event it will be redirected to either *Base Building* or *Tactical Planning*.

Hot Spots

Event Groups: The user of the framework will have to define which buildings and units that will potentially receive Native AI events.

Standard Implementation

The only event the standard implementation will handle is the "under attack" warning. This warning will cause the module to warn *Tactical Planning* or *Base Building* depending on whether it is a unit or building that is under attack.

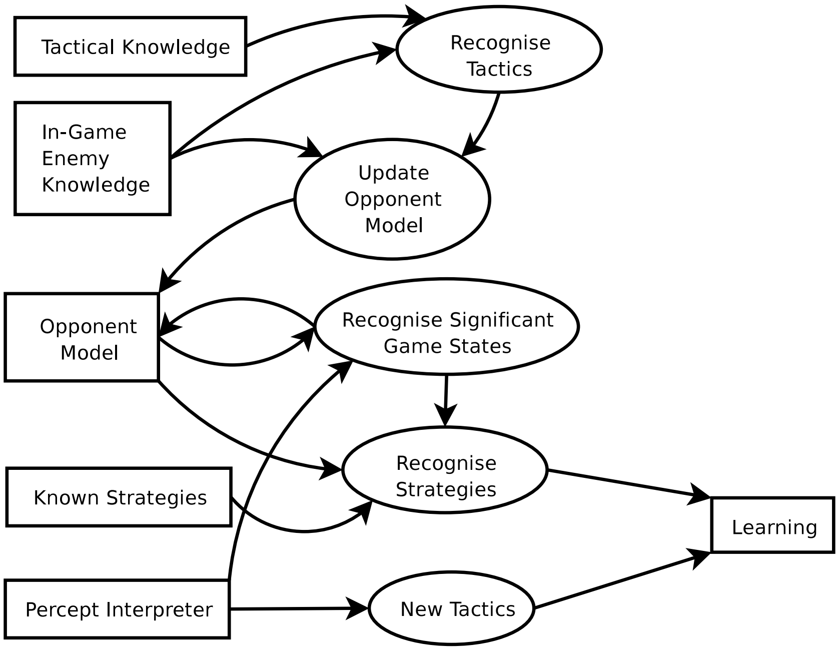


Figure B.2: The internal architecture of the Pattern Recognition module

B.3 Pattern Recognition

This module is responsible for recognising different strategies and tactics used by the enemy, and for keeping track of an enemy's strategic decisions throughout a game. The internal architecture of this module can be seen on Figure B.2. Circles represents sub-modules and boxes represents other modules or knowledge bases. Arrows indicate how they influence each other.

B.3.1 Responsibilities

Recognise Tactics: The module is responsible for recognising tactics used by the opponent during the game.

Recognise Strategies: Based on enemy unit and building composition, as well as enemy unit movement and tactics used, the module is responsible for recognising strategies.

Update Opponent Model: During the game, the opponent's unit and building composition will change as well as several other strategic important variables and it is the responsibility of this module to keep track of these and thereby keep an updated opponent model at all times.

Monitor Strategic Choices of Opponent: The opponent will make several crucial strategic choices during a game, which will influence the strategic possibilities open to her at a later stage in the game. This module will be required to keep track of these decisions.

B.3.2 Structure Overview

This module is divided into sub-modules based on the four different areas of responsibility defined above. The functionality can basically be divided into two parts: One branch for handling updating the *Opponent Model* and one for providing the *Learning* module with the necessary information used to learn new things. Updating the *Opponent Model* consists of two steps. First, the module will attempt to recognise tactics used by the opponent and this will be used when actually updating the *Opponent Model* with the information currently known about the enemy including the number of different units, buildings and research upgrades. The second part of the module ensures that all strategic choices made by the opponent during the game can be monitored. This is ensured by a module that recognises significant game states, to determine when the opponent makes a significant switch in strategy, which should be reflected in a strategy tree built for each opponent during the game. This strategy tree is considered a part of the *Opponent model*, and can be used to both, more clearly determine the opponent's strategy, and for easier determining when the opponent is doing a strategy the AI have not seen before. In the latter case, the *Learning* module can take advantage of the strategy tree built, and easily add it to the strategy tree representing all strategies currently known by the AI. The last sub-module is a module that ensures the possibility of adding game specific recognising methods.

B.3.3 Recognise Significant Game States

This module is responsible for recognising important game states in the opponent's strategy. With this information it will be possible to build a strategy tree during the game for the opponent's strategy, which in turn will make it possible to learn the opponent's strategy. When a new significant game state has occurred, which means the strategy tree for the opponent has been updated, the module that recognises strategies should be activated to determine if this is a new strategy or not.

Important game states are game specific and this module will therefore be very dependent of hooks. In a game like Warcraft III for instance, the first significant game state is when the player has created her hero and is ready to either attack the NPCs placed around the map or harass the enemy. In Starcraft on the other hand, the first significant game state can vary greatly from being an extremely fast attack on the enemy to expanding maybe two times before engaging the enemy.

Responsibilities

Recognise Significant Game State: Any significant game state occurring during the game must be recognised so that the opponent's strategy tree can be updated.

Hot Spots

Classification of Significant States: The user of the framework would in most cases be required to classify when a significant game state has occurred, so that a proper strategy tree for the enemy can be built, which allows for learning new strategies in a sensible way.

Standard Implementation

It may be possible to make a game independent algorithm, which for instance classifies important states as when the AI's army is either attacking or being under attack, but it would in most games not be enough to really classify a game's significant states. The problem with just using a standard implementation is that the strategy trees built for a game using only the standard implementation, will often not provide a very accurate picture of a particular strategy and this will reflect negatively in several other framework modules. However, a standard implementation should be provided for at least demonstrating how to specify significant states.

B.3.4 Recognise Strategies

This module is responsible for recognising the opponent's strategy in the strategy tree from *Known Strategies*, and if it is unknown, inform the *Learning* module about this. This is an operation on strategy trees and since these do not vary from game to game, this module can be left unspecified by the user of the framework.

Although strategies do vary from game to game, the methods for recognising strategies in strategy trees do not. Strategy trees are created specifically to deal with each particular game, and contain all possible information related to a strategy in that game. A strategy in a strategy tree is defined as a number of strategic important elements (like expansions, research etc.) as well as unit and building compositions, and hence it does not matter whether the game is Starcraft or Age of Empires.

Responsibilities

Recognise Strategies: The module must recognise the strategy the opponent has been doing throughout the game and determine whether it has seen this kind of strategy before.

Hot Spots

This module has no hot spots as everything is handled by operations on strategy trees.

Standard Implementation

Everything in this module is handled by the framework.

B.3.5 Recognise Tactics

The purpose of this module is to recognise tactics used by the opponent during the game. This knowledge is both used to update the *Opponent Model* with the current tactics being used and the opponent game tree, which will allow for the game tree to note at which point in the current strategy different tactics have been used.

The methods used to recognise tactics should in most cases be hook methods, because all game allow for very different types of tactics, and it is furthermore very different how much effect a certain tactic has from game to game. In Warcraft III for instance, a tactic could be to harass the enemy base with some units while levelling the AI's hero by attacking NPC's at the same time. This is a unique tactic for that particular game and it would make no sense in most other games.

Responsibilities

Recognise Tactics: The module must be able to recognise tactics used at any point during the game, and inform the *Update Opponent Model* sub-module about its results.

Hot Spots

Recognise Tactics Methods: The user of the framework must specify how the AI is to recognise a certain tactic used by the opponent.

Standard Implementation

Some tactics can be used in several different RTS games. An example could be the tactic of splitting up the army and attacking several resource gathering locations hold by the enemy at once and then kill workers there. Tactics like that are viable in almost any RTS game. Recognising the tactic should in theory be easy regardless of the game, and should be provided as a standard implementation. This is, however, only the case with tactics included with the framework. New game specific tactics must have recognising methods provided along with them for this module to work to its full potential.

B.3.6 Update Opponent Model

This module is responsible for controlling all updates of the *Opponent Model*. The reason this cannot be done directly, is because the AI cannot just add every unit it sees to the *Opponent Model*. Enemy units may disappear into fog of war and return again the next second and it is the task of this module to control that the same units are not added once again to the unit count of a particular unit type. Furthermore, as the *Opponent Model* must always reflect the current situation, the module must control when attributes expire in the *Opponent Model* and also notify other modules when a seemingly significant change has occurred.

When an attribute of the *Opponent model* should be considered outdated, depends of both the game and attribute in question. If the opponent for instance have used an air harass tactic earlier in the game, at what point should the AI realise that this is not what the opponent is trying to do anymore? Another factor that is game specific is determining when a significant change has happened in the *Opponent Model*. This of course also depends on the attribute in question. If the number of expansions attribute is changed, it would probably be a significant change, while an update consisting of the observation that the enemy now has five footmen instead of just four, would not.

Responsibilities

Update Attributes in Opponent Model: The module must keep an eye on all attributes of the *Opponent Model* and ensure that all new observations are properly reflected in the model. This includes keeping track of units seen earlier, which have left the vision of the AI player and then reentered.

Check Expiring Attributes: Because of the need for an updated *Opponent Model* at all times, the module must ensure that all attributes reflects the current situation. This means removing or reducing belief in attributes that has not been confirmed for a long period of time.

Check for Significant Updates: The module must after updating the *Opponent Model* check if the update is significant enough to be able to change the current belief of what the opponent is doing. If this is the case, it must activate the *Probabilistic Reasoning* module.

Hot Spots

Expiration Limits: The user of the framework should specify when attributes becomes outdated and how the belief of a certain attribute deteriorates over time.

Method for Determining Significant Updates: A method for determining when a significant update to the *Opponent Model* has been made is required to make sure the *Probabilistic Reasoning* module is activated at the appropriate times.

Standard Implementation

The standard implementation will have a predefined expiration date on attributes and a percentage change in attributes that will activate a signal that a significant change has occurred.

B.3.7 New Tactics

This module is basically just one hook module, which allow an AI designer to specify how the AI should recognise new tactics that should be learned by the the AI. It will process the data received from the *Percept Interpreter* module, and based on this information, decide whether the opponent has tried a new tactic not seen before.

In Warcraft III for instance, the undead race has a unit, the ghoul, which is intended to be both a harvesting unit and a light melee unit. In early versions, this caused a very special tactic to arise. When the ghouls were targeting trees, they had the ability to walk through other units (to avoid pathfinding problems in the base). This made it possible for the ghouls to walk right through the enemy army if they targeted a tree behind the army, which in turn made it possible to get behind the army and easily surround for instance a ranged enemy hero. This module is used to learn new and game specific tactics like the example tactic described here.

Responsibilities

Recognise new Tactics: The module must recognise new tactics used by the opponent.

Hot Spots

Methods for Recognising Tactics: The user must specify how the AI is to recognise new game specific tactics.

Standard Implementation

There is no standard implementation of this module, but it can be left unspecified, which results in the AI's inability to learn new tactics. It does

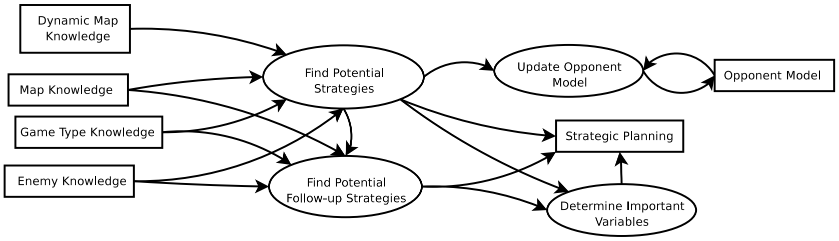


Figure B.3: The internal architecture of the Probabilistic Reasoning module

however, not limit the AI's capability to combine old tactics in new strategies.

B.4 Probabilistic Reasoning

This module will determine the most likely strategy used by the enemy, and determine what kind of strategies this could lead to in the future. Furthermore, it will specify what variables are important to watch, when determining the opponent's final choice of strategy. The internal architecture of the module can be seen in Figure B.3. Circles are internal sub-modules, boxes represent other modules or knowledge bases and arrows indicate how they influence each other. The following will describe the overall responsibility of this module and explain each sub-module in detail.

B.4.1 Responsibilities

Determine Most Likely Strategy: The module must, based on the current *Opponent Model*, determine the most likely strategy being done by the opponent.

Determine Most Likely Follow-up Strategy: Depending on the most likely strategies found, the module must determine the most likely follow-up strategies.

Determine Important Variables: Given a number of possible follow-up strategies, the module must determine important variables that will indicate the final choice among the possible strategies.

Update Opponent Model with new Beliefs: When a new most likely strategy has been found, the *Opponent Model* must be updated with new beliefs about attributes not currently known from the result of an observation.

B.4.2 Structure Overview

This module is divided into sub-modules based on the four different areas of responsibility defined above. The first thing the module has to do is determine the most likely strategy used by the opponent based on the current *Opponent Model*. This is basically a search through a strategy tree to find a matching node compared to the *Opponent Model*. Afterwards, two things must be done: The *Opponent Model* must be updated with new beliefs and potential follow-up strategies must be determined. For each follow-up strategy found, the probability for each must be calculated. Finally, important variables that determines the opponent's final choice of strategy must be found, so that appropriate scouting can be done.

B.4.3 Find Potential Strategies

This module is responsible for finding all potential strategies being done by the opponent based on the observations made about her so far and then find the probabilities for each possible strategy being used. The basic idea behind this module is to search through three strategy trees: *Game Type Knowledge*, *Map Knowledge* and *Enemy Knowledge*. Each provides a different aspect of the possible strategies the opponent may be doing.

The search through strategy trees do not vary from game to game, but the criteria for matching a node in the strategy tree to the *Opponent Model* do. From game to game, it changes how much two nodes in a strategy tree have to be different to represent different strategies and the strategic importance of certain attributes may also change.

Responsibilities

Find Potential Strategies: The module must find all potential strategies given the current *Opponent Model*.

Calculate Probabilities: Depending on the strategies found, the module must find the probability of each of them being the one currently used by the opponent.

Hot Spots

Maximum Node Deviation: The user of the framework must specify how much two strategy nodes should differ to be considered two different strategies. This includes defining the strategic importance of different attributes of the *Opponent Model*.

Standard Implementation

As a default implementation, the framework will provide a percentage match that must be fulfilled for two nodes to be considered the same.

B.4.4 Update Opponent Model

This module is responsible for updating the *Opponent Model* with new belief knowledge based on what kind of strategy the AI believes the opponent is currently doing. All attributes in the *Opponent Model* that are not currently based on real observations should be updated with what the AI currently believes about the opponent.

The reason that this process is defined as a sub-module in this architecture is that this allows for an AI designer to decide when an observation should be replaced by a belief. This could for instance be when an observation is several minutes old, and the attribute is known to change frequently. This varies from game to game.

Responsibilities

Update Opponent Model: The module must update the *Opponent Model* with new beliefs based on the most likely strategy used by the opponent.

Hot Spots

Updating Beliefs: The user of the framework must specify how to update the *Opponent Model* with beliefs.

Standard Implementation

The standard implementation should provide a simple approach to updating beliefs in the *Opponent Model*, replacing only those attributes who have never been observed.

B.4.5 Find Potential Follow-up Strategies

This module looks at all possible strategies being done by the opponent, and finds all potential follow-up strategies along with percentages of their likelihood of being used. The strategy tree has direct support for this operation, by simply looking further in the tree from each potential strategy node.

How many follow-up strategies to consider should be based on the particular game in question. It depends a lot on the strategy tree in question and how each particular game's strategies are reflected in the strategy nodes.

Responsibilities

Find Potential Follow-up Strategies: Given a number of possible current strategies, the module must determine the most likely follow-up strategies.

Hot Spots

Considered Follow-up Strategies: The user of the framework should be able to specify how far ahead in time the AI should look to find potential follow-up strategies.

Standard Implementation

By default, the search through strategy trees should look a predefined number of nodes ahead when considering potential follow-up strategies.

B.4.6 Determine Important Variables

The responsibility of this module is to determine the currently unknown variables that are essential for choosing among the most likely strategies the opponent is doing. The module will consider only the most likely strategies, and determine variables that are differing and essential for the opponent's choice among them. This will later help the *Strategic Planning* module to scout the right things, which are more likely to reveal the opponent's final choice of strategy.

This module should be independent of the game in question, because finding the variables that differ in the potential follow-up strategies have nothing to do with the actual game being played.

Responsibilities

Determine Important Variables: The module must specify the variables that should be investigated further, because of them being important in regards to the opponent's final choice of strategy.

Hot Spots

There are no hot spots in this module, as it is all handled by the framework independent of the game in question.

Standard Implementation

N/A

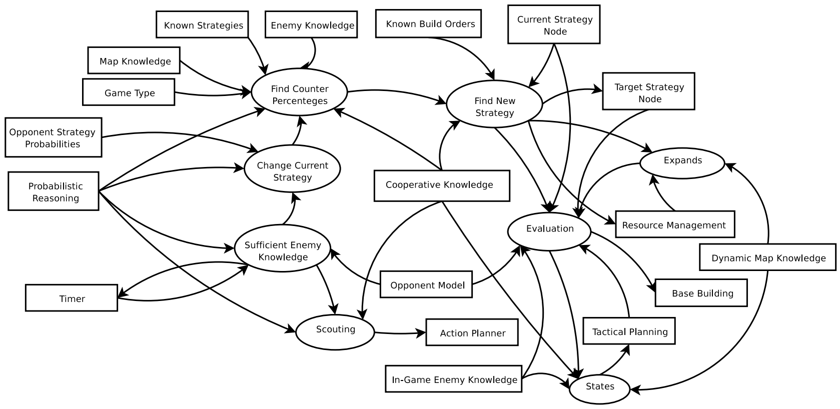


Figure B.4: Internal architecture of the Strategic Planning module

B.5 Strategic Planning

This module will handle all strategic decisions. This includes determining when the AI has enough information to choose a good strategy and of course actually choosing a strategy. The choice of strategy should depend heavily on what counters the opponent's strategy, but also the current state of the AI. The module is furthermore also responsible for decisions about exactly where the AI's army should be and if it should split up etc. The internal architecture of the module can be seen in Figure B.4. Circles in the figure represents internal sub-modules and boxes represents other modules or knowledge bases. The following will first discuss the overall responsibilities of the *Strategic Planning* module, and then present each of the sub-modules in the internal architecture along with a discussion of how the sub-module is to complete its task.

B.5.1 Responsibilities

Determine if the AI posses Sufficient Knowledge: The module must determine whether the AI has enough knowledge about the enemy to choose a good strategy that counters the enemy.

Determine Scouting Missions: If there is insufficient enemy knowledge or if all data in the *Opponent Model* is outdated, the module must assign one or more units a scouting mission, telling it exactly where to go and what to scout for.

Find New Strategy: The module must determine if there is a need for a new strategy, and if so, find the best possible strategy suiting the current situation.

Execute Strategy: Finally, the module is responsible for dictating where on the map all army units should be during the execution of the chosen strategy.

B.5.2 Structure Overview

This module consists of several distinct parts. When the module is activated by a significant update to the *Opponent Model* or by a timer, the first thing that is done is checking whether there is sufficient enemy knowledge to decide on a good strategy. If there is not, the *Scouting* sub-module is activated and one or more units are put on a scouting mission. Either way, it must be decided whether or not to change strategy. This is done by determining if the foundation on which the last strategy was decided has changed. If it has changed, two things are done. First, probabilities for different strategies countering the opponent's strategy is calculated, and partly based on this, a new strategy is selected. The new strategy is represented as a strategy node and hence this is not enough to determine the actions taking by the AI's units. For this, an *Evaluation* sub-module determines the current situation of the AI, and places it in an appropriate state. This state will divide units into groups and give orders dependent on the current situation of the AI.

B.5.3 Sufficient Enemy Knowledge

This module is responsible for determining whether the AI has enough information to decide upon a good strategy. This can be determined by looking at the *Opponent Model*, and at what the AI designer has defined as being enough information. In some cases the framework could override the hook specified by the AI designer, if for instance a certain attribute is vital for knowing which strategy the opponent is going for, and hence should be scouted. If there is insufficient information, the *Scouting* sub-module is activated and provided with one or more variables that are to be scouted. This module will always trigger the *Change Current Strategy* sub-module, as even though enough information is not present, the AI must still pick a strategy according to its best guess of what the opponent is doing.

Defining what qualifies as being enough information is game specific. All attributes may have very different importance in relation to countering the opponent's strategy. In Warcraft III for instance, it is a huge factor what kind of buildings the opponent has in tier two and tier three, while this is far less important in games like Command & Conquer. In Command & Conquer it is far more important what kind of units the enemy has and how many, compared to Warcraft III where the technology branch pursued by the enemy is far more important for recognising her strategy.

Responsibilities

Sufficient Information: The module must decide whether the AI has enough information about the enemy to choose a good strategy.

Hot Spots

Enough Information Criteria: The user of the framework must specify when the AI has enough information, and thereby basically decide scouting frequency.

Standard Implementation

The standard implementation will assume that all attributes of the *Opponent Model* are equally important. This means that the standard implementation can simply keep a predefined percentage of how much an attribute may deviate from the most likely strategies found before a scouting mission should be determined. Furthermore, the framework can specify a time limit that basically decides the scouting frequency of an attribute.

B.5.4 Scouting

This module is responsible for selecting a unit to scout and determining what specifically that unit is to scout. Selecting what to scout should be a decision based on the input from the *Probabilistic Reasoning* module, which determines the currently most interesting unknown variables. How to obtain this information can in part be specified by the framework (buildings are in the enemy base, units are near the enemy army etc.), while in special cases the AI designer should decide how to obtain it.

In some games or strategies, the player may want to scout for very specific things. In Starcraft for instance, one may want to have a Zerg Overlord patrolling between a Terran's main base and an island to be able to scout if the enemy decides to fly a Control Center to the island to create an expansion. Gaining this information in time would make it possible to attack the Control Center before it gets to its expansion site.

Responsibilities

Selecting Scouting Unit: This module must select the best unit(s) to scout with depending on the scouting mission.

Determine Scouting Target: Depending on what the AI wants to know more about, the module must determine where to find this information and then scout to obtain it.

Hot Spots

Unit Scouting Ability: The user of the framework should select which units in a particular game should be preferred as scouting units.

Scouting Locations: The user of the framework should define where to find certain information.

Standard Implementation

The standard implementation could choose either the fastest or cheapest unit to scout, and always send the scout towards the enemy base, unless it has a good idea of where the enemy army is, and is scouting for some unit attribute. If it is looking for expansions, it could simply start scouting the nearest expansion possibility (compared to the enemy main base) and then work through all expansion possibilities in that order. When playing on a randomly generated map, the AI must also be able to scout the map and not just the enemy. To scout the map, the standard implementation could use influence maps to determine unexplored areas of the map.

B.5.5 Change Current Strategy

This module is responsible for deciding whether a change in strategy should be considered. The AI should basically only consider changing its strategy if it has some new information, which can lead to a new and better strategy. This means that the primary task of this module is to test whether the information, which were used in choosing the last strategy, has changed in such a degree that a new strategy should be considered. If this is the case, the *Find Counter Percentages* sub-module is activated, and if not, the *Strategic Planning* module goes straight to the *Evaluation* sub-module explained later.

It is very game specific how often a strategy should be re-considered. In general, games with strong counters will require players to change their strategy very often, because so much depends on information about the enemy army. This means that games in the Craft series will require changing strategy often when the opponent model changes, while in games like Command & Conquer, the AI will be able to keep her current strategy more often, because counters have less effect.

Responsibilities

Consider New Strategy: The module must determine whether the foundation that the last strategy was built upon has changed and through this, decide whether a new strategy should be considered.

Hot Spots

Significant Changes: The user of the framework should be allowed to specify how much of a change (compared to last time a strategy was selected) is necessary for the AI to reconsider its strategy.

Standard Implementation

A standard implementation could simply reconsider its strategy every time the *Probabilistic Reasoning* module changes what it considers to be the most likely strategy done by the opponent.

B.5.6 Find Counter Percentages

The purpose of this module is first to find all possible counters to the possible strategies found in the *Probabilistic Reasoning* module, and then to find the probability for each counter being an effective counter to what the opponent could be doing. Each strategy done by the opponent may have several counters and each counter may have a different percentage attached it, representing how often this counter should be used compared to the others. First a joint probability between the probability of the strategy being used, and the probability of the counter being used should be computed. Then it should be examined if any of the counters are practical the same, and if thats the case, these percentages should be computed into another joint probability for each counter being successful. The result would be a probability for each distinct counter, the highest dictating the counter which is most likely to counter the enemy's strategy. This entire process should be done for both current and follow-up strategies.

All of this is handled by strategy trees or operations on them and is hence not game specific.

Responsibilities

Find Counter Percentages: This module is responsible for finding the percentage chance of a strategy countering the opponent's strategy.

Hot Spots

N/A

Standard Implementation

N/A

B.5.7 Find New Strategy

This module is responsible for selecting the target strategy for the other framework modules to try and accomplish. It uses the counters and their percentage chance of countering produced by the *Find Counter Percentages* module and the *Current Strategy Node* to help make its decision. The module must make a trade-off between choosing the best possible counter and choosing a strategy that is not too far away from the current strategy node. In some cases (often the beginning of the game), the AI will to a certain degree ignore the counters and focus only on its own strategy. As an extra element, the strategic decision could also depend on knowledge of what the AI's allies are doing or whether it has a strong build order for a certain strategy.

The implementation of this module depends heavily on the game in question. Basically, the more counter oriented the game is, the more the AI should be willing to deviate from its current strategy. This means that in games like Warcraft III, the AI should often completely change its strategy, while in games like Command & Conquer, the AI should often not deviate too much from the original strategy.

Responsibilities

Find New Strategy: This module is responsible for finding a new strategy based on the information provided by the *Probabilistic Reasoning* module.

Hot Spots

Choice of Strategy: The user of the framework should specify how the AI should make the trade-off between countering the enemy and not changing strategy completely every time new information is received.

Standard Implementation

One fairly general mechanism for choosing the strategy could be implemented, but in most cases it would be so game specific that it is better left to the AI designer. A general approach could for instance be to let the AI counter as much as possible, but never let it deviate more than 50% from the current strategy.

B.5.8 Expands

It is the responsibility of the *Expands* module to test if the AI needs to take action before expanding to a certain location. This module is necessary for two reasons: Some RTS games place NPC characters around the map (often guarding expansions) and other times the enemy may be occupying a resource location. If either of these are the case, the AI needs to take action before an expansion is possible. This will often be in the form of an attack at the units or buildings occupying the resource location.

In Warcraft III, all gold mines are occupied by NPC units, and these must be removed before the AI can expand at a certain position. In a game like Starcraft however, there are no NPCs at all, but it is very common for players to leave a single unit at different expansion sites to simply remove the opportunity of the opponent to expand without the player noticing. This also results in the requirement of attacking this unit before an expansion is possible.

Responsibilities

Check Possible Expansions: This module must analyse expansion sites and determine if the AI's army need to take action before an expansion is possible. Furthermore, it must determine the army strength required to attack the enemy units at the expansion.

Hot Spots

Protected Expansions: As mentioned earlier, some games have expansions protected by units by default, and the user of the framework should define whether this is the case.

Army Comparison: The user of the framework must define how the AI is to compare two different armies to each other, which makes it possible to determine the army required to attack a certain expansion point.

Standard Implementation

As default, the framework will assume that all expansions are left unguarded, as is the case in most games. When trying to determine a sufficient army force, the AI can use a very simplified system of trying to have more or fewer but better units than the opponent.

B.5.9 Evaluation

This module is responsible for evaluating the current strategic situation for the AI. At this point the target strategy node has been decided, which the *Base Building*, *Resource Management* and *Action Planner* modules uses to follow the strategy. However, a strategy node does not say anything about where the army should be going and where it should be attacking. This is where the *Evaluation* sub-module comes in. Given the AI's army and the current game situation, this module evaluates in what state the AI should be in. It must take into account things such as army sizes, technology trees, income rates etc. Depending on the state, units will be dispatched in order to best accomplish the overall strategy.

When to switch from one state to another is very game specific. Imagine the situation where both armies are at each others base attacking the main building. In Warcraft III, players would have the option of using a town portal to get home and defend their base, while in Starcraft, and in most other games, the player would have to walk home. This would often result in that a Warcraft III player would switch to a defend state, while a Starcraft player would keep itself in an attack state. This of course always depends on the actual situation.

Responsibilities

Evaluate Current Situation: The module must evaluate the current situation, and place the AI in one of the available states.

Hot Spots

Evaluate Situation: The user of the framework must specify how the AI is to evaluate the situation, and which situations corresponds to which states.

Standard Implementation

As will be explained in the following section, the framework will by default include three states to choose among: Attack, Defend and Harass. A standard implementation could implement an evaluation method, which chooses between these three states in a relatively simple manner. The AI should be attacking if its army is larger than the opponent, it should defend if any important buildings are under attack and it should harass if it has chosen a strategy, which entail having a small number of units compared to the enemy in the beginning phase of the strategy. In the latter case, harassing the enemy would buy the AI time to successfully either tech to the wanted units or get an expansion up and running.

B.5.10 States

This module is responsible for executing whatever that state dictates the AI to do. Three game independent states will be provided with the standard implementation, but with the possibility of adding more depending on the game. The three standard states will be explained in the following.

Attack

The attack state must determine where to attack, and decide if it is necessary to split the army into several groups and thereby try to accomplish more than one objective at once. It basically goes through the steps specified below:

Find Possible Attack Positions: Analyses the map and the enemy to determine possible locations to attack. This could for instance be the enemy main base, an enemy expansion or the current position of the enemy army. The army strength needed to complete a successful attack is specified along with some form of desirability value of each attack target. How to calculate these values should be specified by the AI designer by hook methods.

Analyse Map Situation: Adjusts desirability values according to the current map situation. This includes analysing the position of all armies on the map, including the AI's own army. If for instance the attack desirability of two different locations are close to the same, but the AI's army is closer to one compared to the other, it should of course attack the closest target.

Coordinate Attacks: Depending on whether the AI's army is strong enough to carry out multiple attack orders, the army should be split up in a sensible way. It is game specific when it is reasonable to split up the army, and should hence be primarily specified as a hook. Splitting up the army could also be because the AI wants to perform some game specific tactic.

Assign Actions: The last task is to specify target map positions for each group and notify the *Tactical Planning* of this. It should also be considered here whether the army is already gathered, or if this has to be done before moving on to the attack location.

Defend

This state should describe the state where the AI is under pressure, probably outnumbered, and should simply try to protect itself until it reaches a stronger state. This could be for instance when the AI is teching, and is attacked by the enemy. Then it should only fight in its main base using base defence as well as terrain and position advantage (high ground, small passages etc.). The following will describe the reasoning the AI must go through to decide how to handle itself in this state.

Evaluate Situation: First of all, the AI must determine whether it is under an attack or not. The answer determines how the AI should handle defending itself.

Possible Attack Analysis: If the AI is not under attack, it must analyse the map to determine where the opponent is most likely to attack. This includes analysing its own weakest points, as well as trying to determine if the opponent knows about these. After determining this, it must send the order to move to the most likely attacked location, and ensure that the AI is in a good position for the potential enemy attack.

Defend Analysis: If the AI is under attack, it must determine whether the location, building or units are valuable enough to try and defend and if this is even possible (the AI may be far away from the position being attacked). It should also take into consideration whether it can even reach the position being attacked before everything is destroyed.

Position Analysis: If the location is valuable enough to be defended, the AI must move into position. This means analysing the right way to approach the enemy and to decide whether it must gather its army before moving in. In some games the decision can be even more complex, like for instance in Warcraft III, where it is possible to town portal back to the base to defend. In that case the AI must decide the position to town portal, which will bring the AI into an optimal battle situation.

Harass

This is the state, where the AI knows the opponent is trying to accomplish some strategy, and for whatever reason, wants to slow it down in doing so. Harassing could consist of a number of different things, like killing workers, destroying buildings that are being built, or harassing the main army of the enemy so that it cannot perform whatever it should be doing. When the AI decides how to harass, it must go through the following tasks.

Find Possible Targets: First all kinds of possible targets for harassing must be discovered. Harassment targets do not vary much from game to game, but the degree of how effective a certain harassment tactic is does. Possible harassment targets includes:

- Enemy workers
- Weak units (hit and run attacks on for instance support units)
- Hurt units (units at critical health)
- Buildings that are under construction
- Important buildings for the opponent's strategy
- Harvesting buildings (including main buildings)

Analyse found targets: After finding the different possible targets, the AI must examine each of them and determine how many and what kind of units are needed to successfully attack each different target. In some cases the degree of what determines a successful attack must also be evaluated (for instance how many workers should a harassment kill before it can be considered successful?). Finally, it must determine which of the possible targets will harm the enemy the most compared to the cost of executing the harassment. How to evaluate the different missions will in most cases be a game specific task.

Assign Units: Finally, the AI must determine which of the harassment targets are to be executed and which units are grouped together to execute a particular mission. Units not picked for any harassment mission must also be sent to some specified location (often the main base). All group specifications and target positions is then sent to the *Tactical Planning* module, which takes care of the actual execution of each harassment mission.

Responsibilities

Assign Groups and Unit orders: The module must, depending on the state, divide the army into groups and assign them an order to go to a position on the map. The actual execution of how to get there and what to do when they get there, is handled by the *Tactical Planning* module.

Hot Spots

States: In some games the three default states will not be enough and this is why the user of the framework should be allowed to add extra states depending on the game. This could be things like a Creeping state, a Push state etc.

Configure Standard States: Even though the framework provides three standard states that covers all kinds of RTS games, the user of the framework must configure these modules to suit the game in question best possible.

Standard Implementation

The standard implementation will in this case be the three default states provided with the framework and a standard configuration of these.

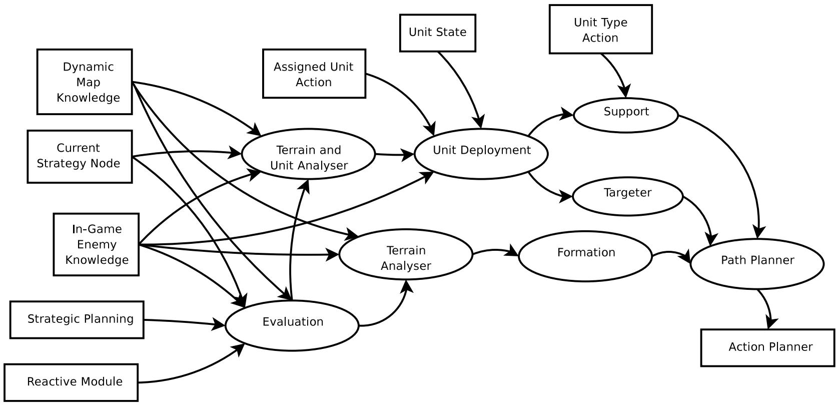


Figure B.5: The internal architecture of the Tactical Planning module

B.6 Tactical Planning

This module will handle all unit actions that is not directly associated with *Resource Management* or *Base Building*. The internal architecture of the module can be seen in Figure B.5. Circles are internal sub-modules, boxes represent other modules or knowledge bases and arrows indicate how they influence each other.

B.6.1 Responsibilities

Unit Actions: This module will carry out all unit actions that are not resource or base building activities.

B.6.2 Structure Overview

This module consists of two parts: Unit movement and unit engagement. These two parts have two sub-modules in common: The *Evaluation* module and the *Path Planner*. Any action must first pass through the *Evaluation* module before being carried out. This module will among other things decide whether the AI's forces are strong enough to engage in combat or if they should turn and flee. The *Path Planner* is not just a normal pathfinder but also takes other factors into account such as flow. The movement part first analyses the known terrain, then it finds a suitable formation for the units that are to move according to the collected terrain information. The engagement part first analyses terrain, units and buildings in the combat area. This information is then passed on to the *Unit Deployment* module that will find a suitable formation for the units available. It will also decide which units are assigned attacking roles and which are assigned supporting roles. These are then passed on to the *Support* module and the *Targeter*.

B.6.3 Evaluation

Evaluation is the first module within the *Tactical Planning* that is activated. The *Evaluation* module will first determine whether the *Tactical Planning* was triggered due to a movement order, a change in unit state or an engagement order. Movement orders will be passed on to the *Terrain Analyser* and so will a change in unit state trigger if this means that the unit in question will have to be withdrawn from battle. If this is not the case change in unit state events will be passed on to the *Terrain and Unit Analyser*. This is also the case with any engagement order if the *Evaluation* module decides that the battle is worth engaging. The *Evaluation* itself will be based on the amount of units, unit strength, strategy and position.

Different games require different ways of evaluating a situation. In Starcraft for instance a situation where a player is outnumbered does not necessarily mean that the player should retreat but perhaps rather kamikaze and do as much damage as possible before the army is beaten. In Warcraft the situation is quite different as units are more 'valuable' and should be saved as often as possible.

Responsibilities

Reroute Orders: All orders must be checked and decided whether they are movement orders or engagement orders.

Situation Evaluation: When facing the enemy this module must decide whether to fight or to flee.

Hot Spots

Evaluation Method: The user of the framework must define an evaluation method that analyses a given situation and decides whether or not to engage.

Standard Implementation

The standard implementation will compare the damage output and the total amount of hitpoints of the two armies and base its decision on this.

B.6.4 Terrain Analyser

This module will look at the terrain over which a unit or a group of units will move. Essentially it will transform this part of the map into an influence map that takes every little facet into account. This is everything from height variations in the terrain, to resource clusters and NPC units. This will result in a multilevel influence map that the *Formation* module can place the desired formation on and the *Path Planner* can move the unit/units through.

In Warcraft the map is so simple that there are no elevations or objects that units can hide behind. This means that no matter where the units stand they will receive full damage from ranged attacks. In Starcraft units standing above other units will receive a damage reduction when fired upon. This is just one of the different aspects the terrain analyser will have to handle from game to game.

Responsibilities

Translate Area Information: The product of this module is a spatial representation of the area that the unit/units must move through. This spatial representation must include all known information of any value to the task of moving through the area.

Hot Spots

Handling Area Types: The user must define all types of areas that must be accounted for in the analysis.

Standard Implementation

The standard implementation will only handle areas in which the units can move and areas in which they cannot.

B.6.5 Formation

The *Formation* module is responsible for ordering units in a predefined formation. It also has to account for critical areas in the terrain or rather the influence map that is received from the *Terrain Analyser*. This means that the *Formation* module may have to reorder the formation at the critical points such as choke points. All this can be done by first identifying the critical points and afterwards plan the formations that will be used between critical points and in the points themselves.

In Command and Conquer the formation used is not really that important. The only formation detail that is used is mostly keeping artillery at the back of the army. In Age of Empires, however, the formation is crucial. Tanker units can keep the enemy at bay while the lighter armoured units can deal a lot of damage.

Responsibilities

Identify Critical Spots: The module must be able to identify critical spots - spots that are potentially dangerous.

Draw Formation: Given the situation a suitable formation must the found.

Hot Spots

Identify Critical Spots: The user must define a method to identify critical spots. This heavily depends on the terrain and general map structure and is thus game specific.

Formations: Formations vary from game to game given the different units available in the games and their use. Therefore the user must define a set of formations and their use.

Standard Implementation

A few simple formations based on amount of hitpoints and armour will be implemented by default.

B.6.6 Terrain and Unit Analyser

This module will not only do the same tasks as its counterpart the *Terrain Analyser* but it will also take units and buildings into account. Furthermore it will also be able to work with believes of the whereabout and number of unseen enemy units if such exist. The product of this module is a multilayered influence map that takes all this into account.

In some games towers for instance are more of a nuisance than a real threat. The damage output of a tower cannot be used as the only factor to be taken into account when analysing the threat of a tower from game to game. A weak tower in Command and Conquer can be ignored while a weak tower in Warcraft III may have a side effect such as mana drain or a slowing effect that can have a serious impact on the outcome of a battle.

Responsibilities

Translate Area And Unit Information: All relevant information available must be translated into a usable spatial representation.

Hot Spots

Handling Terrain and Unit Types: The analysis that is handled by default will only be able to handle simple cases and in order to get good results the user will have to define rule sets through the tactics for all terrains, units and map specific objects.

Standard Implementation

In addition to the functionality found in the *Terrain Analyser* the standard implementation of this module will also account for units and buildings. It will look at damage output, amount of hitpoints and amount of armour.

B.6.7 Unit Deployment

Based on the information passed on by the *Terrain and Unit Analyser* as well as a number of knowledge bases this module will decide upon the position and assignments of friendly units during an engagement. Basically it does the same as the *Formation* module but takes the concept a step further by passing units on to the *Support* module and the *Targeter* module depending which assignment they have. The formation itself will be handled by a combination of the default unit behaviour and the strategy specific behaviour defined by the tactics in the current strategy node. Influence maps seems the obvious tool to handle much of this work.

As mentioned in the *Formation* module Command and Conquer does not require much consideration when dealing with positioning. In Warcraft III however light armoured units will die several times as fast as the heavily armoured units if attacked. As a lot of units can only attack in close combat the heavily armoured units will have to be between the enemy and the light armoured units.

Responsibilities

Unit Positioning: When engaging, tanking units will have to be placed at the front facing the enemy and lighter units in a secure distance from enemy units. Additionally support units have to either be well distributed among friendly units or within range of the target enemy units.

Unit Task Assignment: Units have to be assigned a task: to support or to attack. This has to fit with the deployment.

Hot Spots

Unit Deployment Plans: Given a strategy and the tactics defined for this strategy, the available units, the terrain and other map specific information the user has to define a method that deploys and assigns actions the best possible way.

Standard Implementation

By default the heavily armoured/high hit point units will be placed closer to the enemy than lighter armoured/low hit point units.

B.6.8 Support

The *Support* module is responsible for selecting the best skills and targets for the skills for all the units passed on to it. Depending on the chosen strategy support unit will be assigned different skills to use on different targets. This will be determined by the tactics stated in the current strategy node. If no such rules exist default behaviour will be assigned. The influence maps needed for this module depend on the available support skills.

Age of Empires has a fewer means of support available than games like Starcraft and Warcraft. Support in Age of Empires is much less important. In Starcraft the good use of support will be able to win almost any situation.

Responsibilities

Assign Support Actions: This module must assign the best possible actions to all available support units given the available information.

Hot Spots

Support Action Rule Set: The user must define a rule set that dictates how different support units should react in various situations. The tactics in strategy nodes can override this behaviour if a different behaviour is required in a specific strategy.

Standard Implementation

The standard implementation will distribute support according to the rule set defined in Unit Type Action.

B.6.9 Targeter

All the units passed on to the *Targeter* module will be assigned an enemy unit to attack. The *Targeter* will have to take counter focus, focusing strategically important unit and maximising damage (no excessive) into account. In order to do this the *Targeter* will have to know which unit counters which unit and use this information to assign targets. The *Targeter* will also have to consider which target are important to the success of the current strategy.

Contrary to many of the other games Warcraft III features a series of different armour types and attack types. Different armour types have bonuses and penalties when hit by different attack types. This means that in this case the targeter will have to take armour type - attack type match-ups into account when assigning targets, contrary to just focus firing.

Responsibilities

Assign Targets: The *Targeter* must assign targets to all available units in such a way that important enemy units are eliminated, the damage is maximised, and the current strategy is not compromised.

Hot spots

Target Priority Rule Set: A target priority rule set must be defined through tactics that lists all units and buildings prioritised in the order they should be targeted.

Counter Table: The user also has to define a table that describes the counter relations in the game.

Standard Implementation

By default the *Targeter* will only take the amount of hitpoints and the amount of armour into consideration when assigning targets.

B.6.10 Path Planner

The *Path Planner* is an advanced version of a normal path finder. The *Path Planner* has to find the fastest path (not necessarily the shortest) given a formation, flow, varying unit speed etc. and on top of this it will have to pass assigned unit actions on to the *Assigned Unit Action* knowledge base.

Responsibilities

Plan Best Path: Given formation, flow, and unit speed, find the best path for each unit that has to be handled.

Reroute Actions: For all units that pass through the *Tactical Planning* module, reroute their assigned actions to the *Assigned Unit Action* module.

Hot Spots

Hot Spot: N/A

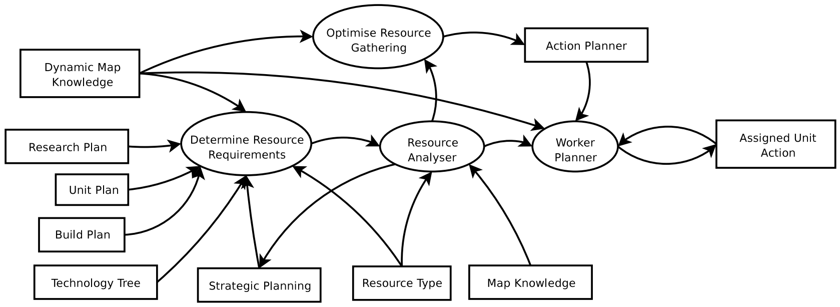


Figure B.6: Internal architecture of the Resource Manager

Standard Implementation

The entire *Path Planner* will be implemented by default and will act on the information produced by the *Terrain Analyser* and the *Terrain and Unit Analyser*.

B.7 Resource Manager

This module should make sure that there are resources enough for building units and buildings. The module is run when there have been assigned workers to it, which makes these workers start gathering resources. The type of resources gathered should fit the things that have to be constructed to follow the strategy. When there is a change in the strategy or if there is a shortage of resources, this module should be activated again. The module should however anticipate the best it can, what resources that will be required. It should also be run with some frequency to check that it is gathering resources in the most optimal way, and that there are no harvesters standing around not gathering, and if there is not enough workers, request that more workers are built.

The *Resource Manager* module's architecture can be seen in Figure B.6. Rectangles represent knowledge bases or other modules and circles are the sub-modules. The following present the responsibility of the *Resource Manager* module, and present each of the sub-modules, and discuss how the sub-modules complete their tasks.

B.7.1 Responsibilities

Determine Resource Requirements: The module most determine what resources that are necessary to reach the target strategy.

Analysing Resources: The module must find the best places to harvest resources.

Planning Worker Tasks: The module makes the worker go out and harvest resources, and come back and deposit them.

Optimise Gathering of Resources: Finally this module makes sure to optimise the gathering of resources.

B.7.2 Structure Overview

The module consist of four sub-modules. When a change have happened to the strategy, the new resource requirements are found by running the *Determine Resource Requirements* module. Then it is analysed where the best place to gather resources are. After this the *Planning Workers Tasks* module will make sure that the workers that have been assigned to the module are sent to gather those resources. When the workers have reached their goal this module will make the workers gather the resource and make sure that it gets back to the depot, and redo this cycle. Once in a while the *Optimise Gathering of Resources* module is run, to make sure that resource gathering is optimised.

B.7.3 Determine Resource Requirements

This sub-module figures out the anticipated resource needs, according to the *Build Plan*, *Unit Plan* and *Research Plan*. These plans contain the list of what is going to be built or researched within the next short time span. These plans are constructed from the *Target Strategy Node*, so indirectly the resources requirements are determined from this. Depending on what kind of resource these plans might require the most, the harvesting/production of this resource will be increased. Using the *Build*/*Unit*/*Research Plan* it can also account for what resources will be required in the near future. The *Strategic Planning* can tell this module to find a place to put an expansion. This is told to the *Resource Analyser*, which will make sure that it finds a spot to expand on.

Responsibilities

Resource needs: The module must determines what resources are required, to construct all the things that are in the *Build*, *Unit*, and *Research Plan*.

Hot Spots

N/A

Standard Implementation

The standard implementation will try and determine where the best place to harvest each of the resource types found in the knowledge base *Resource Types*. If there is not enough resources to fulfil what should be built according to the plans, an expansion is requested to be constructed.

B.7.4 Resource Analyser

This module analyses where to harvest resources, making sure that the workers do not go to far to get them. When a decision has been made to create an expansion, this is the module that should find the best area to place this, according to where there are resources.

Responsibilities

Best Resource Positions: The module has to find the best places to harvest all the types of resources that are required.

Best Place to Expand: Also the module should find the best position to place an expansion, according to its knowledge about the resources on the map.

Hot Spots

N/A

Standard Implementation

The standard implementation will assign the workers to go to the nearest available resource of the type that needs to be gathered. In the case that there is already assigned the maximum amount of workers to gather from that resource, the second nearest will be found, and so forth until an available resource is found. If non is found, it will be assigned to gather another resource type. When requested to find an expansion, the closest grouping or position of resources outside the current base and expansions is found.

B.7.5 Worker Planner

This sub-module assigns workers to harvesting jobs, by looking if there is any workers that is not doing anything. If there are no available workers it should consider if some workers should be reassigned to new tasks. To consider this, the distance from where the workers are to where they are required and the type of job in question, should be considered. This module also makes sure to re-assign workers when they have completed parts of a harvesting task, like moving from the resource back to the depot or dumping the resources into the depot. This behaviour of walking back and forth is controlled by a simple state machine.

Responsibilities

Harvest Resources: The module must issue the commands to move the workers to the resources they should harvest. The module then makes the workers harvest the resources. After this the module moves the workers back to the resource depot, and deposit the resources.

Hot Spots

N/A

Standard Implementation

The standard implementation will make sure that the workers are moved from the depository and to the resource, and when either is reached the worker will dump or harvest accordingly. If there is a change in the distribution of what resources that are required, the module will consider if some workers should be assigned to gather a different type of resource.

B.7.6 Optimise Resource Gathering

This module makes sure that there are not assigned too many workers to harvest from the same resource, because this will be inefficient, and will just create a queue of workers, that are not able to do anything. But if there is too few, this module will make sure that there will be constructed more workers. When these workers are built, the next time the *Resource Manger* is run, these workers will automatically be assigned to gather resources.

Responsibilities

Optimal Use of Resources: The module makes sure that if there are not enough workers assigned on the same resource, more workers will be constructed.

Hot Spots

Number of workers at same resources at same time: There can usually be a certain number of workers harvesting from the same resource at the same time. This number is used to calibrate the number of workers that should be assigned to a certain resource.

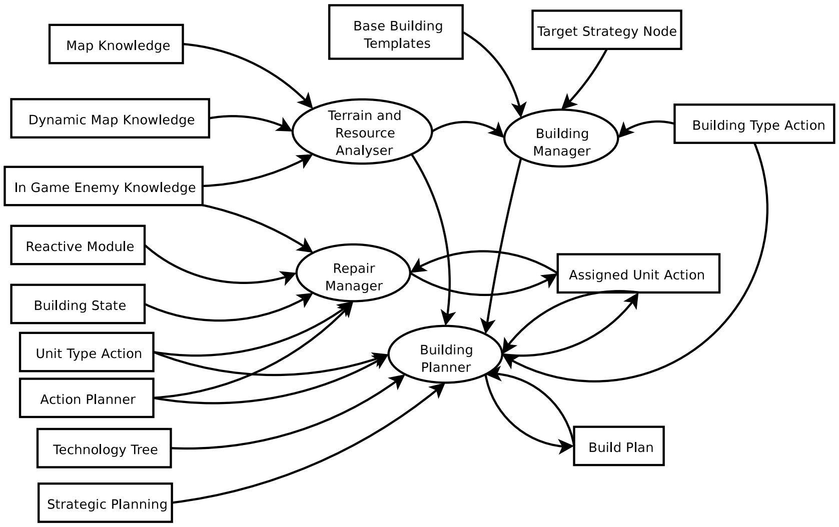


Figure B.7: Internal architecture of the Base Building module

Standard Implementation

The standard implementation will look at the resources harvested from, and identify if there is room for more workers to harvest from this resource. If there is a demand for this resource type it will make sure that more workers will be constructed to harvest from this resource.

B.8 Base Building

This module creates the structure of the base, including placement, and repairing of buildings. When the game starts this module should be triggered to create a starting plan of how to build up the base. If the strategy changes this modules should also be triggered to figure out what additional buildings that might be required. Later when a building is complete, this module should check if there are any additional buildings that should be built. If a request comes from the *Strategic Planning* about creating an expansion, this should be taken into consideration when planning what buildings to create.

The architecture of the *Base Building* module can be seen in Figure B.7. Circles are the sub-modules and the rectangles represents knowledge bases or other modules. The following will present the responsibility of the *Base Building* module, and present each of the sub-modules, and discuss how the sub-modules complete their tasks.

B.8.1 Responsibilities

Analyse Terrain and Resources: Responsible for analysing the environment for the most suitable positioning of building types.

Building Placement: Uses what have been analysed from the environment and what is best for the strategy and from this find the best building placement.

Planning and Prioritising Buildings: Some buildings are more important to build than others when following certain strategies. This should be planned, and the current resource amounts should also be taken into consideration.

Repair of Buildings: When buildings are damaged, there is assigned worker units to repair these buildings.

B.8.2 Structure Overview

This module is divided into sub-modules that can handle each of the subtasks necessary to create and maintain a base. It is made general in the way that the user should define what kind of criteria should be met, and the best base layout according to the strategy. The analysis of the map that is created in the *Terrain and Resource Analyser* is used in combination with the base building templates in the *Building Manager* module, to find what position each building should have. Some buildings have higher priority than others, and some can only be built if other buildings have been built in advance.

B.8.3 Terrain and Resource Analyser

This sub-module should provide analyses of the terrain for optimal defensive positions of buildings, and find the best resource gathering locations. This is done by creating influence maps.

Responsibilities

Areas of Interest: The module must find the areas of interest, like for instance locations with many resources.

Hot Spots

Define Analyser: When looking for the different things in the terrain. The user should define how to analyse for the terrain resources and such.

Standard Implementation

This is very specific from game to game, but there will be an example of how to use influence maps to analyse a map. This part is so specific to the game type, what resources there is, and how the terrain is created so no standard implementation is possible.

B.8.4 Building Manager

This sub-module figures out what the best placement of the buildings is according to terrain, resources, and the current strategy. This module is used when there have been strategic changes, or there have been an attack from the enemy, and parts of the base have to be rebuilt. If a lot of resources have been harvested and through this, changed the defensive structure of the base, or there have become room for more buildings, this module should also react, so that the defence of the base can stay intact, or better placement of buildings can be found.

Responsibilities

Position: The module must in accordance to the base building templates and the analysis of the terrain find the best place to place buildings.

Hot Spots

Base Building Templates: Base building templates are used to define the layout of the base, and through that indirectly dictate the position of buildings.

Standard Implementation

The standard implementation uses the analysis and combines this with the base buildings template and the target strategy, to decide where to place each building. The use of different influence maps can dictate optimal placement of the different types of buildings.

B.8.5 Building Planner

This module sends a request to the *Action Planner* about the construction of a building, which will make sure that there are resources available for the construction. When the request is authorised, a worker is moved to where the building should be placed. Then the worker is assigned the action of constructing the building. A request from the *Strategic Planning* about creating an expansion can be received. This event contains an area where the expansion should be placed and an estimated time before this area is cleared for enemy troops so that construction can begin.

Responsibilities

Control Workers: The module makes sure that each worker is moved to the position where the building is to be constructed.

Expansion Construction: The module sends a work to the area where an expansion should be built, so that the worker is there, when it is estimated that the area is cleared.

Hot Spots

N/A

B.8.6 Repair Manager

The *Repair Manager* handles situations where some buildings are or have been under attack. Then it should make sure that some workers will be assigned to repair these buildings. If there are enemies near the buildings, it should be considered whether to let the workers repair the building.

Responsibilities

Move Unit to Building: The module moves the worker to the damaged building.

Repair Buildings: When a worker is right next to a damaged building, it should start repair that building.

Hot Spots

N/A

B.9 Learning

Learning is responsible for evaluating and updating those prior knowledge bases that can be updated. The internal structure of the *Learning* module can be seen in Figure B.8.

B.9.1 Responsibilities

Evaluate and Revise Known Strategies, Tactics and Base Building Templates (BBT): There is no guaranty that a strategy, tactic or a base building template is perfect from the start. In order to be able

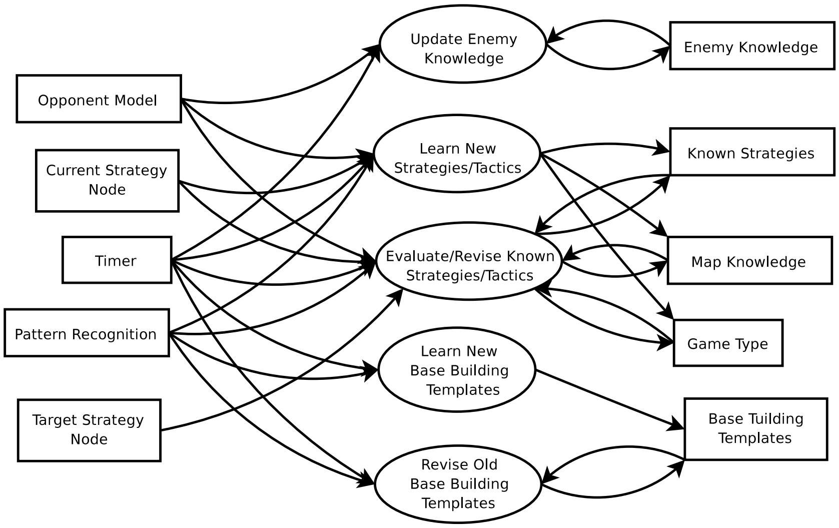


Figure B.8: The internal architecture of the Learning module

to improve these the AI must constantly be able to evaluate their success and be able to revise them to get a better result.

Learn New Strategies, Tactics and BBT: Playing against different opponents or perhaps even the same opponent will make the AI face new strategies, tactics and base building templates. In order to evolve and improve the AI, it has to learn these new things both to be able to recognise the same patterns in a later game and to be able to use them itself.

Update Enemy Knowledge Base: From time to time the prior knowledge base have to be updated with new information. The learning module is responsible for updating the Enemy Knowledge base.

B.9.2 Structure Overview

The overall structure of this module consists of three parts: Evaluate and revise, learn and update knowledge base. Both the evaluate and revise as well as the learn parts furthermore consist of three parts. One for: Strategies, tactics, and base building templates. The general structure of the evaluate and revise known strategies and evaluate and revise known tactics is identical, which means only one will be explained in detail.

B.9.3 Evaluate and Revise Known Strategies

This module is to look back on the strategies that the AI has used and see if there are anything that can be changed or optimised to make the strategy more efficient. In order to do this three functions are needed: One function to evaluate whether the strategy did good or bad, another function to find the key factor that made the strategy good or bad and finally a function that updates the strategy to either focus more on the key factor or correct the mistake. The first function will have to have some memory of two or more game states while using the strategy among which the initial state and the end state should be represented. Using these states it should be able to evaluate whether the AI is in a better situation after using the strategy or not. It should of course take other factors into account. The enemy could have made mistakes and the AI may not have enough information about the enemy to draw a good conclusion. The second function will have to have a log of all decisions made throughout the strategy in order to identify what made the difference. To be able to do this it will also need some way of linking effects to the decisions that caused them. Finally the third function will have to find the rule, controlling the decision that was identified in all the affected strategy nodes and correct it so that it now corresponds to the conclusion of the evaluation.

The factors that need to be considered when evaluating a strategy vary from game to game. In Command and Conquer the factors are more a question of optimisations and in Warcraft it is all about countering the enemy and making decisions, that is, anticipating the enemy's actions and engage in combat at favourable times.

Responsibilities

Evaluate Known Strategies: In order to be able to improve strategies the AI must decide whether or not a strategy is working as it should.

Revise Known Strategies: If a strategy is found not to be working perfectly the AI must identify which factors can be changed or improved.

Hot Spots

Evaluation Method: This method has to decide whether or not the current strategy is working as it should. This is done by looking at the progress made since applying the strategy.

Identify Key Factors: After deciding whether the strategy was effective or not the key factors for this outcome have to be identified so that they can either be enhanced or corrected. If the strategy worked flawlessly then nothing should be changed.

Find Improvements: After the key factors have been identified bad effects have to be corrected and good effects exploited.

Standard Implementation

By default this module will look at the present situation and the situation at last evaluation. The evaluation will simply be based on the AI's own condition change and the enemy's condition change from the previous situation to the present situation.

B.9.4 Evaluate and Revise Known Tactics

Like the previous module this module will also look back and see if anything can be changed or optimised, but this time it is the tactics that are in focus. Basically the three functions needed are more or less the same.

Also here the there are different important factors all depending on the game. In Command and Conquer the positioning of the various units is not as important as in Age of Empires, also the general use of support varies. Some games are almost without support while in other games it is most important.

Responsibilities

Evaluate Known Tactics: Evaluate whether a tactic worked as was intended.

Revise Known Tactics: If the tactic can be improved in any way, do so.

Hot Spots

Evaluation Method: This method has to evaluate whether the tactic had the intended effect or not.

Identify Key Factors: Knowing the outcome this method has to identify the key factors that lead to this.

Find Improvements: Finally improvements have to be found. This can be anything from an alternative deployment to a different unit utilisation.

Standard Implementation

This module will basically use the same method of evaluating a tactic as was used in the strategy evaluation.

B.9.5 Evaluate and Revise Known BBT

This module will have to deal with finding strong and weak points in the BBT using information gathered from games. The result will be templates better suited to deal with a certain map or strategy.

Both the evaluation and the revision of BBT are different from game to game. In some games only the defensive buildings like towers and walls are of any importance, but in most other games the placement of all buildings is important.

Responsibilities

Evaluate Known BBT: Once in a while the AI will have to look at its BBT and see if anything can be improved. The cause for this can be anything from a bad outcome of a battle in the AI's base in which a different base structure might have made the outcome different to optimisation in resource gathering.

Revise Known BBT: If the result of the evaluation is that something has to be improved, the areas that can be improved must be identified and alternatives found.

Hot Spots

Evaluation Method: Given the outcome of a battle or resource gathering optimisation, does the currently used BBT need to be improved?

Identify Key Factors: Identify the factors that were responsible for the outcome.

Find Improvements: In case of a bad outcome, steps must be taken towards a new BBT. This can either mean a new build order or a different building placement.

Standard Implementation

The standard implementation will re-evaluate: The positioning of harvest related buildings if resource gathering needs to be improved, build orders compared to the used strategy, and the positioning of defensive structures based on battles in the base.

B.9.6 Learn New Strategies

By observing the enemy or an ally the AI may gather enough information to model a complete strategy node for the player. If the AI does not know this strategy already it will add it to the strategy tree in *Map Knowledge*, *Enemy Knowledge*, *Game Type knowledge* and *Known Strategies*.

The task of learning new strategies does as such not vary from game to game. The strategy nodes themselves do however. The nodes have to be able to model a complete state of a game and in order to do so all units, buildings, upgrades and other map related information must be accounted for.

Responsibilities

Fitness: The first step in learning a new strategy is in fact to recognise that it is a new strategy. The fitness function will try to match the seen strategy to known strategies in the *Known Strategies* knowledge base. If the strategy deviates from all known strategies by more than a certain value, it will considered a new strategy.

Record New Strategy: When the strategy is identified as a new strategy the a strategy node has to be filled with all known information about it and inserted into *Map Knowledge*, *Enemy Knowledge*, *Game Type knowledge* and *Known Strategies*.

Hot Spots

Fitness Function: The user must define a function to handle the fitness problem mentioned above.

Standard Implementation

The standard implementation of this module will simply insert a strategy node based on the knowledge found in the *In-Game Enemy Knowledge* base if this deviates more than a certain threshold from any known strategy.

B.9.7 Learn New Tactics

The AI can likewise see new tactics be used combined with known or new strategies. When it sees a new tactic, it will have to add this to the strategy node. That is, add the set of rules that describe how this is carried out.

The idea behind the *Learn New Tactics* module is as such not game specific but different games have different rules and different actions available.

This means that the tactics themselves and the rules that they consist of have to be defined from game to game as well as the work done on these.

Responsibilities

Fitness: The module will first have to find the strategy node that is currently used in the strategy tree. If this node does not contain the tactic, the tactic is indeed a new tactic and should be added.

Record New Tactic: The set of new rules representing the tactic must be added to the strategy node. If this means a substitution of the old tactic a new strategy node must be made and the tactic inserted in this.

Hot Spots

Insert New Tactic: The new set of rules have to be defined and inserted so that it can be inserted into the right strategy node. The rules are game specific so the user is responsible for all work done upon these.

Fitness Function: The user must define a function to handle the fitness problem mentioned above.

Standard Implementation

The standard implementation will simply try to imitate the observed actions. It will identify actions done by the involved units and base the rules on these.

B.9.8 Learn New BBT

When scouting an enemy base or seeing how allies build their bases this module must compare the base design to its templates and decide whether or not the seen design is a good one. If it is indeed a good design it must record the design as a template and assign the needed numbers(build order, etc.).

Not only the buildings themselves are different in different games, but also the rules defining how and where they can be built vary. In Warcraft I buildings could only be built next to roads, in Command and Conquer buildings have to be built close to other buildings unless it is a command centre, and in Age of Empires and Warcraft II and Warcraft III buildings can be built anywhere that is free of obstacles.

Responsibilities

Fitness: Like the other learning modules this module also first has to identify the BBT as a new BBT. This is done by searching for the BBT among all the known BBTs. The deviation threshold may vary from game to game.

Create New BBT: When a new BBT has been identified it must be added to the BBT knowledge base.

Hot Spots

Fitness Function: The user must define a function to handle the fitness problem mentioned above.

Standard Implementation

This module will identify important spots such as the location of resources and entrances to the base and record the placement of other buildings relative to these.

B.9.9 Update Enemy Knowledge

This module will simply add information to *Enemy Knowledge* updating *Enemy Knowledge* with information gathered from the game.

When playing any game it is always useful to know how the enemy has played previously. The strategies are modelled by strategy nodes and the strategy trees will model strategy dependencies and frequencies. The only game specific task is to fill out new strategy nodes.

Responsibilities

Update Enemy Knowledge: The soul purpose of this module is to update the prior knowledge base: *Enemy Knowledge*.

Hot Spots

Create New Strategy Node: As already mentioned the creation of the strategy node will have to be defined by the user of the framework.

Standard Implementation

The strategy node will be added to the knowledge base. If the strategy node is already present in the strategy tree the edges leading to it will be incremented by 1.

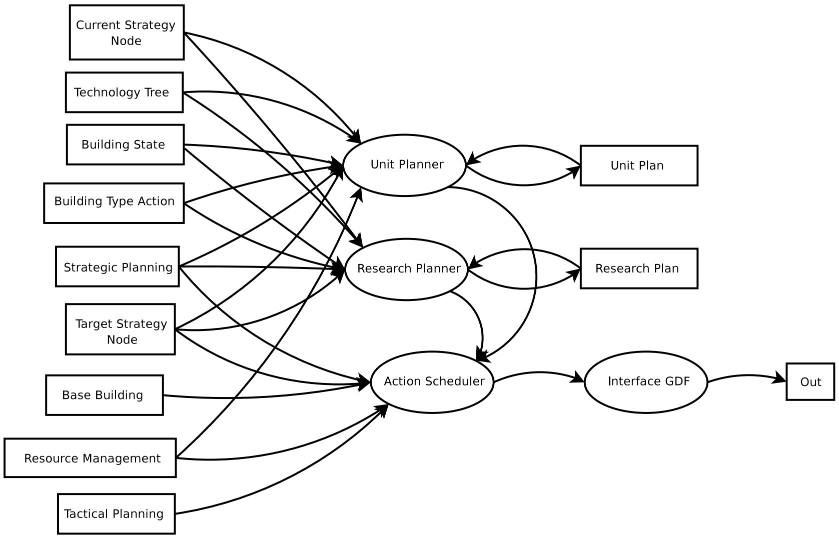


Figure B.9: Internal architecture of the Action Planner

B.10 Action Planner

This module takes care of the final operations necessary to interact with the GDF. It makes sure that units are being build, and that technology is being researched. It schedules the operations that is most critical to be performed first, and then send these operations to the GDF.

The internal architecture of the module can be seen in Figure B.9. Circles in the figure represents internal sub-modules and boxes represents other modules or knowledge bases. The following will first discuss the overall responsibilities of the *Action Planner* module, and then present each of the sub-modules in the internal architecture along with a discussion of how the sub-module is to complete its task.

B.10.1 Responsibilities

Unit Production: The module is responsible for creating all units, and figure out which have the highest priority.

Research: The module prioritises the research required to fulfil the strategy.

Schedule all Actions: The module takes all actions and plans the execution of these according to priority and current resources available.

Using the GDF: The module interfaces with the GDF so the actions that is created in the framework can be mapped to one or more actions in the GDF.

B.10.2 Structure Overview

Each of the sub-modules are responsible for different tasks. The module is created so that it reads proposed actions of the other module in the *Assigned Unit Actions* and *Assigned Building Actions* knowledge bases, and ends up with a list of instructions that calls the GDF, in the way the user has defined.

B.10.3 Unit Planner

The *Unit Planning* sub-module should make sure that there are enough workers, to gather resources and build buildings. It should also make sure the fighting units that fit the current strategy is created, and the right type of scouting units is produced. These units are put into the *Unit Plan* that contains the list of units that should be created. They are then passed on to the action scheduler that figure out when it is possible to start production of the units.

Responsibilities

Unit construction: The module must figure out what units should be constructed in accordance to the strategy.

Hot Spots

Prioritisation of unit types: The type of units that have the highest priority should be defined.

Standard Implementation

The implementation of this module will take the unit types in the target strategy and try and construct the units, so that the distribution of each unit type is always the same as in the strategy.

B.10.4 Research Planner

This module should make sure that the *Technology Tree* is researched in the way that best fits the strategic plan. It will create a *Research Plan*. This plan contains a list of the things that should be researched, and in what order. Each time it is possible the next thing that should be researched is sent to the action scheduler, which will start this research when it has resources and time available for this.

Responsibilities

Research technology: The module must figure out what technology to research in accordance with the strategy.

Hot Spots

N/A

Standard Implementation

The standard implementation will take the target strategy, and with the use of the technology tree figure out how to get to the technology level that is required to follow the strategy.

B.10.5 Action Scheduler

This module should schedule all operations, making sure that the most urgent ones are done first. Because of simulating a human, it should not be possible to do an unlimited amount of operations in one game tick. It should also take into consideration what resources are available, and what is going on at the moment, if the AI is in a battle, it should prioritise after this. Each unit action that is placed in the *Assigned Unit Action* knowledge base is performed.

Responsibilities

Prioritise Construction: There are limited resources, and the module must figure out which constructions have the highest priority, and should be constructed first.

Prioritise Actions: If a unit is requested to do more than one action at the same time, it should figure out what action have the highest priority.

Hot Spots

Prioritising scheme: Depending on how the game is, there is used different prioritising schemes, to tell what constructions and actions have the highest priority, in accordance with all known knowledge.

Standard Implementation

A simple prioritising scheme will be implemented as default.

B.10.6 Interface GDF

This interface should make sure that the operations scheduled will be mapped to operations that can be done in the GDF.

Responsibilities

Interaction with GDF: The module should make sure that the actions are performed in the GDF.

Hot Spots

Actions: The entire module is a hot spot, because depending on how actions are done in the GDF it should be performed in different ways.

Standard Implementation

There is no standard implementation because this module is completely GDF dependent, so there can be no standard implementation of this. There is only defined an interface that this module should implement, and this interface takes a list of actions as input.

Appendix C

Knowledge Bases

This chapter presents all knowledge bases within the framework.

C.1 Prior Knowledge Bases

Map Knowledge: This area represents knowledge about the map terrain, map size, resource locations, strategic and tactical important locations etc.

Enemy Knowledge: Experiences against players throughout several games will give the player an idea of how the enemy player thinks and what kind of strategies she uses. This prevents the player from losing to the same strategies again and again, against the same opponent, as she is capable of trying new things and thereby countering the opponent's strategy. This of course only applies to players of equal skill level in all areas, because knowing the opponent's strategy will often not be enough for novice players to beat professional players.

Gametype Knowledge: Depending on whether the game played is a team game, a 1on1 game or an FFA (Free For All) game, the strategic considerations change.

Known Strategies: Most players have a number of strategies they have either invented for themselves, learned from watching other players or found on the Internet. This area affects both the number and quality of strategies used by the player, but also the capability of predicting the opponent's strategy, and knowing how to counter it.

Known Build Orders: In all RTS games the start of the game is very important and an effective build order can prove invaluable. The build order defines in which order to build everything such as workers, buildings and combat units and also specifies what each worker should be doing at any given time. A build order is often used in connection with a certain strategy trying to maximise the player's resources and getting to a certain point in the strategy as fast as possible.

Resource Types: This knowledge base defines what kind of resources are available in the game.

Technology Tree: This knowledge base defines game specific building dependencies, unit dependencies and research dependencies as well as resource cost for everything in the tree. Furthermore, it includes knowledge about what actions each unit or building is capable of.

Base Building Templates: Contains templates for structuring base building. These templates also contains a prioritised list of buildings to build first for each building plan.

Tactical Knowledge: A knowledge base describing all tactics possible in a certain game. These are essentially also present in the Known Strategies knowledge base, but is here hidden within the different strategy nodes. This knowledge base is basically for easy referencing the different kinds of tactics.

C.2 In-Game Knowledge Bases

Opponent Model: Contains information about the current strategy of the enemy, including a strategy tree and current node information for the enemy. It also specifies beliefs about attributes that have not been scouted, which are there only to represent what the AI currently thinks the opponent is doing. All updates includes a time stamp, which allow the AI to give less importance to variables not updated for a long time.

In-Game Enemy Knowledge: Contains the position of each enemy unit currently visible on the map and knowledge about where certain units have been seen earlier (So the AI do not forget enemy units when they enter fog of war)

Assigned Unit Actions: Information about each controlled unit and the current action assigned to it.

Assigned Building Actions: Information about each controlled building and the current action assigned to it.

Unit State: Contains a collection of all controlled units and the state each of them are in.

Building State: Contains a collection of all controlled buildings and the state each of them are in.

Current Strategy Node: Maintains the current strategy node for the AI player.

Goal Strategy Node: Describes the goal strategy node.

In-Game Own Knowledge: Contains the position and current status of all friendly units and buildings.

Building Plan: Contains the current building plan for the AI's base.

Unit Plan: Contains information about which units to build and in what order.

Research Plan: Contains information about which research upgrades to purchase and in what order.

Mission Knowledge: Contains information about different missions that should be executed in accordance with the current strategy. Each mission is noted along with the goal of the mission and the units assigned to perform it.

Dynamic Map Knowledge: Includes dynamic elements such as resource locations and amounts. Will differ a lot depending on the game in question.

Dynamic Obstacles: Contains the position of all obstacles currently in view that are able to move from one game tick to another.

Appendix D

Test Model

Based on the human model described in Chapter 3 several different areas have been found that is handled by the human player. To test the AI's capabilities in each of these areas several features have been found that together describe how well or how bad the AI handle the same areas. A table showing an overview of the AIs in all the games tested can be found in Appendix E.1. A mark in one of the squares means that the AI, in the game referred to, is capable of handling the described situation. If several different questions are proposed a mark means that the majority of the questions are reasonably handled and the main question satisfactory dealt with.

Below each area is listed along with the chosen situations, each situation described for clarification of the purpose of the situation, and the way this is tested.

D.1 Strategic Planning

Using Counters: If the enemy has chosen a specific strategy most games offer a counter to this specific strategy. Any human player would try to counter the strategy as soon as she discovered what was going on. Is the AI capable of this? This can be tested rather easily. The tester just chooses an extreme strategy that is a strategy that will resolve in victory if not countered, but on the other hand countered rather easily if measures are taken towards this.

Exploiting Weak Spots: Upon scouting an enemy base a human player would immediately identify a weak spot, if any exists. She will then use this information when attacking. The AI can be tested for this capability by identifying the most likely spot to be attacked by the AI. This spot is then fortified with a lot of defensive buildings while leaving a different less likely spot to be attacked defenceless.

Strategic Variation in one Game: Does the AI vary its strategy throughout a single game? If, for instance, the AI has chosen a strategy at game start and this strategy fails, will it then try to change its strategy, perhaps even towards countering the enemy strategy? This is tested by simply noting the strategy that the AI is using at the start of the game. If the AI does not change the strategy (unit combination, point of attack, etc.) even when losing, it is incapable of this.

Strategic Variation Game to Game: Does the AI change its strategy from game to game? A human player would change her strategy from game to game especially when playing against the same opponent. By doing this, she is less likely to let her opponent know what she is up to. The AI is tested by simply playing a series of games and observing which strategy the AI chooses.

Reasonable Expansions: This question actually covers two questions: Is the AI able to choose a good time for expanding? And does it choose a good spot for expanding. The first question is hard to test, because it is based upon the chosen strategy and general game experience. Here it is up to the tester to judge how well this is done. The other question is a bit simpler to test. There are several criteria for a good expansion site: Is it close to the main base? Is it well hidden? Is the harvesting building close to the resources? Is the expansion well-placed in relation to the enemy?

Using Map: Being able to use the map can put the AI in favourable positions when fighting, prevent it from falling into ambushes at bad locations, and even open the possibility for using map specific strategies. This can be tested by trying to lure the AI into an ambush in a choke point, using high ground against it, and also observe, whether it is trying to do the same to the tester.

Good Build Order: A good build order is crucial, especially in the early stages of the game. This can be tested by observing the AI throughout the first 3-5 minutes and see how well it manages buildings, workers and resources compared to the chosen strategy.

D.2 Tactical Planning

Using Formations: Using formations can prevent the wrong units from being exposed to damage and it generally means that the units end up in the position that they were designed for when entering a battle. This can be tested by observing how the AI moves its army. This is especially the case when entering a battle, or just if the army consists of different units of varying movement speed.

Map Considered when Moving: How does the AI handle choke points, exiting transports and other map specific situations?Is it just pushing the army through the hole as fast as possible, letting the first units walk on ahead of the rest of the army or is it keeping the army gathered? Is the AI avoiding goose walk? This is tested by observing the AI in such a situation.

Using Tactical Manoeuvres: Does the AI use tactical manoeuvres? A tactical manoeuvre can be anything from trying to flank the enemy to get through the lines and attack the light armoured units at the back, to lure the enemy into a bad location. The possibilities of tactical manoeuvres vary from game to game.

Measure Own Str. vs Enemy Str.: How well does the AI measure its own strength compared to the strength of the enemy? This can be seen when the AI attacks with an inferior army. What does it do when it sees the enemy army? Does it attack anyway or retreat to pick up more units?

Staying in Control of Units: An unattended army can easily be divided by attacking one of the units at the perimeter of the army and run away. In most cases the AI in each unit will make the attacked units and the immediate surrounding units to follow the attacked. The attacker is thus able to split up an army and deal with each portion separately. How well is the AI at dealing with this?

D.3 Micromanagement

Saving Hurt Units: If the game features healing (either creature regeneration or by support) it is in most cases an advantage to save as many units as possible throughout a battle. As soon as a unit is severely hurt, it should either be removed from the battle field or at least from the line of fire. This is easily observed in any battle.

Focus Fire: The counter to saving the hurt units is to focus all (or at least a lot of) fire on a single unit in turn so that the opponent has not got time to remove it from the battlefield. The idea is also that for every single unit, you can kill, there is one less unit dealing damage to your army. This is also easily observed in a battle.

Counter Focus: Some games feature unit to unit counters. That means that given some unit type A there exists a unit type B that is designed to deal with unit type A. During a battle, how well is the AI to manoeuvre the units of type B, so that they are faced with units of the type A?

Using Support: The correct use of support units can mean the difference between failure or victory. How well is the AI to decide, which units should receive the support, and when to use support at all (given that the use is limited by for instance mana). This is harder to observe in a battle as it can be rather subtle, but it can easily be seen, if the tester is either in an observing position or is able to review replays.

D.4 Resource Management

Predicting Resource Needs: By predicting the resource needs, the AI is able to minimise the time it takes to reach a certain technology level, or the production of a certain number of a specific unit. This can be tested by observing the resource usage of the AI. Does it stock up the resources needed to carry out the strategy, or does it end up waiting for the required resources?

Spending Available Resources: How well is the AI at spending the available resources? There is no point in expanding if the extra resources are not spent, or at least taken into account, when evaluating the strategy. This is tested by looking at the AI's resource amount throughout the game. Is it spending the resources? Does it upgrade units? Does it produce enough units?

Flexible Resource Gathering: Some strategies require one branch of resources, and very little or none of another branch of resources. If such a strategy is chosen by the AI, it would be stupid to gather all kinds of resources, instead of just the one that is needed. How well is the AI at this, and does it change the resource gathering strategy at all, when a different overall strategy is chosen?

D.5 Base Building

Good Placement of Def. Buildings: A bad placement of defensive buildings can mean that they are hardly worth anything at all. A good placement, however, can mean that the base is almost impregnable. The use of defensive buildings is varying a lot from game to game so it is up to the tester to judge, how well the AI is placing these.

Good Placement of Hrv. Buildings: A good placement of a harvesting building can mean the speed up of harvesting by several orders of magnitude compared to a bad placement. Harvesting buildings should be placed as close to the resource as possible.

Sensible Base: How good is the overall building placement in the base? Building placement strategies are also very game specific, so once again it is up to the tester to judge. The tester should however take into account: How well the base is defended against drops, direct attack and ultimate weapons (Nuclear missiles, Area of effect spells, and the like).

D.6 Scouting

Does It Scout At All: This question covers the entire area as if the answer to this question is negative, the following questions will all be negative. Does the AI scout at all? The alternative to scouting is cheating by having the entire map available. This is rather easy to test. Does the AI use scouts, or does it move around like it knows what is happening on the entire map? This is best seen by either observing the AI or reviewing a replay.

Scouting Map: How well is the AI at scouting the map? Does it scout possible expansion sites for enemy expansions? Does it scout different starting locations for the enemy base? Etc.

Scouting Enemy: Does the AI scout the enemy? By scouting well the AI will be able to know exactly what the enemy is up to and take measures to counter this.

Scouting at Sensible Times: Is the AI scouting at sensible times? This is game specific, but the tester should note how many times the AI scouts the enemy, whether the interval is reasonable, and if the time, it scouts, is well-chosen compared to the time, it will be able to see which branch of the technology tree, the enemy has chosen.

Using the Acquired Information: Does the AI use the acquired information to adjust its strategy or is it just for show? This is easily tested by choosing an extreme strategy and make sure that the AI sees this. If it counters this is obviously the case.

Sensible Unit Used for Scouting: Choosing the right unit for scouting is also important, as the unit is in danger of being caught when scouting. Choosing the right unit will minimise the cost of the sacrifice. This can be done in several different ways. One way is to send a low cost unit and the other is to send a unit that is unlikely to be caught. How well is the AI doing this?

D.7 Learning

Learning: This should be tested in two ways. The first way is to play against the AI in one game. Does the AI seem to learn new strategies throughout the game by either observing what the tester does or by reasoning? The other is to observe the AI throughout several games. Does the AI seems to learn from game to game, that is learning from past experience.

D.8 Cooperation

AI-AI Cooperative Strategy: When two AIs are allied, how well are they at choosing a shared strategy, and do they do this at all? This is tested by simply observing the AIs' strategy in a couple of games.

Cooperating: Do the AIs cooperate? Are they coordinating attacks and defence? This can also be seen by observing a couple of games.

Resource Sharing: How well are the AIs at resource sharing, and do they do it at all? This is best tested by observing the resource amount of both AIs throughout the game. The tester could for instance take out all the workers belonging to one of the AIs to force a situation, where resource sharing would be obvious.

Human-AI Communication Available: Is it possible for the human player to communicate with the AI?

AI-Human Communication Available: Is it possible for the AI to communicate with the player?

Helping if Human Attacks: Does the AI join forces with the human player, when the human player decides to attack the enemy?

Helping if Human is Attacked: Does the AI come to help if the human player is attacked?

Handling Temporary Alliances(FFA): How well is the AI at handling temporary alliances like the ones encountered in Free For All games?

Appendix E

Test Table A

	Red Alert	Dark Reign 2	Warzone 2100	Age of Mythology	Empire Earth 2	Starcraft	Armies of Exigo	Warcraft II	Warcraft III
Strategic Planning									
Using Counters				X					
Exploiting Weak Spots									
Strategic Variation in one Game				X	X				
Strategic Variation Game to Game				X	X	X	X	X	
Reasonable Expansions				X	X				X
Using Map									
Good Buildorder				X	X	X	X		X
Tactical Planner									
Using Formations			X	X	X		X		X
Map Considered when Moving									
Using Tactical Manoeuvres							X		
Measure Own Str. vs Enemy Str.							X	X	X
Staying in Control of Units					X		X		
Micromanagement									
Saving Hurt Units			X						X
Focus Fire			X						X
Counter Focus									
Using Support				X	X	X	X		X
Resource Management									
Predicting Resource Needs									
Spending Available Resources	X	X	X	X	X				
Flexible Resource Gathering									
Base Building									
Good Placement of Def. Buildings						X	X		X
Good Placement of Hrv. Buildings				X	X	X			
Sensible Base				X	X				X
Scouting									
Does It Scout At All				X	X				
Scouting Map				X	X				
Scouting Enemy				X	X				
Scouting at Sensible Times									
Using the Acquired Information					X				
Sensible Unit Used for Scouting				X	X				
Learning									
Learning									
Cooperation									
AI-AI Cooperative Strategy									
Cooperating					X		X		X
Resource Sharing									
Human-AI Communication Available					X				
AI-Human Communication Available									X
Helping if Human Attacks									X
Helping if Human is Attacked									X
Handling Temporary Alliances									

Table E.1: Test Table A

Appendix F

Test Table B

	Red Alert	Dark Reign 2	Warzone 2100	Age of Mythology	Empire Earth 2	Starcraft	Armies of Exigo	Warcraft II	Warcraft III	Prototype Implementation	Complete Implementation
Strategic Planning											
Using Counters				X						X	X
Exploiting Weak Spots											X
Strategic Variation in one Game				X	X					X	X
Strategic Variation Game to Game				X	X	X	X	X		X	X
Reasonable Expansions				X	X				X		X
Using Map											X
Good Buildorder				X	X	X	X		X		X
Tactical Planner											
Using Formations			X	X	X		X		X		X
Map Considered when Moving											X
Using Tactical Manoeuvres							X				X
Measure Own Str. vs Enemy Str.							X	X	X	/	X
Staying in Control of Units					X		X				X
Micromanagement											
Saving Hurt Units			X						X	/	X
Focus Fire			X						X	/	X
Counter Focus											X
Using Support				X	X	X	X		X		X
Resource Management											
Predicting Resource Needs											X
Spending Available Resources	X	X	X	X	X					/	X
Flexible Resource Gathering											X
Base Building											
Good Placement of Def. Buildings						X	X		X		X
Good Placement of Hrv. Buildings				X	X	X					X
Sensible Base				X	X				X		X
Scouting											
Does It Scout At All				X	X					X	X
Scouting Map				X	X						X
Scouting Enemy				X	X					/	X
Scouting at Sensible Times											X
Using the Acquired Information					X					X	X
Sensible Unit Used for Scouting				X	X					X	X
Learning											
Learning											X
Cooperation											
AI-AI Cooperative Strategy											
Cooperating					X		X		X		
Resource Sharing											
Human-AI Communication Available					X						
AI-Human Communication Available									X		
Helping if Human Attacks									X		
Helping if Human is Attacked									X		
Handling Temporary Alliances											

Table F.1: Test Table B

Appendix G

Game Logs

The following will list five game log examples, demonstrating what the AI is doing and reasoning about during a game. All game logs will include the AI's choice of strategy, when it is scouts and what unit it chooses, and reasoning about the opponent's strategy. The AI uses the strategy tree shown in Figure K.3 in all the examples. When new information is discovered about the enemy, the opponent model is printed, as well as the reasoning about the opponent's strategy done in the *Probabilistic Reasonoing* module. The "Potential Strategies" output indicates which strategies the AI currently thinks the opponent is doing, and the number after each potential strategy indicates a deviation factor compared to the actual strategy tree node. Following this, the *Strategic Planning* module prints the potential counter strategies, and a number indicating how likely it is that the strategy is going to counter the opponent's strategy. Game log G.1 and G.2 furthermore includes when the AI has built a unit, to show how the AI is capable of following the currently selected strategy.

Listing G.1: AI game log 1

```
[Game Tick: 1]
Chosen start strategy: Fast tech
[Game Tick: 2]
...
[Game Tick: 31]
Scouting Mission Started:
Unit type selected for scouting: worker
[Game Tick: 32]
...
[Game Tick: 61]
Opponent model:
- Name: marine - 0 - Percentage: 0
- Name: tank - 0 - Percentage: 0
- Name: worker - 6 - Percentage: 100
- Name: barracks - 0 - Percentage: 0
- Name: controlCenter - 0 - Percentage: 0
- Name: factory - 0 - Percentage: 0
```

```
Potential Strategies:
- Fast expand - 0
Counter strategy: Marines - Ability to counter strategy: 100
Chosen Counter Strategy: Marines
[Game Tick: 62]
...
[Game Tick: 89]
Opponent model:
- Name: marine - 2 - Percentage: 18.1818
- Name: tank - 3 - Percentage: 27.2727
- Name: worker - 6 - Percentage: 54.5455
- Name: barracks - 0 - Percentage: 0
- Name: controlCenter - 1 - Percentage: 50
- Name: factory - 1 - Percentage: 50
Potential Strategies:
- Fast expand - 7.75
- Fast tech - 7.1
- Mixed - 6.75
Counter strategy: Fast expand - Ability to counter strategy: 92.9
Counter strategy: Marines - Ability to counter strategy: 92.25
Counter strategy: Mass tanks - Ability to counter strategy: 93.25
Chosen Counter Strategy: Mass tanks
[Game Tick: 90]
...
```

Listing G.2: AI game log 2

```
[Game Tick: 1]
Chosen start strategy: Marines
[Game Tick: 2]
...
[Game Tick: 31]
Scouting Mission Started:
Unit type selected for scouting: worker
[Game Tick: 32]
...
[Game Tick: 167]
Opponent model:
- Name: marine - 2 - Percentage: 22.2222
- Name: tank - 1 - Percentage: 11.1111
- Name: worker - 6 - Percentage: 66.6667
- Name: barracks - 0 - Percentage: 0
- Name: controlCenter - 0 - Percentage: 0
- Name: factory - 0 - Percentage: 0
Potential Strategies:
- Fast expand - 8.25
Counter strategy: Marines - Ability to counter strategy: 91.75
Chosen Counter Strategy: Marines
[Game Tick: 168]
...
[Game Tick: 184]
Opponent model:
- Name: marine - 5 - Percentage: 41.6667
- Name: tank - 1 - Percentage: 8.33333
- Name: worker - 6 - Percentage: 50
- Name: barracks - 0 - Percentage: 0
- Name: controlCenter - 1 - Percentage: 50
- Name: factory - 1 - Percentage: 50
Potential Strategies:
- Fast expand - 8.25
- Marines - 7.25
- Mass marines - 9
- Mixed - 7.41667
Counter strategy: Fast tech - Ability to counter strategy: 92.75
Counter strategy: Marines - Ability to counter strategy: 91.25
Counter strategy: Mass tanks - Ability to counter strategy: 92.5833
Counter strategy: Mixed - Ability to counter strategy: 91
Chosen Counter Strategy: Fast tech
[Game Tick: 185]
...
```

Listing G.3: AI game log 3

```
[Game Tick: 1]
Chosen start strategy: Marines
[Game Tick: 2]
...
[Game Tick: 31]
Scouting Mission Started:
Unit type selected for scouting: worker
[Game Tick: 32]
...
[Game Tick: 56]
Unit Built: worker
[Game Tick: 57]
...
[Game Tick: 64]
Unit Built: marine
[Game Tick: 65]
...
[Game Tick: 112]
Unit Built: worker
[Game Tick: 113]
...
[Game Tick: 128]
Unit Built: marine
[Game Tick: 129]
...
[Game Tick: 192]
Unit Built: marine
[Game Tick: 193]
...
[Game Tick: 256]
Unit Built: marine
[Game Tick: 257]
...
[Game Tick: 320]
Unit Built: marine
```

```
[Game Tick: 321]
...
[Game Tick: 346]
Opponent model:
- Name: marine - 3 - Percentage: 30
- Name: tank - 1 - Percentage: 10
- Name: worker - 6 - Percentage: 60
- Name: barracks - 0 - Percentage: 0
- Name: controlCenter - 0 - Percentage: 0
- Name: factory - 0 - Percentage: 0
Potential Strategies:
- Fast expand - 7.33333
- Marines - 9.41667
- Mass marines - 7.36667
- Mixed - 9.91667
Counter strategy: Fast tech - Ability to counter strategy: 90.5833
Counter strategy: Marines - Ability to counter strategy: 92.6667
Counter strategy: Mass tanks - Ability to counter strategy: 90.0833
Counter strategy: Mixed - Ability to counter strategy: 92.6333
Chosen Counter Strategy: Marines
[Game Tick: 347]
...
[Game Tick: 358]
Opponent model:
- Name: marine - 8 - Percentage: 38.0952
- Name: tank - 7 - Percentage: 33.3333
- Name: worker - 6 - Percentage: 28.5714
- Name: barracks - 0 - Percentage: 0
- Name: controlCenter - 1 - Percentage: 50
- Name: factory - 1 - Percentage: 50
Potential Strategies:
- Fast expand - 7.33333
- Fast tech - 9.86667
- Marines - 9.65
- Mass marines - 9.96667
- Mixed - 3.5
Counter strategy: Fast expand - Ability to counter strategy: 90.1333
Counter strategy: Fast tech - Ability to counter strategy: 90.35
Counter strategy: Marines - Ability to counter strategy: 92.6667
Counter strategy: Mass tanks - Ability to counter strategy: 96.5
Counter strategy: Mixed - Ability to counter strategy: 90.0333
Chosen Counter Strategy: Mass tanks
[Game Tick: 359]
...
[Game Tick: 414]
Unit Built: worker
[Game Tick: 415]
...
[Game Tick: 470]
Unit Built: worker
[Game Tick: 471]
...
[Game Tick: 486]
Unit Built: tank
```

```
[Game Tick: 487]
...
[Game Tick: 614]
Unit Built: tank
[Game Tick: 615]
...
[Game Tick: 742]
Unit Built: tank
[Game Tick: 743]
...
```

Listing G.4: AI game log 4

```
[Game Tick: 1]
Chosen start strategy: Fast tech
[Game Tick: 2]
...
[Game Tick: 31]
Scouting Mission Started:
Unit type selected for scouting: worker
[Game Tick: 32]
...
[Game Tick: 56:]
Unit Built: worker
[Game Tick: 57:]
...
[Game Tick: 112:]
Unit Built: worker
[Game Tick: 113:]
...
[Game Tick: 128:]
Unit Built: tank
[Game Tick: 129:]
...
[Game Tick: 256]
Unit Built: tank
[Game Tick: 257]
...
[Game Tick: 384]
Unit Built: tank
[Game Tick: 385]
...
[Game Tick: 392]
Opponent model:
- Name: marine - 0 - Percentage: 0
- Name: tank - 1 - Percentage: 14.2857
- Name: worker - 6 - Percentage: 85.7143
- Name: barracks - 0 - Percentage: 0
- Name: controlCenter - 0 - Percentage: 0
- Name: factory - 0 - Percentage: 0
Potential Strategies:
- Fast expand - 2.56667
- Fast tech - 8.8
Counter strategy: Fast expand - Ability to counter strategy: 91.2
Counter strategy: Marines – Ability to counter strategy: 97.4333
Chosen Counter Strategy: Marines
[Game Tick: 393]
...
[Game Tick: 400]
Opponent model:
– Name: marine – 2 – Percentage: 16.6667
– Name: tank – 4 – Percentage: 33.3333
– Name: worker – 6 – Percentage: 50
– Name: barracks – 0 – Percentage: 0
– Name: controlCenter – 1 – Percentage: 100
– Name: factory – 0 – Percentage: 0
Potential Strategies:
– Fast expand – 6.05
– Fast tech – 5.85
– Mass tanks – 9.36667
– Mixed – 7.51667
Counter strategy: Fast expand – Ability to counter strategy: 94.15
Counter strategy: Marines – Ability to counter strategy: 93.95
Counter strategy: Mass marines – Ability to counter strategy: 90.6333
Counter strategy: Mass tanks – Ability to counter strategy: 92.4833
Chosen Counter Strategy: Fast expand
[Game Tick: 401]
...
```

Listing G.5: AI game log 5

```
[Game Tick: 1]
Chosen start strategy: Marines
[Game Tick: 2]
...
[Game Tick: 31]
Scouting Mission Started:
Unit type selected for scouting: worker
[Game Tick: 32]
...
[Game Tick: 367]
Opponent model:
– Name: marine – 2 – Percentage: 50
– Name: tank – 0 – Percentage: 0
– Name: worker – 2 – Percentage: 50
– Name: barracks – 0 – Percentage: 0
– Name: controlCenter – 0 – Percentage: 0
– Name: factory – 0 – Percentage: 0
Potential Strategies:
– Fast expand – 9.16667
– Marines – 4.25
– Mass marines – 2.2
Counter strategy: Fast tech – Ability to counter strategy: 95.75
Counter strategy: Marines – Ability to counter strategy: 90.8333
Counter strategy: Mixed – Ability to counter strategy: 97.8
Chosen Counter Strategy: Mixed
[Game Tick: 368]
...
```

Appendix H

Performance Log

Listing H.1: Module game tick performance log

```
Game tick: 1
 Action planner 0
 Base building 0
 GDF connection 0
 Percept interpreter 42
 Reactive module 0
 Resource manager 0
 Strategic planner 0
 Tactical planner 15

Game tick: 2
 Action planner 1
 Base building 0
 GDF connection 0
 Percept interpreter 0
 Reactive module 0
 Resource manager 0
 Tactical planner 14

Game tick: 3
 Action planner 0
 GDF connection 0
 Percept interpreter 0
 Reactive module 0
 Resource manager 11
 Tactical planner 14

Game tick: 4
 Action planner 0
 GDF connection 0
 Percept interpreter 0
 Reactive module 0
 Tactical planner 14

...

```

```
Game tick: 31
 Action planner 0
 GDF connection 0
 Percept interpreter 0
 Reactive module 0
 Tactical planner 14

Game tick: 32
 Action planner 0
 GDF connection 0
 Percept interpreter 0
 Reactive module 0
 Resource manager 0
 Strategic planner 0
 Tactical planner 28

Game tick: 33
 Action planner 0
 GDF connection 0
 Percept interpreter 0
 Reactive module 0
 Tactical planner 14
...
```

Appendix I

Pathfinding Tests

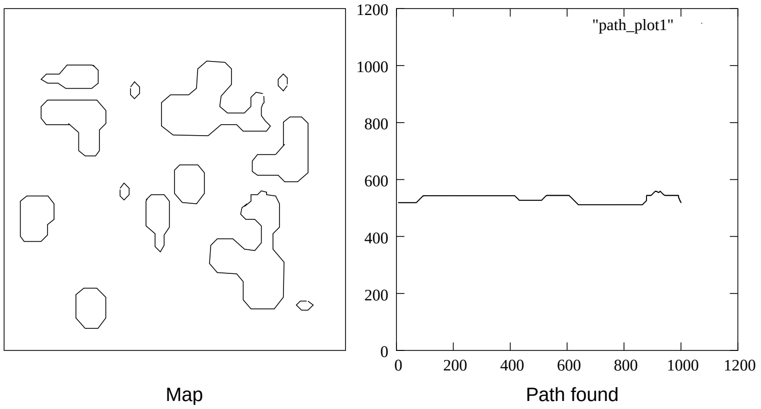


Figure I.1: Path found in pathfinding test

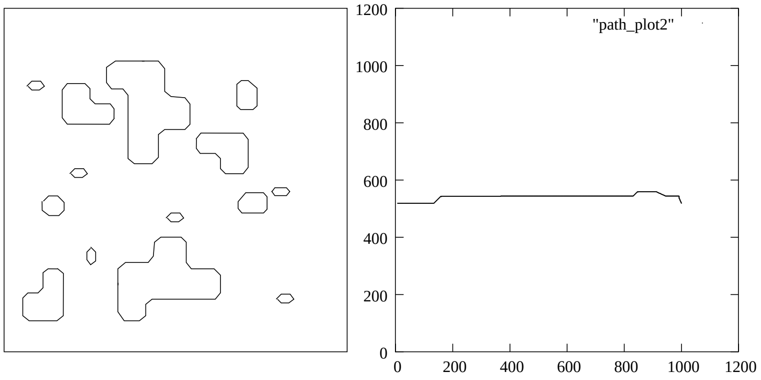


Figure I.2: Path found in pathfinding test

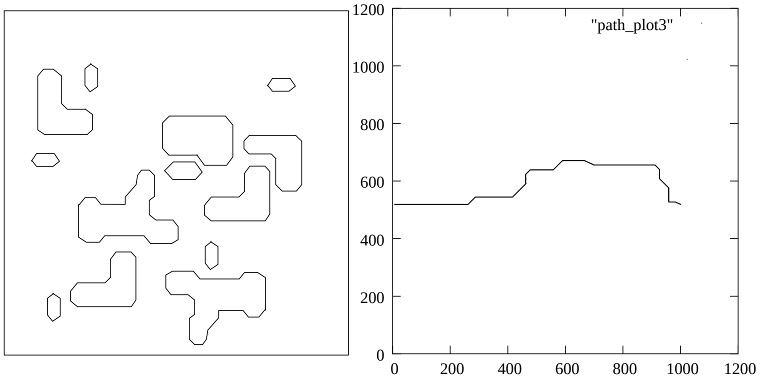


Figure I.3: Path found in pathfinding test

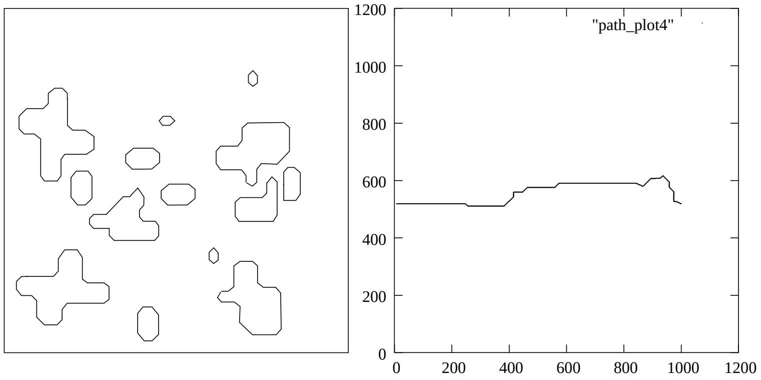


Figure I.4: Path found in pathfinding test

Appendix J

Code Examples

J.1 Unit and Building Specifications

Listing J.1: Specification of a worker type

```
name = "worker"
type = "Unit"
preconditions = ["controlCenter"]
hitpoints = 60
attack_max = 5
attack_min = 2
ground_attack_range = 4
movement_speed = 3
sight_range = 5
actions = ["move", "attack", "stop", "mine", "return_resources",
  "build_controlCenter", "build_barracks", "build_factory"]
minerals = 75
gas = 0
built_by = "controlCenter"
build_time = 75
supply_cost = 0
```

Listing J.2: Specification of a marine type

```
name = "marine"
type = "Unit"
preconditions = ["barrack"]
hitpoints = 100
attack_max = 50
attack_min = 30
ground_attack_range = 8
movement_speed = 3
sight_range = 6
actions = ["move", "attack", "stop"]
minerals = 100
built_by = "barrack"
build_time = 100
supply_cost = 1
```

Listing J.3: Specification of a tank type

```
name = "tank"
type = "Unit"
preconditions = ["factory"]
hitpoints = 500
attack_max = 10
attack_min = 5
ground_attack_range = 6
movement_speed = 2
sight_range = 7
actions = ["move", "attack", "stop"]
minerals = 400
gas = 0
built_by = "factory"
build_time = 0
supply_cost = 2
```

Listing J.4: Specification of a command center type

```
name = "controlCenter"
type = "Building"
preconditions = []
hitpoints = 5000
attack_max = 0
attack_min = 0
ground_attack_range = 0
movement_speed = 2
sight_range = 4
actions = ["train_worker", "stop"]
minerals = 600
gas = 0
built_by = "worker"
build_time = 300
```

Listing J.5: Specification of a barracks type

```
name = "barracks"
type = "Building"
preconditions = ["controlCenter"]
hitpoints = 1000
attack_max = 0
attack_min = 0
ground_attack_range = 0
movement_speed = 0
sight_range = 4
actions = ["train_marine", "stop"]
minerals = 400
gas = 0
built_by = "worker"
build_time = 200
```

Listing J.6: Specification of a factory type

```
name = "factory"
```

```
type = "Building"
preconditions = ["controlCenter", "barracks"]
hitpoints = 1400
attack_max = 0
attack_min = 0
ground_attack_range = 0
movement_speed = 0
sight_range = 4
actions = ["build_tank", "stop"]
minerals = 400
gas = 0
built_by = "worker"
build_time = 200
```

J.2 Known Strategies

Listing J.7: Code for defining strategies in the Known Strategies knowledge base

```
starting_point = {
"name" : "Starting␣point",
"precondition" : "",
"follow_up_strategies" : ["Fast␣expand", "Marines",
   "Fast␣tech"],
"counters" : [],
"percentage_use" : 100,
"time" : 0,
"purpose" : "step",
"expansions" : 0,
"controlCenter" : 1,
"barracks" : 0,
"factory" : 0,
"worker" : 6,
"marine" : 0,
"tank" : 0
}

fast_expand = {
"name" : "Fast␣expand",
"precondition" : "Starting␣point",
"follow_up_strategies" : ["Mass␣marines"],
"counters" : ["Marine␣Rush"],
"percentage_use" : 30,
"time" : 500,
"purpose" : "step",
"expansions" : 1,
"controlCenter" : 2,
"barracks" : 0,
"factory" : 0,
"worker" : 12,
"marine" : 0,
"tank" : 0
}

mass_marines = {
```

```
"name" : "Mass␣marines",
"precondition" : "Fast␣expand",
"follow_up_strategies" : [],
"counters" : ["Mixed"],
"percentage_use" : 100,
"time" : 1000,
"purpose" : "attack",
"expansions" : 1,
"controlCenter" : 2,
"barracks" : 0,
"factory" : 0,
"worker" : 18,
"marine" : 30,
"tank" : 0
}

marines = {
"name" : "Marines",
"precondition" : "Starting␣point",
"follow_up_strategies" : ["Mixed"],
"counters" : ["Fast␣tech"],
"percentage_use" : 40,
"time" : 500,
"purpose" : "attack",
"expansions" : 0,
"controlCenter" : 1,
"barracks" : 1,
"factory" : 0,
"worker" : 8,
"marine" : 10,
"tank" : 0
}

mixed = {
"name" : "Mixed",
"precondition" : "Marines",
"follow_up_strategies" : [],
"counters" : ["Mass␣tanks"],
"percentage_use" : 100,
"time" : 1000,
"expansions" : 0,
"purpose" : "attack",
"controlCenter" : 1,
"barracks" : 1,
"factory" : 1,
"worker" : 10,
"marine" : 15,
"tank" : 10
}

fast_tech = {
"name" : "Fast␣tech",
"precondition" : "Starting␣Point",
"follow_up_strategies" : ["Mass␣tanks"],
"counters" : ["Fast␣expand"],
"percentage_use" : 30,
"time" : 500,
"purpose" : "step",
"expansions" : 0,
"controlCenter" : 1,
"barracks" : 1,
"factory" : 1,
"worker" : 8,
"marine" : 0,
"tank" : 5
}

mass_tanks = {
"name" : "Mass␣tanks",
"precondition" : "Fast␣tech",
"follow_up_strategies" : [],
"counters" : ["Mass␣marines"],
"percentage_use" : 100,
"time" : 1000,
"purpose" : "attack",
"expansions" : 0,
"controlCenter" : 1,
"barracks" : 1,
"factory" : 2,
"worker" : 12,
"marine" : 0,
"tank" : 20
}

strategies = [starting_point, fast_expand, marines, fast_tech,
    mass_marines, mixed, mass_tanks]
```

Appendix K

Important Figures

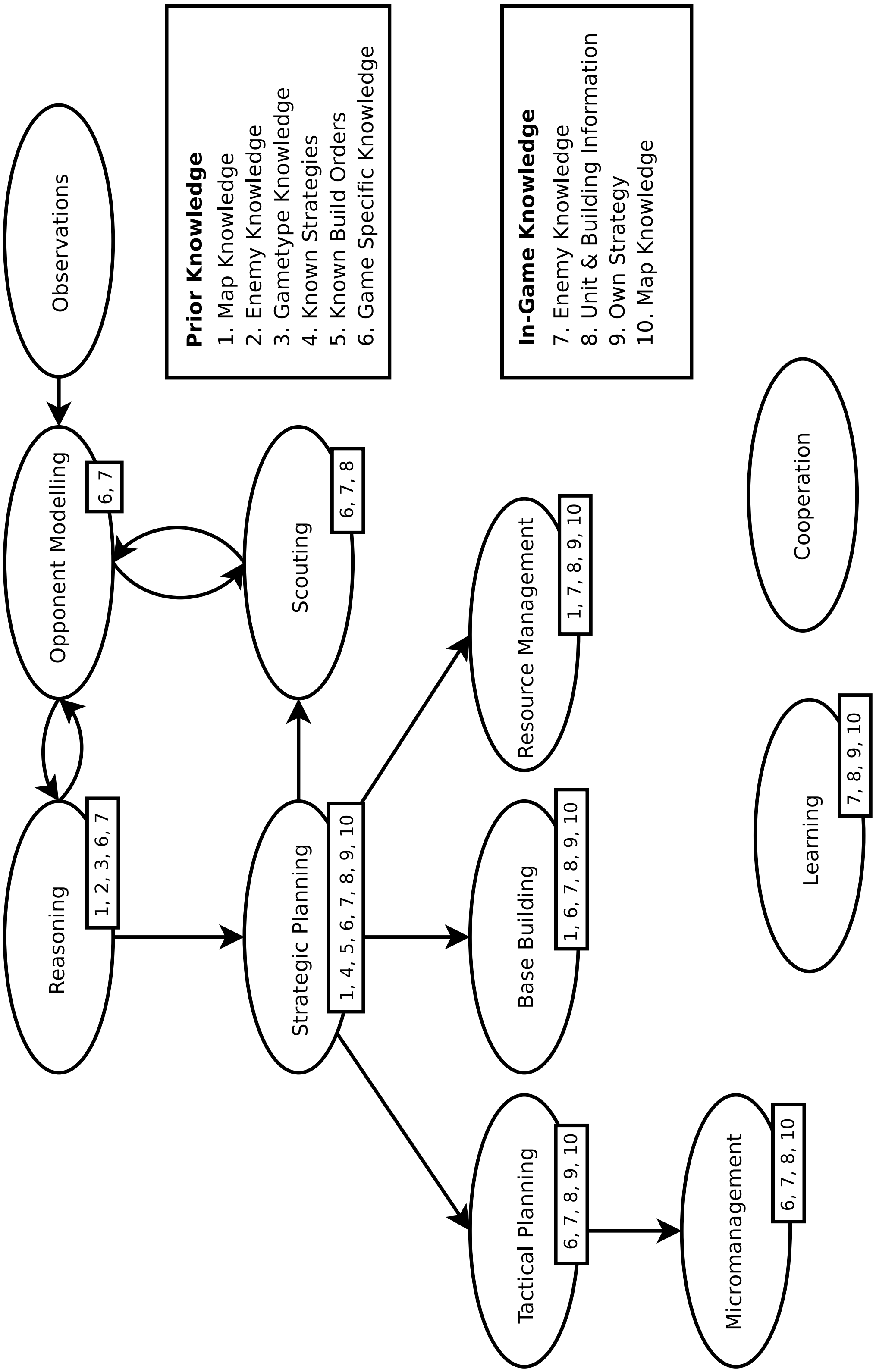


Figure K.1: A human model for playing RTS games

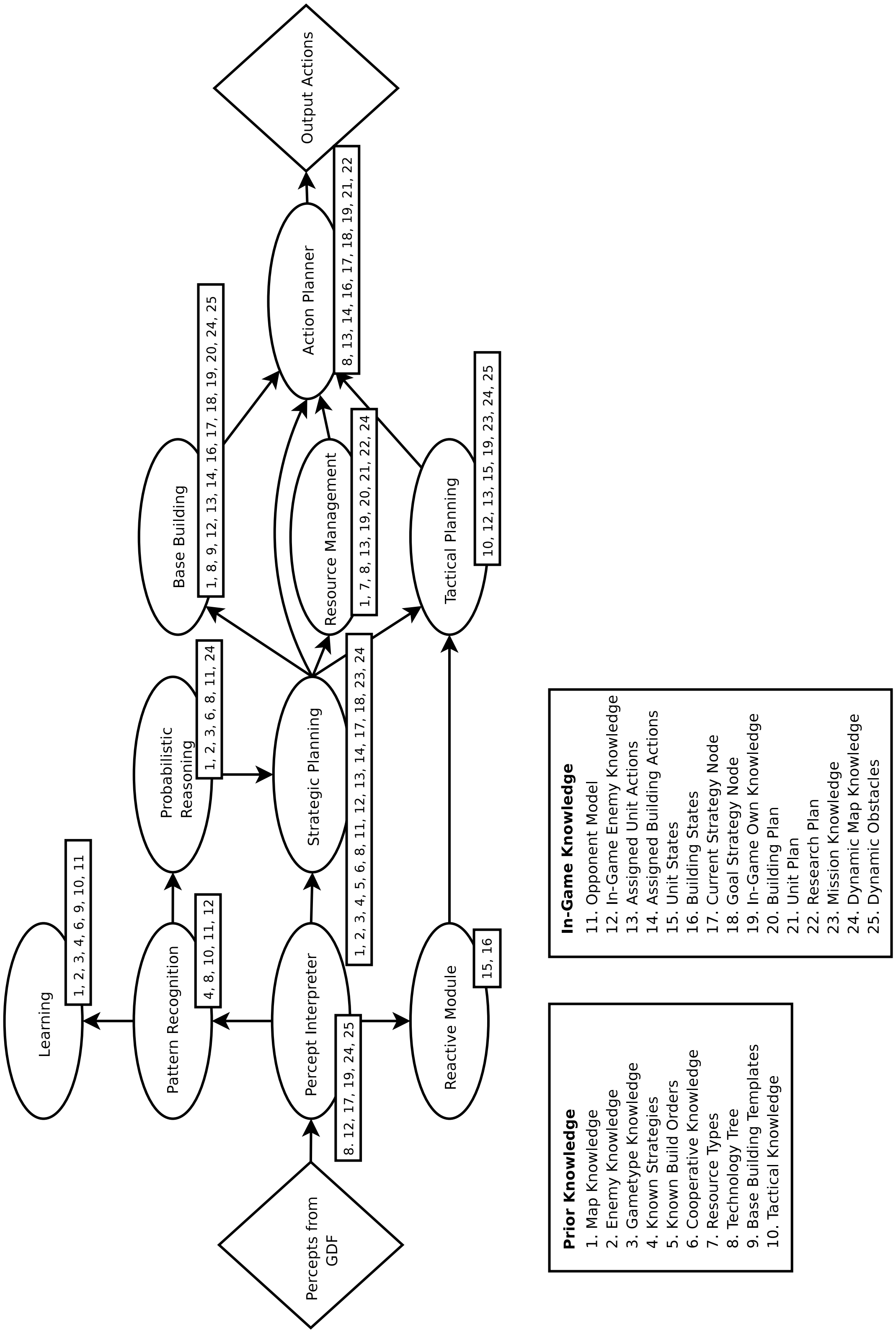
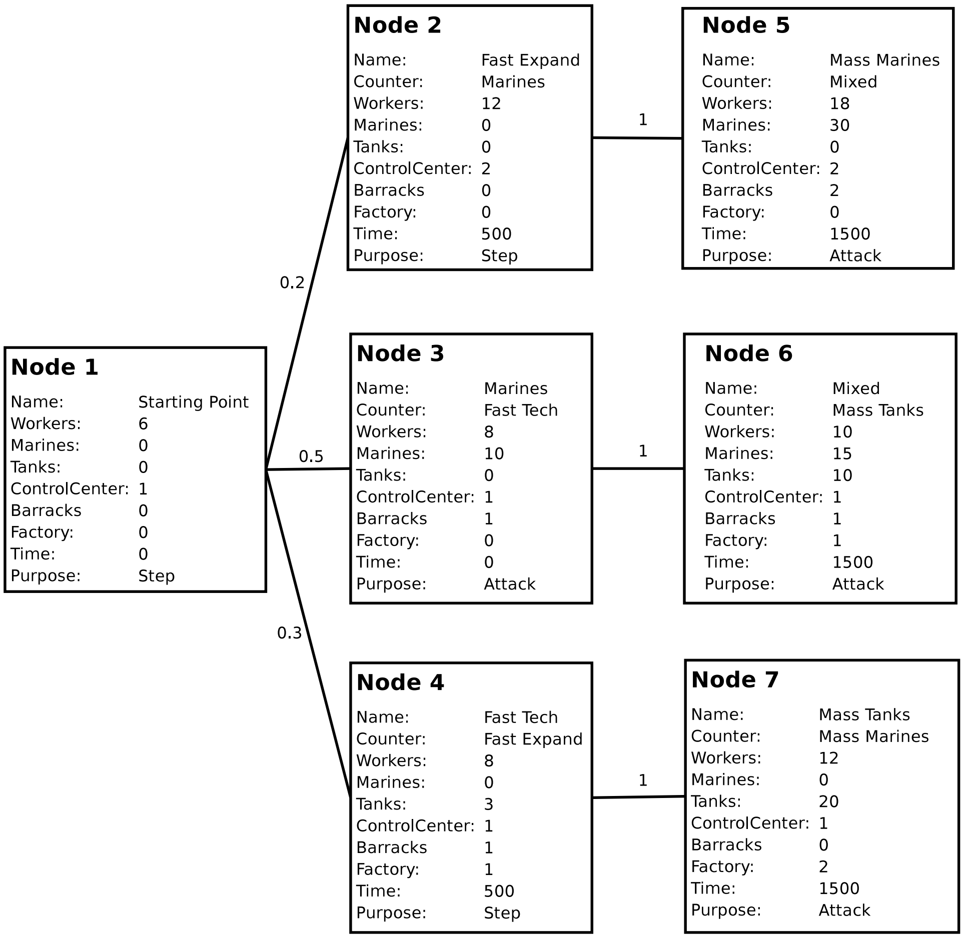


Figure K.2: The cognitive framework architecture



Node 1

Name:	Starting Point
Workers:	6
Marines:	0
Tanks:	0
ControlCenter:	1
Barracks	0
Factory:	0
Time:	0
Purpose:	Step

Node 2

Name:	Fast Expand
Counter:	Marines
Workers:	12
Marines:	0
Tanks:	0
ControlCenter:	2
Barracks	0
Factory:	0
Time:	500
Purpose:	Step

Node 3

Name:	Marines
Counter:	Fast Tech
Workers:	8
Marines:	10
Tanks:	0
ControlCenter:	1
Barracks	1
Factory:	0
Time:	0
Purpose:	Attack

Node 4

Name:	Fast Tech
Counter:	Fast Expand
Workers:	8
Marines:	0
Tanks:	3
ControlCenter:	1
Barracks	1
Factory:	1
Time:	500
Purpose:	Step

Node 5

Name:	Mass Marines
Counter:	Mixed
Workers:	18
Marines:	30
Tanks:	0
ControlCenter:	2
Barracks	2
Factory:	0
Time:	1500
Purpose:	Attack

Node 6

Name:	Mixed
Counter:	Mass Tanks
Workers:	10
Marines:	15
Tanks:	10
ControlCenter:	1
Barracks	1
Factory:	1
Time:	1500
Purpose:	Attack

Node 7

Name:	Mass Tanks
Counter:	Mass Marines
Workers:	12
Marines:	0
Tanks:	20
ControlCenter:	1
Barracks	0
Factory:	2
Time:	1500
Purpose:	Attack

Figure K.3: The strategy tree used for testing the AI

Appendix L

AI Development in Industry

L.1 RTS Companies Contacted

The following will list the companies we have attempted to contact, along with the RTS games they have developed:

Digital Realily: War Front: Turning Point
(http://www.digitalreality.hu/ - info@digitalreality.hu)

Stormregion: Codename: Panzers Phase One + Rush for Berlin
(http://www.stormregion.com/ - info@stormregion.com)

Big Huge Games: Rise of Nations
(http://www.bighugegames.com/ - info@bighugegames.com)

Relic: Homeworld 1+2
(http://www.relic.com/ - amy.farris@vugames.com)

Battlefront: Histway: Les Grognards
(http://www.battlefront.com/ - support@battlefront.com)

Mad Doc Software: Empire Earh I + II
(http://www.maddocsoftware.com/ - sotoole@maddocsoftware.com)

Inhuman Games: Trash
(http://www.inhumangames.com/ - info@inhumangames.com)

G2games: Alliance: Future Combat
(http://www.g2games.com/ - http://www.g2games.com/corporate.shtml)

Creative Assembly: Total War series
(http://www.creative-assembly.co.uk/ - info@creative-assembly.co.uk)

Evolution Vault: Galactic Dream
(http://www.evolutionvault.net/ - contact@evolutionvault.net)

Reality Pump: Earth 2160
(http://www.realitypump.pl/ - office@realitypump.pl)

Ensemble Studios: Age of Empires series
(http://www.ensemblestudios.com/
- webmaster@ensemblestudios.com)

Haemimont Games: Rising Kingdoms
(http://www.haemimontgames.com/ - info@haemimontgames.com)

Pyro Studios: Imperial Glory
(http://www.pyrostudios.com/ - pyrostudios@pyrostudios.com)

Cat Daddy Games: American Civil War: Gettysburg + Medievel Conquest
(http://www.catdaddygames.com/ - catdaddy@catdaddy.com)

GSC Game World: Cossacks
(http://www.gsc-game.com/ - anton@gsc-game.kiev.ua)

Enemy Technology: I of the Enemy
(http://www.enemytechnology.com/ - info@enemytechnology.com)

K-D Lab: Maelstrom
(http://www.kdlab.com/eng/ - contacts@kdlab.com)

Magictech: Takedo 1+2
(http://www.ezgame.com/ - magictect@ezgame.com)

Oddlabs: Tribal Trouble
(http://www.oddlabs.com/ - mail@oddlabs.com)

Black Hole Entertainment: Armies of Exigo
(http://www.blackholegames.com/ - info@blackholegames.com)

Fireglow Games: Sudden Strike
(http://www.fireglowgames.com/ - contact@fireglowgames.com)

Related Design: Castle Strike
(http://www.related-designs.de/ - info@related-designs.de)

Timegate Studios: Kohan
(http://www.timegate.com/ - inf-05@timegate.com)

Massive Entertainment: Ground Control I + II
(http://www.massive.se/ - info@massive.se)

Primal Software: Besiger
(http://www.primal-soft.com/en/ - info@primal-soft.com)

Infinite Interactive: Warlords Battlecry 3 + 4
(http://www.infinite-interactive.com/
- contact@infinite-interactive.com)

Independent Arts: Against Rome
(http://www.independent-arts-software.de/
- info@independent-arts-software.de)

Legend Studios: War Times
(http://www.lsgames.com/ - info@lsgames.com)

Lesta Studio: WWI: The Great War
(http://www.lesta.ru/ - serg@lesta.ru)

THQ: Supreme Commander + Warhammer: Dawn of Way
(http://www.thq-games.com/ - info.thq.com/support/generalsupport.asp)

CDV Software: Hidden Stroke + Cossacks 2: Napoleon Wars
(http://www.cdv.de/ - mail@cdv.de)

Strategy First: Nexagon: Deathmatch
(http://www.strategyfirst.com/ - info@strategyfirst.com)

Blizzard Entertainment: Starcraft and Warcraft series
(http://www.blizzard.com/ - support@blizzard.com)

The Bitmap Brothers: World War II: Frontline
(http://www.bitmap-brothers.co.uk/
- contact@bitmap-brothers.co.uk)

Rival Interactive: Real War: Roque States
(http://www.real-war.com/ - Jim.Omer@RivalInteractive.com)

Zuxxex: World War II: Panzer Claws
(http://www.zuxxez.com/ - info@zuxxez.com)

Pandemic Studios: Army Men
(http://www.pandemicstudios.com/ - info@pandemicstudios.com)

Object Software: Dragon Throne: Battle of Red Cliffs
(http://eng.objectgames.com/ - info@objectsw.com)

Electronic Arts LA: Command & Conquer Series
(http://westwood.ea.com/ - info@ea.com)

L.2 Mail to RTS Game Development Companies

Hi

In relation to our master-thesis developed at the department of computer science, Aalborg University we would like your help in answering a few questions concerning development of AI in the game industry. If this request was sent to the wrong department, please forward it to a person who can help us. We are writing this to your company, because you have a history of developing RTS games, which is the focus of our master-thesis.

We are currently developing an AI framework for RTS games based on the decision process of a human player. Knowing that different RTS games have different focus in game style, the modular design allows the developer to focus on the areas that are important for that particular genre. Less important modules can be left handled by standard implementations in the framework. We believe that using this framework for AI development will have the following effects:

- Structured overview of the AI development process.
- Significantly improve the AI.
- Reduced development cost.
- Reduced development time.
- Workload shifted towards game designers instead of programmers.

We hope that you will take a few minutes to answer the following questions. Please indicate how you base your answers/estimates - e.g. on your own experience or on the current practice in your company/development team.

1. How much time would you estimate is currently used on developing AI in RTS games - e.g. how many man-hours are used?
2. Who develops the AI? Is it programmers or game designers?
3. Is the AI created from scratch or are AI libraries used?
4. How connected are the game engine and the AI? Is it completely separated or closely integrated in the engine?
5. Do you think that our idea/product of an generic RTS AI framework could be of use in the industry? Why/why not?

Your answers will be used to get insight into the process and use of tools in AI development.

L.3 Answers from RTS Game Development Companies

All answers corresponds to the questions in Appendix L.2.

L.3.1 Oddlabs

Answers from Oddslabs were in danish:

1. Jeg ved ikke så meget om andre spils AI, men AI'en til TT er en meget simpel state-machine der på yderst naiv vis tager stilling fra tur til tur. Den har ikke taget meget mere end 3 uger at lave.

2. Den er udviklet af en programmør.

3. Helt fra bunden.

4. AI'en er meget stærk bundet til TT.

5. Jeg kan godt se potentiale i at have et generisk RTS AI framework, men jeg er ikke sikker på det vil virke i praksis. Det skal virkelig være let at gå til, og give nogle meget store fordele i form af kompliceret logik og lign., hvis man skal bruge tid på at integrere et tredjeparts system ind i sit spil, frem for selv at bygge noget ind, som er skrædersyet til situationen. I vores tilfælde havde det måske været smart da vi var nået til et punkt i udviklingen hvor vi måtte "nøjes" med en primitiv AI, fordi der ikke var tid til at gå i dybden med udviklingen. Til gengæld tvivler jeg på vi kunne have gjort det lige så hurtigt hvis vi skulle sætte os ind i et generisk system der samtidig skulle bankes ind i den struktur vi havde i spillet.

   Med andre ord, så kan jeg nok ikke sige om det er en god ide før jeg har set produktet.

L.3.2 Infinite Interactive

1. About 12 man-months in all of our RTS games so far.

2. Programmers develop an AI framework, based on a movement/pathing system. Then they work WITH the game designers to build and refine an AI. As we use more and more scripting (LUA is our language of choice), more and more AI is being by our designers rather than the programmers.

3. We have our own movement/pathing libraries on which everything is built. Everything apart from the movement and pathing is created from scratch on every game.

4. They are kept completely seperate. However, various functions of the engine have been added to help with AI, such as line-of-sight calculations.

5. Possible, but difficult to apply to *every* RTS game, because of the variations on design in each game. But I think that a limited framework would be useful, as long as all of the items were quite independent and quite easily extended: Some of the areas we break our AI's down into are:

   - Movement
   - Pathing
   - Formations
   - Influence maps (e.g. for detection of danger)
   - Threat assessment
   - Actions/orders
   - A state machine of actions of individual actors
   - Grouping mechanisms
   - A method for tracking and remembering enemies
   - Building and production hierarchies
   - Resource usage and needs
   - Managing and prioritizing objectives

   If an AI framework consisted of base classes for dealing with things like this, then it would indeed save time.

L.3.3 Inhuman Games

First off I would define classify AI into two groups: low level and high level. Low level AI mostly includes pathfinding, target selection, and misc actions (spell casting, loading/unloading resources, etc). High level AI includes deciding what to build and where to send your forces.

1. At least two man years. About half the time for low-level AI, and another half for high-level stuff.

   The hardest part of the low-level AI is probably pathfinding. Pathfinding itself can take a long time to develop, especially if you are trying to make good pathfinding that scales well.

2. Programmers tend to do most of the AI development. Increasingly game designers with scripting ability are developing AI. Game designers tend to only control very high level aspects of AI.

3. I believe they are usually created from stratch.

4. This probably varies greatly between projects. In the RTS, Trash, the high level AI and pathfinding are well seperated. Target selection is not seperated as well as it could be.

5. If your AI is the great, I think it could be sold. It would have to be extremely good and easy to integrate into any RTS game engine. If this was the case, perhaps you could charge $100k USD for it–if sold to big AAA studios.

L.3.4 Fireglow Games

1. It's difficult to make a precise estimation, because vagueness of frames of which part of the game engine is AI and which is not. About 5000 man-hours.

2. Both programmers and designers do, and even if consider a Virtual Machine to be not a part of the AI engine, there's a plenty of work done by programmers.

3. Partially our AI engine is based on the third-party Virtual Machine/Script System, but most part is written in-house.

4. Our AI engine consists of several modules, some of them are external, and some are closely tied with the game and gameplay.

5. As always, it would have some use. How much? It depens on the framefork's quality and its price. If the framework will contain necessary functions (most demanded are probably pathfinding and scripting language with virtual machine), and it is affordable, it will be used by developes.