Evaluating a Cognitive Agent-Orientated Approach for the creation of Artificial Intelligence in StarCraft

Tom Peeters

TU Delft

Evaluating a Cognitive Agent-Orientated Approach for the creation of Artificial Intelligence in StarCraft

by

Tom Peeters

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday August 28, 2018 at 3:00 PM.

Student number:	4176510	
Project duration:	April 1, 2017 – August 28, 2018	
Thesis committee:	Dr. ir. K. Hindriks	TU Delft, supervisor
	Dr. ir. J. Broekens	TU Delft
	Dr. ir. R. Bidarra	TU Delft
	Ir. V. Koeman	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Acknowledgements

After 25 years, this thesis marks the end of my education, it was a rough ride at times. In high school I suffered from motivation problems, caused by my disinterest towards various classes. My self-confidence ended up getting the better of me, which resulted in me being lowered a grade. It was a rough awakening, but a necessary one. Instead of the 6-year VWO route, I dropped down to the 5-year HAVO route. However, this allowed me to start on a computer science education a year earlier. And for the first time, I found myself interested in all classes being offered to me. Although my pride was shattered, I can only feel happy at being able to discover what I love doing a year earlier. After finding out what motivates me, my studies improved greatly, and now I find myself at the end of my Master course, having fought my way back up.

I want to thank my family for all the love and support. Especially my late mother, who passed away at the end of the 2nd year of my Bachelor. Without your support and understanding of my problems in high school, I am convinced that I would not be where I am today. Even today, you serve as a source of motivation and inspiration to me. And I'd like to thank my father and sister as well, who also supported me, provided me with input when I needed it, and put up with all my quirks.

Lastly, I'd like to thank everyone who helped me for this thesis. Koen Hindriks, for acting as supervisor and providing me with a topic for my thesis. Vincent Koeman, for putting up with my various requests and complaints over the course of the development. And all of the SSCAIT crew and other StarCraft bot developers, who are always eager to help.

And finally, I'd like to thank you: the reader. I hope that you find this thesis to be of interest and my bot to be entertaining.

Tom Peeters
Delft, August 2018

Abstract

In this thesis a cognitive multi-agent system is developed for the real-time strategy game StarCraft: Brood War using GOAL, a language developed at the Delft University of Technology. StarCraft provides various challenges to GOAL that it has not faced before in terms of complexity and quantity of agents. A connector is required for GOAL to interact with StarCraft: the StarCraft-GOAL Environment Connector. In order to provide context for the development stage, this thesis starts by explaining the basics of StarCraft, GOAL and the connector. The development of the bot that will be created, named ForceBot, takes place over the course of four tournaments that serve as milestones and are used to measure the game performance of ForceBot. As the conditions and participants of these tournaments vary, an additional setup for hosting internal test tournaments is established in order to provide a form of testing that remains consistent over the course of the development. Based on the results of tournaments, it can be concluded that the final game performance of ForceBot is above average with regards to existing StarCraft bots. A total of four aspects of GOAL, the language, tools, connector and computational performance, are analysed based on their usage and performance during development, and suggestions for improvements are made accordingly. Among these four aspects, the computational performance is identified as the primary bottleneck for the game performance of GOAL-based StarCraft bots.

Contents

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 GOAL . . . 3
1.3 Research Questions . . . 4
1.4 Approach . . . 5
1.4.1 Milestone 1: MAS Project . . . 5
1.4.2 Milestone 2: AIIDE . . . 5
1.4.3 Milestone 3: SSCAIT . . . 6
1.4.4 Milestone 4: SAIL . . . 6
1.4.5 Conclusions . . . 6

2 StarCraft Basics **7**
2.1 What is StarCraft? . . . 7
2.2 StarCraft Strategy . . . 13
2.2.1 Protoss Strategy . . . 14
2.2.2 Terran Strategy . . . 15
2.2.3 Zerg Strategy . . . 15

3 Literature **16**
3.1 Tournament results . . . 16
3.1.1 AIIDE . . . 16
3.1.2 CIG . . . 17
3.1.3 SSCAIT . . . 18
3.2 Agent-Based Artificial Intelligence in Games . . . 18

4 GOAL and Connector Basics **20**
4.1 GOAL . . . 20
4.1.1 GOAL Project Structure . . . 21
4.1.2 Messaging . . . 23
4.1.3 IDE & Tools . . . 24
4.2 StarCraft-GOAL Connector . . . 24
4.2.1 Connector Outline . . . 26
4.2.2 Global Static Percepts . . . 26
4.2.3 Global Dynamic Percepts . . . 26
4.2.4 Generic Unit Percepts . . . 27
4.2.5 Unit-Specific Percepts . . . 27

5 Milestone 1: MAS Project **29**
5.1 StarCraft-GOAL Connector Development . . . 29
5.1.1 Global Static Percepts . . . 30
5.1.2 Global Dynamic Percepts . . . 30
5.1.3 Generic Unit Percepts . . . 30
5.1.4 Unit-Specific Percepts . . . 31
5.1.5 Action Changes . . . 31
5.1.6 Connector Change Overview . . . 32
5.2 Design and Development of ForceBot: V1 - Prototype . . . 32
5.3 Design and Development of ForceBot: V2 - Beginning . . . 33
5.3.1 Designing a Manager . . . 33
5.3.2 Spending Resources . . . 33
5.3.3 Unit Deaths . . . 34
5.3.4 Tournament Results . . . 34

5.4 Design and Development of ForceBot: V3 - MAS Project . . . 36
5.4.1 Tournament Results . . . 36
5.5 Conclusion . . . 38

6 Milestone 2: AIIDE **39**
6.1 StarCraft-GOAL Connector Development . . . 39
6.1.1 Managers . . . 39
6.1.2 Subscribing to Percepts . . . 41
6.1.3 Percept Changes . . . 42
6.1.4 Global Static Percepts . . . 42
6.1.5 Global Dynamic Percepts . . . 42
6.1.6 Generic Unit Percepts . . . 43
6.1.7 Action Changes . . . 44
6.2 Design and Development of ForceBot: V4 - Manager . . . 44
6.3 Design and Development of ForceBot: V5 - Combat Simulator . . . 46
6.3.1 Combat Simulator: Goals and Challenges . . . 46
6.3.2 Combat Simulator: Version 1. . . . 47
6.3.3 Combat Simulator: Version 2. . . . 47
6.3.4 Combat Simulator: Version 3. . . . 48
6.4 Design and Development of ForceBot: V6 - Subscribe. . . . 50
6.5 Design and Development of ForceBot: V7 - AIIDE. . . . 50
6.5.1 Enemy Targeting . . . 51
6.5.2 Robustness. . . . 51
6.6 AIIDE Results . . . 51
6.7 Conclusion . . . 52

7 Milestone 3: SSCAIT **54**
7.1 Internal Testing . . . 54
7.2 Internal Testing Setup . . . 54
7.2.1 Internal Testing Results . . . 56
7.3 StarCraft-GOAL Connector Development . . . 57
7.4 Design and Development of ForceBot: V8 - SSCAIT . . . 57
7.4.1 Lurker Behaviour . . . 59
7.4.2 Start-Up Difficulties . . . 59
7.5 SSCAIT Results . . . 59
7.5.1 Internal Test Results . . . 60
7.6 Conclusion . . . 61

8 Milestone 4: SAIL **63**
8.1 Design and Development of ForceBot: V9 - Final . . . 63
8.1.1 Enemy Strategy Detection . . . 64
8.1.2 Profiling . . . 65
8.1.3 Drone Defence . . . 66
8.2 SAIL. . . . 66
8.2.1 SAIL Startup Problems . . . 67
8.2.2 SAIL Results . . . 67
8.2.3 Internal Testing Results . . . 70
8.3 Conclusion . . . 71

9 Development, Tools and Language **72**
9.1 IDE . . . 72
9.2 Debugging . . . 73
9.2.1 Debug Mode . . . 73
9.2.2 Logging . . . 74
9.3 Profiling & Optimisation . . . 75
9.4 GOAL Language . . . 76
9.4.1 Nesting of `not` . . . 76
9.4.2 Unused Features . . . 77

9.5 Conclusion 79

10 Lessons Learned **80**

10.1 Language 80

10.2 Connector 81

10.2.1 MAS Project 2018 81

10.2.2 Global Dynamic Percepts 82

10.2.3 Generic Unit Percepts 82

10.2.4 Unit-Specific Percepts 82

10.2.5 Connector Overview 82

10.3 Tools 83

10.4 Computational Performance 85

10.4.1 Prolog 85

10.4.2 Agent Scheduling 86

10.4.3 Synchronisation 86

10.4.4 Agent to Unit Mapping 87

10.4.5 Connector Abstraction 88

10.5 Conclusion 89

11 Conclusions **90**

11.1 Recommendation for Future Research 91

Bibliography **92**

Introduction

Almost every video game features Artificial Intelligence (AI). Starting with games as old as Pac-Man (1979), the occurrence of AI in video games steadily increased, to the point where it is difficult to find a modern game that does not have any. Through the use of AI, games can become more interactive and engaging. However, as with the rest of the game, it is important for AI to find a fine balance. If the game is too easy to beat, it can become boring. If the game is too hard to beat, it may become frustrating. Achieving this balance allows players to enter what is commonly referred to as 'the flow' [12]. When the player is constantly being challenged enough to avoid boredom, while also not being pressured enough to cause frustration, the player can stay in 'the flow', a state where the player is fully engaged with and immersed in the game.

Therefore, much work in game development is aimed at making sure that the game is balanced to give the player an enjoyable experience, this includes the AI. In fact, many games exist in which AI-controlled enemies are the main source of challenge provided to the player. It is not an exaggeration to say that, for these games, the creation of balanced AI is critical to the success of the game. This also means, however, that AI must exist which can beat humans of all skill levels in order to properly challenge any given player. Attempts to solve that have led to the creation of AI such as Deep Blue by IBM, which beat the at-the-time top human chess player Garry Kasparov $3\frac{1}{2} - 2\frac{1}{2}$ in 1997 [4]. As well as AlphaGo, an AI for Go whose most recent version, AlphaGo Zero, remains undefeated by human players [6].

Games like Chess and Go are known as fully observable games. This means that the complete state of the game is known to the players at all times. In such games, AI have seen a great level of success in recent years, often elevating themselves to the point of being unbeatable by human players. There has been less success in partially observable games, in which the player is not aware of everything that happens in the game at all times. One of the more successful AI in partially observable games is Deepstack in poker [30], which is able to beat professional human players, but has not yet managed to beat the best human players.

The game of poker is able to be solved to some extent however, as being a card game, a great deal of probability and prediction is present. There are also partially observable games in which this is not the case, such as Dota 2, a Multiplayer Online Battle Arena (MOBA) game. OpenAI saw success in Dota 2 in August 2017, convincingly beating the top professional players. However, it did so in a simplified version of the game, which restricted play to 1-on-1 combat, rather than the traditional 5-on-5, as well as using only a single character in the game, instead of the 113 characters available at the time.

Another partially observable game which has seen a great deal of attention in recent years is StarCraft: Brood War. StarCraft is a Real-Time Strategy (RTS) game developed and published by Blizzard Entertainment. In the game the player can pick from 3 races: Terran, Protoss and Zerg. Each race features a unique set of units, buildings and mechanics, giving rise to various different ways of playing the game. For example, one can choose to try and win in the early-game with a single small army, in the hopes of being able to take down the enemy before defences can be constructed. Other strategies focus on the late-game by playing defensively until they have constructed a powerful economy and a large army with which they can overwhelm the enemy with sheer force.

Roughly a decade ago, the Brood War Application Programming Interface (BWAPI) was released, giving third parties the ability to write powerful, custom AI for StarCraft. This has led to the creation of a large amount of StarCraft bots being built, capable of playing the entire game on their own. These various bots use various different strategies to try and beat even the strongest of players. A good example of this is the Berkeley

Overmind project [23], which succeeded in beating one of the top European StarCraft players [21]. But such victories are only on occasion, the strongest human StarCraft players have yet to be defeated by bots.

1.1. Problem Statement

Traditionally, bots are written with a single program in charge of all units. Traditional StarCraft bots are no different, with bots typically consisting of a singular process, although there are a number of bots which have split themselves into a handful of modules that are relegated to specific tasks such as combat, economy and exploration [32]. In this thesis, a different approach is taken to the challenge of creating StarCraft bots. Instead of using a single program, the aim is to approach the problem from a multi-agent perspective, with each unit being controlled by an independent cognitive agent. Although technically there is only one player in control in StarCraft, by assigning an independent AI to every unit present in the game, a multi-agent system can be used to tackle the problem.

An example of this is an attempt to utilise multi-agent systems for a chess bot [31]. In this chess bot, each chess piece has its own AI which can reason for itself. It determines its best action, both in trying to capture enemy pieces, as well as preserving itself. The bot can then decide which action is best to perform based on the moves offered to it by each of its pieces. A multi-agent system in StarCraft works in a similar way, although unlike in the chess bot there is no need to select a single move to perform – each unit may act independently.

However, a multi-agent approach using cognitive agents is not something which has been attempted before for games as complex as StarCraft. Although the aim of this thesis is to construct a bot that is capable of playing StarCraft with advanced proficiency, it is also the aim to find out whether this is even possible with the currently available knowledge and technology. In short, this thesis asks the question: *What are the benefits to using cognitive and multi-agent-based artificial intelligence techniques over the traditional single-agent approach for the creation of StarCraft AI*?

The environment of StarCraft provides various challenges that make it an excellent testing ground for a large number of fundamental AI research problems. These challenges can be outlined as follows [29]:

Adversarial real-time planning
In StarCraft, decisions are made in real-time, giving a continuous time constraint to the processing of the bot. Furthermore, opponents continuously interact with the game as well, modifying the game state asynchronously.

Decision making under uncertainty
Units in StarCraft possess a vision range. A limited sight range in which they can perceive enemy units. Anything outside of this range is covered by what is called the 'Fog of War'. This makes the game very different from a number of traditional AI research fields, such as Chess or Go, where the AI can perceive the full game state at all times. Fog of War turns StarCraft into a partially observable game, in which there is uncertainty as to what the state of the opponent is.

Opponent Modelling
It is important for players in StarCraft to both react to and predict actions by their opponent. Furthermore, by analysing the opponents' actions a player can potentially spot and exploit a weakness in their opponents strategy.

Spatial and temporal reasoning
Proper utilization of the map is an important aspect in StarCraft. For example, by placing units strategically on the high ground, players gain an advantage in vision, as well as combat, as units that are on terrain lower than the enemy are prone to missing their attacks. An example of temporal reasoning is how Zerg players will frequently save up Larvae to use simultaneously, allowing for sudden bursts of combat units to take the enemy by surprise.

Resource management
The management of the resources found in StarCraft, minerals, vespene gas and supply, is vital to successful play. The player must continuously make trade-offs: deciding whether to use their resources on building an army, upgrading their army, or increasing the power of their economy.

Collaboration
A player in StarCraft controls a large number of units. The army is often treated as a multi-agent system

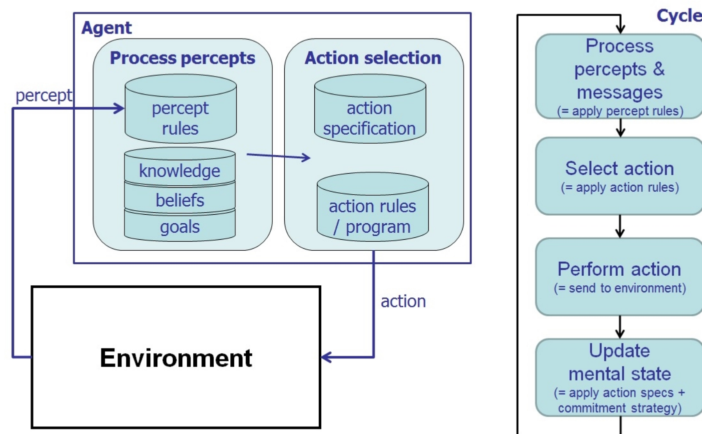


Figure 1.1: The interaction between environment and GOAL, and the structure of cycles

even in bots which are not designed as such. Units need to be able to take individual action, but must also coordinate with one another in order to function successfully as an army.

Pathfinding
Units in StarCraft continuously move around, and players may place defensive building to ward off enemies. This leads to path-finding in StarCraft being a task more complex than finding the shortest route. Often times, players will want to find alternate routes to get around dangerous areas and keep their units safe.

1.2. GOAL

For the development of a cognitive multi-agent system for StarCraft, GOAL [20] will be used. GOAL is a rule-based programming language for programming cognitive agents that interact with an environment and with each other. Information about the environment is received through percepts, and agents can act upon the environment using actions. Agents are part of a multi-agent system and can exchange information between each other through messages. Agents maintain a cognitive state that consists of knowledge, beliefs and goals of the agent which are represented in some knowledge representation language. Agents are autonomous decision-making agents that derive their choice of action from their beliefs and goals. Agents operate in cycles. In each cycle, first new percepts and messages are received, then the actions are selected and performed, and finally the mental state of the agent is updated.

GOAL is developed at the Technical University of Delft and has been used in various experimental and game environments before, most notable of which being Unreal Tournament [19]. GOAL utilises the Environment Interface Standard (EIS) [18], a Java-based interface standard for connecting agents to controllable entities in an environment. In June 2016, Harm Griffioen and Danny Plenge created the StarCraft-GOAL Environment Connector. This connector creates a bridge between BWAPI and EIS-enabled multi-agent systems, allowing for the creation of GOAL-based StarCraft bots.

StarCraft can be modelled as a multi-agent system within GOAL by using a 1-on-1 mapping of agents and units/buildings within StarCraft. Every agent will control only the unit or building that it is connected to, and use messaging between agents in order to coordinate, forming a multi-agent system. GOAL is well-suited to the aspect of partial observability in StarCraft, as GOAL's percept-based design is centred around the concept of partial observability. Previous environments of GOAL such as BW4T [22] feature this, where agents may only perceive the blocks in the room they are currently located in. Agents in Unreal Tournament also cannot perceive enemy players outside of their line of sight.

In order to model the game state of StarCraft as percepts, a level of abstraction is applied by the connector. Abstraction is a technique that has frequently been used to develop game AI, such as for Poker [36]. By simplifying the game state, a problem that is easier to solve is created. However, some amount of information is lost in abstraction, which can negatively affect the game performance of the AI. One of the tasks for this thesis will be to determine whether the level of abstraction employed by the connector is helpful to the development of a bot for StarCraft.

StarCraft provides a challenge to GOAL in terms of number of agents, as well as complexity. Previous

environment in which GOAL has been used always contained either few percepts, few agents, or both. For example, in Unreal Tournament, ignoring percept types which are perceived only upon agent startup, only the following percept types can be received more than once per cycle:

- `weapon` – A player can hold a maximum of 9 weapons.
- `item` – It is rare for there to be more 2 items visible at once.
- `bot` – Typical matches involve only 8 to 16 players, of which only a few will be visible at any time.

This means that the number of percepts that a single agent is likely to receive within a single frame will usually be below 20. In StarCraft, however, more than a hundred player-owned units can be on the field simultaneously. Each of these units has its own agent attached, which can all perceive any friendly and enemy units visible on the map. The quantity of agents also plays a role in the complexity, as previous environments in which GOAL has operated would typically feature no more than 10 agents. There are also more strategic elements in StarCraft, such as coordinating and planning the collection and spending of resources found on the map. The increased levels of coordination and planning that are required for StarCraft will provide new challenges to GOAL.

Finally, a multi-agent system is a type of distributed system, which are known to be difficult to debug [2]. To assist in this process, GOAL provides a number of tools to be used for development, debugging and profiling of GOAL bots [25]. As previous environments in which GOAL has been used use significantly fewer agents and feature less complexity than StarCraft, another objective of this thesis is to assess the effectiveness of the tools provided by GOAL in development for a StarCraft bot.

1.3. Research Questions

This section expands on the challenges discussed in the previous sections, and aims to split the challenges into research questions that will be answered over the course of this thesis. The main objective of this thesis is to determine how well GOAL is able to perform in an environment such as StarCraft. This means that the language must be analysed in order to determine whether it is suitable for an environment such as StarCraft, and furthermore whether it provides the computational performance to operate in a complex real-time environment. In addition to this, the StarCraft-GOAL Connector applies a layer of abstraction on StarCraft in order to operate with GOAL. The existing level of abstraction may need to be adjusted in order for a GOAL-based StarCraft bot to perform better. Finally, GOAL provides a number of tools to assist in the development of a multi-agent system. These tools will need to be analysed as well in order to determine whether their current design of these tools is useful for developing, debugging and profiling a StarCraft bot, as well as to determine areas for improvement. Based on these concerns, the following four questions are raised:

What are the advantages and disadvantages of using GOAL for an environment such as StarCraft?
GOAL is a rule-based programming language based on cognitive perception. Traditional StarCraft bots are programmed using object-orientated programming languages such as C++ and Java. It may be that GOAL is not a suitable language for this task. In order to determine this, the advantages and disadvantages of GOAL over languages such as C++ and Java when developing a StarCraft bot must be assessed.

How does the abstraction of the StarCraft-GOAL connector affect GOAL bots?
The connector performs abstraction both in order to interact with GOAL, as well as to simplify the creation of a GOAL bot. However, if the connector abstracts the environment too much, a GOAL bot will become limited in its options. What are the advantages and disadvantages of the level of abstraction placed upon StarCraft by the connector?

How well do the tools provided for GOAL provide support for developing a StarCraft bot?
GOAL provides a number of tools to be used for development, debugging and profiling of GOAL bots. As testing and debugging multi-agent systems is difficult, it needs to be determined whether the testing environment for the bot is sufficient to efficiently find and remove bugs in a StarCraft GOAL bot. What are the pros and cons of each tool? Are any new tools required? How can existing tools be improved?

How does computational performance scale with the quantity of agents and percepts in GOAL?
GOAL utilises a 1-on-1 mapping of units/buildings to agents for StarCraft, which allows for over a hundred agents to be active at once. GOAL has not been used before in an environment featuring such

a large number of agents or percepts. Is a 1-on-1 mapping a feasible approach for tackling such an environment?

These four questions can be summarised as questions regarding the **language**, **connector**, **tools** and **computational performance** respectively. These four points will be returned to over the course of this thesis in order to assess how well each of these aspects of GOAL perform within the StarCraft environment.

Finally, in addition to these GOAL-related questions, a final question is: *how does a cognitive multi-agent system compare to traditional StarCraft bots?* There are no existing cognitive multi-agent systems for StarCraft, although cognitive and multi-agent systems have been attempted before separately, which is explained in more detail in Chapter 3. At the end of this thesis this question will be looked at once more in order to determine an answer to this question.

1.4. Approach

This section discusses the setup for developing a multi-agent bot using GOAL for StarCraft: Brood War, which has been given the name 'ForceBot'. In order to draw any conclusions on ForceBot's strength as a StarCraft bot, it will need to compete against existing bots so that we may measure its game performance. In particular, competition against non-GOAL bots is preferred, as almost all existing StarCraft bots operate using C++ or Java. For this reason, ForceBot competed in a number of tournaments over the course of its development, spanning from April 2017 to March 2018. These tournaments have been split into a total of four milestones: the MAS Project, AIIDE, SSCAIT and SAIL – each of these will be briefly outline in this section.

By splitting the development up into milestones, focus can be placed on improvements for each particular milestone, and efforts can be made in order to resolve flaws or weaknesses of the previous milestone. An overview of all the milestones can be seen in Figure 1.2.

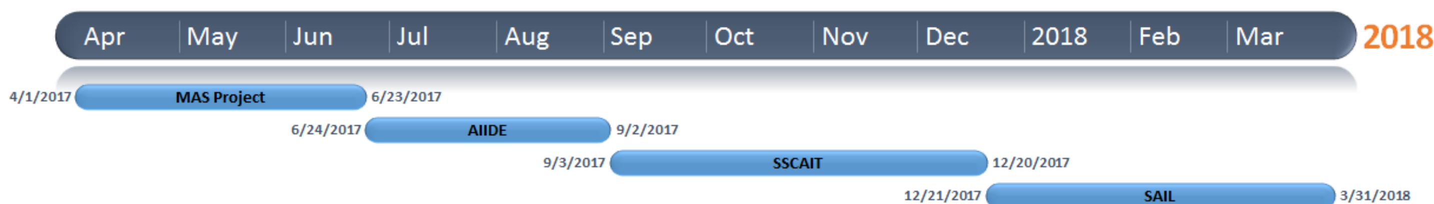


Figure 1.2: Timeline of the milestones

Before the development and results of the milestones are detailed, Chapter 2 will explain the basics of StarCraft, common terms used, as well as basic strategies. Chapter 3 will review the literature of multi-agent and cognitive AI, as well as the development of StarCraft AI thus far. Finally, Chapter 4 explains how GOAL operates, the environment and tools provided by GOAL, as well as the StarCraft-GOAL Environment Connector that it requires in order to interact with StarCraft.

1.4.1. Milestone 1: MAS Project

The first milestone in ForceBot's development life is the Multi-Agent Systems (MAS) Project, which will be covered in Chapter 5. The MAS Project is a course that takes place in Delft during the fourth quarter of the study year. In this course, 36 teams of 6 first year Bachelor students work together to create a StarCraft bot over the course of roughly 8 weeks. During this time, students can voluntarily enter periodic tournaments hosted by the instructors, allowing the student teams to measure the game performance of their bots. Work on ForceBot started at roughly the same time as the MAS Project. Participating in the periodic tournaments made for an excellent means of measuring the game performance of ForceBot and testing how it held up against the various tactics that the MAS Project teams could come up with. The last periodic tournament was held on June 22nd, 2017.

1.4.2. Milestone 2: AIIDE

AIIDE is an annual tournament StarCraft AI tournament. In 2017 it was held during September, concluding shortly before the actual Artificial Intelligence and Interactive Digital Entertainment (AIIDE) conference in October. The submission deadline for this tournament is September 1st, roughly two months after the MAS Project milestone. As AIIDE is the largest tournament of the year, attaining a high score in this tournament

is challenging. The development leading up to AIIDE, as well as the results achieved within this tournament, will be covered in Chapter 6.

In addition to AIIDE there is the Student StarCraft AI Tournament (SSCAIT). SSCAIT is a 24/7 stream on Twitch where anyone can freely submit bots to participate in the continuous bot-on-bot action. Notably, SSCAIT features a wide variety of bots, many of which are present in the annual AIIDE and CIG tournaments. By participating in this live stream the game performance of ForceBot against non-GOAL bots was measured. Furthermore, SSCAIT itself has a ladder system which produces an Elo rating, as well as an ICCUP Formula. These statistics were tracked and used as means to measure ForceBot's game performance over time. Entry into SSCAIT has no deadline, as one is free to enter at any time. Therefore, ForceBot participated in the SSCAIT ladder prior to AIIDE in order to help in the development towards AIIDE.

1.4.3. Milestone 3: SSCAIT

In addition to the 24/7 stream, SSCAIT holds an annual tournament, which in 2017 was held starting December 20th, the development and results of which will be covered in Chapter 7. Notably, this tournament distinguishes between bots developed by single students and bots developed by non-students. As the SSCAIT tournament was held further into ForceBot's development life than AIIDE, as well as the competition within this tournament being less fierce, SSCAIT is the tournament in which ForceBot should perform well.

Additionally, in the months between milestone 2 and 3, the SSCAIT stream provided information regarding ForceBot's game performance, as it played against other bots several times each day. However, ideally, a larger number of matches are played in order to reduce the margin for error when measuring game performance. In order to achieve this, internal test tournaments were hosted, held using local computers. In these tournaments, ForceBot was matched against bots present in AIIDE or SSCAIT, as these bots are freely available for use. By establishing a 'baseline' using nine bot opponents and periodically running these local tournaments, it was possible to measure the effectiveness of changes and ensure that the changes did not harm ForceBot's game performance against certain bots. Finally, this setup allowed for testing specific matchups. For example, by adding a bot that frequently does early rush attacks to the baseline, improvements to ForceBot's response to such tactics were measured.

1.4.4. Milestone 4: SAIL

The development and results after SSCAIT will be discussed in Chapter 8. No further tournaments were set to take place after SSCAIT. Initially, the period after SSCAIT would only be used to address weaknesses exposed during the SSCAIT round robin tournament, as well as assess ForceBot's game performance using internal test tournaments. However, the StarCraft Artificial Intelligence League (SAIL) began operating in March, 2018. SAIL operates similarly to the SSCAIT ladder: bots are randomly matched against one another 24/7. But games are played significantly quicker than on SSCAIT as a result of the games not being live streamed on Twitch. Furthermore, SAIL launched after completion of the final version of ForceBot. This meant that SAIL was ideal in order to measure the game performance of the final version and determine whether the changes that were made as a result of findings from SSCAIT were effective.

1.4.5. Conclusions

Upon completion of the development of ForceBot, the results and development process were analysed in order to determine answers to the questions posed in this chapter. In Chapter 9, the tools used during development, as well as the development itself are analysed. Afterwards, the findings that were made with regards to GOAL, the tools provided by GOAL and the StarCraft-GOAL Environment Connector are reviewed in Chapter 10. Finally, we will look back at the questions asked in this chapter in order to answer them in Chapter 11.

2
StarCraft Basics

StarCraft is a Real-Time Strategy (RTS) game created by Blizzard Entertainment and released in 1998. Players in StarCraft can select 1 of 3 races: Terran, Protoss and Zerg, with each race offering a unique play-style. The technologically advanced alien Protoss race has strong but costly units. The Zerg is a biological hive-mind composed of many cheap but weak creatures. Finally, the human race of Terrans balances strength and cost. The game of StarCraft has seen much success, selling over 11 million copies as of 2017. Over the years, Blizzard Entertainment has continued to balance the game, ensuring that all of the races are equal in strength. This chapter aims to provide an introduction and overview of StarCraft in order to provide knowledge about the environment that ForceBot will be designed for. First, in Section 2.1 the mechanics of StarCraft are explained, as well as the common terms used within the game. Section 2.2 takes a closer look at some of the strategies that may be used within StarCraft, and how a typical match might play out.

2.1. What is StarCraft?

In StarCraft, players start with a single base building and four worker units. Using worker units, players can collect additional resources: minerals and vespene gas. These resources can be used to construct more workers, buildings and combat units as well as researching or upgrading technology. This section aims to shed light on the common StarCraft terminology that is used in this thesis.

Map

StarCraft features different maps on which games are played. Maps vary in size and layout, which can favour certain strategies. For example, maps with short distances to the enemy base favour aggressive strategies, while maps with small entrances into the base favour defensive strategies. The maps most frequently used in competitive play have been balanced over the years using the feedback of players, in order to avoid giving major advantages to any particular race or strategy.

Base

The base building is different for each race: the Protoss base building is called Nexus, the Terran base building is called Command Center, and the Zerg base building is called Hatchery. The main purpose of all base buildings is the same, namely to train worker units and receive harvested resources. Base buildings are typically only constructed at a 'base location', which are specific places on the map where resources are bundled together for the player to harvest. Zerg is an exception to this, as the Zerg trains not just workers from Hatcheries, but almost all of its units. This is done by morphing Larvae into the desired unit. Larvae spawn periodically from any Hatchery, with a maximum of 3 Larvae per Hatchery. This often results in Zerg players constructing multiple Hatcheries in order to be able to train more units simultaneously. Furthermore, the Zerg may morph their Hatchery into a Lair and finally into a Hive. These morphed buildings act the same way, but unlock additional technological options.

A typical base location is seen in Figure 2.1, with Drones collecting resources, a vespene geyser at the top, and resources being displayed in the top-right. On every map there are a number of special base locations known as starting locations. At the start of each game, the player will start on a random starting location. Notably, if there are more starting locations than there are players present in a match, then a player can only become aware of where an opponents' base is located by exploring the map.

Figure 2.1: The start of a typical StarCraft game as Zerg

Expanding

When a player expands, it means that they are constructing a new base building at an unused base location. Although constructing a new base building is expensive, by expanding, additional resources become available to the player, allowing for faster resource collection at a later stage of the game. Most maps feature a so-called 'natural expansion', or simply 'natural'. A natural expansion is another base location located close to the starting location. In many maps the natural expansion serves as a 'natural' blockade before enemy forces can enter the main with ground forces.

Minerals

Minerals are the first of three player resources found in StarCraft. At least some amount of them is needed in order to create any unit or building. Minerals are mined using worker units, which can then deliver the harvested minerals to the base building. Each base location starts with a certain number of mineral fields, which each contain a limited number of mineral resources.

Vespene Gas

Vespene gas is the second type of player resource. This resource is not immediately available: in order to harvest it from a vespene geyser, a Protoss Assimilator, Terran Refinery or Zerg Extractor belonging to the player must be placed on top of a vespene geyser. Once constructed, worker units are able to collect vespene gas from it and deliver the vespene gas to the base building. Because of this requirement, the most basic units and buildings do not require vespene gas to be created, while more advanced units and buildings require larger amounts of vespene gas. Most base locations contain a single vespene geyser, although there are also base locations which contain zero or two vespene geysers instead.

Supply

Supply is the third type of player resource. Unlike other resources, this is not collected by workers, but received by construction a specific type of building or unit. Terrans must construct Supply Depots, Protoss must construct Pylons, and Zerg must train Overlords. When these buildings or units are destroyed, the supply is lost, and new ones must be created. Overlords are slow-moving flying units, rather than buildings. Overlords cannot attack, and are used to explore the map in addition to providing supply.

All units other than Overlords have a supply cost, and cannot be created unless there is enough supply available. When a unit is killed, its supply cost is returned to the owner of the unit. There is a maximum of 200 supply, which limits the number of units that a player can own.

Training & Morphing

Training is the process of spending resources to create new units for Protoss and Terran. This must be

done using a production building, and there are multiple types of production buildings that can train different units. Training a new unit takes a pre-determined time depending on the unit being trained, and production buildings can only train a single unit at once.

Morphing is similar to training, but is only used by the Zerg race. Zerg must morph existing units into new ones in order to create new units or buildings. There are 3 types of morphing:

1. Drones may morph themselves into buildings.
2. Buildings may morph themselves into more advanced versions in order to unlock additional technology. A Hatchery may morph into a Lair, and a Lair may morph into a Hive.
3. Certain units may morph themselves into other units. Mutalisks may morph into a Devourer or Guardian, while Hydralisks may morph into a Lurker. Every other Zerg unit is created by having a Larvae morph into the desired unit.

Regions & Chokepoints

The layout of each map can be split into regions and chokepoints. Regions are wide open areas, while chokepoints are narrow passages which connect regions. In most cases, one must pass through a chokepoint in order to travel between regions. There are also cases where a region is not connected to any other region, these regions are called ‘islands’ and can only be reached using flying units. Islands typically contain resources that are more difficult to reach, as flying is required, but as a result island bases are also easier to defend. A number of maps block off chokepoints using destructible objects or mineral fields, forcing players to clear them before the chokepoint can be passed through. This is seen on the bottom chokepoint in Figure 2.2

Figure 2.2: Regions being connected by chokepoints

Workers & Construction

Worker units are responsible for harvesting resources as well as constructing buildings, with each race performing these tasks in different manners.

- The Protoss need only to place a building, which will then fully construct itself. This allows a single Probe to quickly construct a large number of buildings. However, Protoss buildings other than the Nexus, Assimilator and Pylon cannot be constructed without a nearby Pylon to power these buildings.
- The Terran Space Construction Vehicle (SCV) can only construct a single building at any time, but is also able to repair both buildings and mechanical units.
- The Zerg Drone, rather than constructing a building, morphs itself into a building. Although this means that the Drone is lost in the process, the Zerg are able to use their Larvae to create new workers faster than the other races.

Build Order

A build order is a specific sequence of actions performed at the start of a game, generally leading into a specific strategy. For example, a more economic build order would initially focus on training workers and gaining a resource advantage so it may build a large army. Where as an aggressive build order will train minimal workers and quickly build a small army, aiming to end the game before the opponent can effectively defend themselves. Build orders are only used at the start of games, as the need to adjust actions based on the actions performed by the opponent increases as the game progresses.

Build orders are well-defined thanks to many years of competitive play by human players, with many guides being available online for a large variety of build orders. However, it is important to adapt the build order according to the map, enemy strategy and current situation. As a result, build orders are primarily followed during the early stages of games, when not much is known about the opponent.

Health, Shields and Energy

These three are the resources of any given unit. When a units' health reaches 0, it dies. Health does not regenerate for both Terran and Protoss units. Zerg units slowly regenerate health at all times. Shields are a layer of defence before Health, possessed by all Protoss units. Shields regenerate slowly, at around twice the rate of the health regeneration of Zerg units. Finally, energy is used to cast abilities. The effects of abilities vary, such as damaging units in an area or turning invisible. Not all units have abilities, and not all abilities cost energy. Units which have energy are called 'spell-caster' units. Spell-caster units primarily fight using abilities, as most spell-casters cannot attack normally.

Attacking & Attack Cooldown

Almost all units can attack other units. Each unit type has a different attack that deals a specific amount of damage at a specific attack cooldown. The attack cooldown of a unit refers to the time it requires between attacks. During the attack cooldown, a unit may move and act freely, but it cannot perform another attack until the attack cooldown has completed.

Ground & Flying Units

StarCraft has both units which are grounded, and those that can fly. Flying units are able to pass over terrain and obstacles, giving them an advantage in mobility. A number of units can attack both ground and flying units, however many units can only attack one or the other.

Micro & Macro

In StarCraft, micro and macro are almost like opposites to each other. Micro refers to finer unit controls, such as managing your army and individual units. Micro is making the most of out units by controlling them as precisely as possible. With excellent micro, it is possible to win fights that you would not win otherwise, or have more units survive.

Macro refers to the overall flow of the game and economy. Training new units, performing upgrades, constructing buildings and expanding are all considered macro. This includes strategic choices, such as deciding to construct defences or counters according to what the player believes the opponent is planning. A player who is poor at macro will have excess resources, which is called 'floating' resources, when these resources could be spent to train more units and other macro actions. Poor macro can also lead to the opposite, where the player has more production buildings than the player has resources to use, leading to idle buildings and therefore wasted resources.

A human player who is too focused on micro may forget to use his resources while managing his army, while a player who is too focused on macro may lose some units in his army that could have been saved with better control. It is important for StarCraft players to balance micro and macro. For bots this topic is a little different, as bots have no problem splitting their attention between micro and macro when disregarding computational performance constraints.

Kiting

Kiting is a form of micro. Kiting is when a unit utilises the time between its attack cooldown to run away from enemy units. In fights between two units, kiting is almost always a good strategy for one of the two units, typically the unit that can move faster or has a longer attack range. By running away during the attack cooldown, the enemy unit must run after the kiting unit and be unable to attack. Using kiting, units can reduce the damage taken during combat with other units.

Fog of War

Every unit in StarCraft has a sight range, a limited range that they are able to observe. The parts of the map which are not within the sight range of any of your units is called the Fog of War. Players cannot see enemy units that are located within the Fog of War. Enemy buildings can be seen, but only if the player has had sight of them before. The element of Fog of War adds an element of uncertainty to the game, making StarCraft a partially observable game. The partial observability aspect of StarCraft is a major challenge in the development of bots.

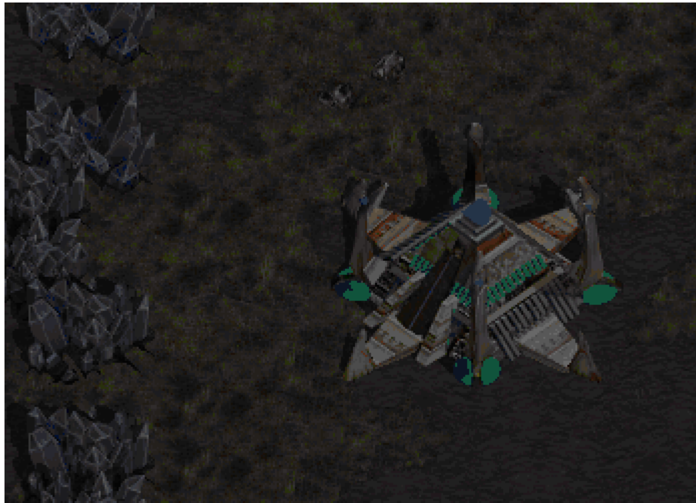


Figure 2.3: An enemy Nexus shrouded in the Fog of War

Scouting

Scouting is a common term in StarCraft used to denote when a unit travels into enemy territory in order to gather information. Most of the time, scouting is performed either by fast or flying units which can avoid enemy forces. Cheap units are also used in some cases, in order to minimize losses if they are destroyed.

Scouting is a vital part of StarCraft gameplay. For example, knowing the buildings that are located in the opponents base can tell players what strategy the opponent is using, allowing a counter strategy to be formed. Knowing the location and size of the opposing army allows players to create suitable defences in advance.

Burrowing, Cloaking and Detection

Certain units have the ability to cloak or burrow, turning them ‘invisible’ and making the opponent unable to attack them. Burrowed units cannot move, and is an ability that can only be used by Zerg units. Units with burrowing or cloaking capabilities are typically unlocked at later stages of the game. There are strategies that aim to quickly create a number of invisible units before the opponent has a means to counter them.

Invisible units can be countered using ‘detection’, which will reveal invisible enemy units, allowing them to be attacked. Each race possesses one static and one flying means of detection:

- Protoss may build the flying Observers, a non-combat unit which is cloaked itself, as well as the static defence Photon Cannon.
- Terran may train the flying Science Vessel and the static defence Missile Turret. Additionally, their base building can be upgraded with Comsat Stations, which can periodically be used to reveal invisible units in a chosen radius anywhere on the map.
- Zerg Overlords may detect invisible units, as well the static defence Spore Colony.

Rushing

Rushing refers early game, aggressive strategies that aim to end the game as soon as possible. In these strategies, economic power is traded in for creating a larger or faster army. This often results in rushing

strategies losing the match unless significant damage is dealt to the enemy. Particularly aggressive strategies, even amongst rushing strategies, are often referred to as 'cheese'. These cheese strategies often try to win using unorthodox strategies, such as using worker units or static defences to fight, rather than a more standard army.

Static Defence

A means of defending oneself in StarCraft is through the construction of static defences. Static defences are cheap and cost-effective, but unable to move. This largely limits these buildings to purely defensive tools, although strategies to use static defences offensively do exist. Static defences are designed to allow players who have the weaker army a means of defending themselves. The placement of static defences is a crucial part of playing the game effectively, as poorly placed static defences can be circumvented by clever movement from the opposing army.

Figure 2.4 shows an expansion defended by Sunken Colonies, with additional units morphing as eggs to the left of the Hatchery.

Figure 2.4: An expansion defended by static defenses

Runby

A runby is when enemy static defences are ignored, and the players' army runs past them in order to attack the base. This is possible if the opponent is relying mostly on poorly placed static defences.

Sim City

Sim City refers to the placement of buildings. By constructing buildings close together, a player can add to the defensive capabilities of locations, or even block paths entirely. When a path is completely blocked it is referred to as a 'wall-in'. The importance of Sim City differs for each race. For Protoss, building placement is a vital part of gameplay due to the need to power buildings using Pylons. For Zerg, it is a minor part of gameplay, as Zerg constructs fewer buildings than other races.

An example of this can be seen in Figure 2.5. A static defense, a Photon Cannon, is surrounded with buildings in order to make it difficult to attack or enter the base.

Figure 2.5: SimCity being used to fortify a Photon Cannon

Upgrades & Researches

Upgrades and researches increase the strength of units in StarCraft. Particularly as a match reaches the later stages, ensuring that you do not fall behind in upgrades is vital to taking the upper hand in a fight. There are 3 different types:

1. **Numeric upgrades** – Numeric upgrades increase either the attack or armour of certain unit types, e.g. 1 additional armour for all flying units. Each numeric upgrade has a total of 3 ranks, making fully-upgraded units significantly more powerful.
2. **Ability upgrades** – These unlock new abilities for units. Most spell-caster units have several ability upgrades.
3. **Passive upgrades** – Passive upgrades increase a specific unit stat, such as movement speed or energy capacity.

2.2. StarCraft Strategy

This section aims to provide an overview of how a typical StarCraft match may play out, strategies that may be used, as well as discuss some of the strengths and weaknesses of each race.

Each StarCraft match starts with a single base building and four worker units. It is possible to immediately begin offensive actions through strategies such as SCV rushes, the worker units for Terrans, or using a 4-pool strategy for Zerg, which aims to create Zerglings as fast as possible. A number of bots specialise in such strategies, however these types of extremely aggressive strategies are not frequently used. This is because the player will be at a major disadvantage if the first attack is unsuccessful.

In most cases, both sides will continuously construct additional worker units in order to strengthen their economy for the first part of the match. The order in which units and buildings are constructed during this time will follow a build order. The build order selected at the start of a match determines the strategies that are available to a player in the middle stages of the match. As such, selecting a build order that is appropriate for your enemy is a major factor in winning a match. Many bots feature build order learning as a result, in which the bot will learn which of its build orders are effective against specific opponents, recognised using their names.

In order to avoid being caught off-guard by enemy forces before any defensive measures have been taken, it is a common strategy to use one worker unit to scout the enemies base, in order to determine the level of aggression of the opponent and react accordingly. If neither side shows signs of aggression, both sides may opt to expand to multiple bases before any attacks are made. Once the middle and later stages of the game have been reached, there are a number of major factors that players will be looking out for.

Army vs Economy

Choosing the level of aggression is an important factor. Devoting all resources into producing combat units while the enemy is not expecting this can win a match. On the other hand, if the enemy is prepared, a game may be lost as a result of an economic disadvantage after devoting a large amount of resources into training an army. Making good decisions is difficult even for professional human players, and many bots perform poorly in this area.

Army Composition

Each unit has its own strengths and weaknesses, making it important to have a balanced army composition. Players will also want to create units that fight effectively against the opposing army, countering the opposing army composition. To this end, a number of bots use learning systems in order to determine appropriate army compositions based on the enemy units which it has seen.

Map Control

Players will look to take control of the map, taking resources for themselves and blocking the enemy from resources. By preventing the opponent from gaining map control, also referred to as 'containing (the enemy)', a player can gain an economic advantage while preventing the enemy from gaining any. If successful, this strategy can be used to overwhelm the opponents defences via an economic advantage over time.

There are a number of common differences between human and bot playstyles and strategies. Professional human players play more strategically and show greater ability at predicting and responding to enemy actions that occur within Fog of War (i.e. which cannot be observed). In order to predict enemy actions, many bots utilise build order learning, in which the bot learns which build orders are effective against which opponents by either name or race. This reduces the need for the bot to determine the opponents strategy during the game itself, as the opponent can be countered to an extent by using a build order that is strong against the strategy used by the opponent. However, this approach is not very effective if the opponent uses a similar learning behaviour or some other form of adaptation.

Many bots respond poorly to certain situations, such as being rushed before the bot was expecting an attack, or being attacked by invisible units before having detection. As a result, there are bots which use strategies aimed at exploiting those weaknesses in other bots.

Professional human players will often favour all-around build orders which are able to respond to a large variety of enemy build orders. Games can be lost simply by using a build order which is countered by the opponents' build order. By using an all-around build order, losses as a result of build order counters can be avoided, but this requires a high level of adaptability by the player. In comparison, many bots favour aggressive strategies which are more susceptible to being countered. In recent years, however, the top StarCraft bots have started using all-around build orders more frequently as well. This is an interesting development, as a similar change occurred with humans, which did not begin using primarily all-around build orders until strategies to counter common rush strategies had become well-practised.

Lastly, many bots obtain advantages, particularly over humans, from their high Actions Per Minute (APM). The high APM of bots allows them to control a large number of individual units simultaneously, giving them a strong micro. This makes particularly kiting-based strategies stronger for bots than for humans, which results in some units being weaker or stronger in bot games than they are in games between humans. Current StarCraft bots have yet to beat the best human players however, indicating that their micro advantages are for now unable to overcome the strategy and predictive abilities of humans.

Next, we will take a look at the playstyle and frequent strategies used by bots or humans for each of the three races in StarCraft.

2.2.1. Protoss Strategy

Protoss is a race which specialises in quality over quantity. Protoss units are expensive, but stronger than those of other races. All Protoss units have a shield that protects them. This shield is an important part of their playstyle, as the shield regenerates itself. This forces opposing races to commit to attacks against Protoss, as damage that is limited to shields is quickly regenerated after the fight. To compensate for this, the army of a Protoss player is less mobile than that of other races.

Building placement, also called 'Sim City', is a part of Protoss gameplay that is more important than that of other races. With exception of their primary base building, the Nexus, every Protoss building requires a nearby Pylon be constructed in order for the building to work. A Protoss player must avoid having a single

Pylon powering multiple buildings, as doing so would allow the opponent to disable multiple buildings by taking out a single Pylon.

The Protoss have various strategies at their disposal, such as early game aggression using Zealots and Dragoons, mid-game hit-and-run tactics using Reavers, and late-game dominance using powerful flying units. This makes the Protoss are a flexible race which can perform any style of gameplay.

2.2.2. Terran Strategy

Terran is a balanced race with no particular focus on quantity or quality. Most Terran strategies focus on defence initially, as Terrans have access to strong defensive options such as wall-ins and Bunkers. Unlike other races, many Terran buildings can lift off and fly to other locations, although the building cannot produce units or perform research during this time. This allows for Terran players to wall-in their base using buildings that can fly, allowing them to open and close the entrance to their base at will. Terran strategies typically belong to one of two categories: biological or mechanical strategies – referred to as 'bio' and 'mech'.

Bio Terran gameplay focuses primarily on two units, Marines and Medics. Being small units, Marines can be stacked closely together while firing from a distance, while Medics are able to heal any Marines that get damaged. When in large numbers, they are often referred to as a 'bioball'. As a result of the units being tightly packed, a bioball possesses a large amount of fire power in a small area, making it difficult to approach. Bio Terran uses a defensive playstyle until enough Marines and Medics are trained to create a large bioball, at which point a Terran will move out to try and win the match.

Mech Terran focuses on mechanical units such as Vultures, Siege Tanks and Wraiths. Vultures, being the fastest unit in the game, naturally excels at hit-and-run strategies. Siege Tanks can enter Siege Mode, becoming immobile but extending their attack range beyond that of any other unit, this allows an army of Siege Tanks to 'crawl forward' and destroy the enemy from a safe distance. Finally, Wraiths are flying units that are often used to support Vultures and Siege Tanks, which cannot attack flying units themselves. However, when produced in large numbers quickly, they can also be used to take the opponent by surprise. As such, mech Terran can use a variety of playstyles depending on which unit they choose to focus on.

2.2.3. Zerg Strategy

Zerg is a race which specialises in quantity over quality. Zerg units are weaker, but cheap to create. One defining feature of Zerg is Larvae. While other races must construct specific buildings in order to train units, almost all Zerg units are created using Larvae. This allows the Zerg to direct all of their resources towards their army or their economy at will. Utilising this flexibility is an important part of Zerg gameplay. Zerg units possess health regeneration just like the shield regeneration of Protoss units, but Zerg units have low maximum health values, making Zerg units easy to kill individually. This, combined with the overall slower regeneration rate compared to shields, results in health regeneration not being a major factor for Zerg playstyle.

When faced with Protoss or Terran opponents, many Zerg strategies favour early aggression. This utilises the ability of Zerg to create large armies in short periods of time. When early aggression fails, Zerg are well-suited to containment plays, in which the Zerg prevents the opponent from expanding. While 'containing' the opponent within his own base, the Zerg takes control of the map to gather large amounts of resources. Zerg is well-suited for this strategy because it is able to train more workers simultaneously than other races, allowing Zerg to take map control faster than other races. Once an economic advantage has been established, the Zerg can wear down the defences of the opponent over time to win the match.

One strategy used by Terran and Protoss players against Zerg players is to quickly produce flying units. This is because a number of popular Zerg strategies do not produce units capable of attacking flying units for some time, making them vulnerable to flying units.

When faced with a Zerg opponent, matches are almost exclusively high-aggression. This is because in a match against the same race, Zerg no longer possesses its advantage in map control. Furthermore, their primary flying unit, Mutalisks, are strong against most Zerg units. This often results in the first player to create Mutalisks winning the match. Therefore, against other Zerg players, most strategies revolve around either creating Mutalisks as quickly as possible, or winning the match before the opponent can create Mutalisks.

3
Literature

The programming of bots for StarCraft is something that has gained a lot of popularity in recent years. Since 2009, many different techniques have been created and experimented with. This chapter aims to delve into these techniques in Section 3.1, to explore their potential effectiveness and usability, particularly with regards to using them in a multi-agent approach. In addition, Section 3.2 looks at the existing usage of research of agent-based artificial intelligence in games.

3.1. Tournament results

The Brood War Application Programming Interface (BWAPI) was released in 2009, and tournaments for StarCraft bots have been a regular occurrence since 2010. This section aims to provide a recap of the tournament results achieved since 2010. This section is based on existing work by Michael Buro and David Churchill [3] for results prior to 2012. Additionally, work in [11] provides a good summary of the results achieved from 2012 to 2015, and most recently by Michal Certicky and David Churchill [10]. For each year starting from 2010, the results of the Artificial Intelligence for Interactive Digital Entertainment (AIIDE) and Computational Intelligence in Games (CIG) tournaments will be explored.

3.1.1. AIIDE

The Artificial Intelligence for Interactive Digital Entertainment (AIIDE) StarCraft tournament is the longest running StarCraft AI tournament, and generally considered as the tournament featuring the highest level of competition. This tournament runs for one to two weeks prior to the AIIDE conference, with matches being played out on multiple machines as quickly as the hardware allows it. The total number of games has increased over the years, with the 2015 tournament playing a total of 20788 games. During the conference, the results and source code of the participating bots is published. The exception to this is the first tournament in 2010, which did not require all bots submitted to the tournament to publish their source code.

In the year 2010 and 2011, each of the bots appeared to have their own speciality. The Berkeley Overmind won AIIDE 2010, with its speciality being to utilise potential fields for the path-finding of flying units, to keep them safe during attacks. The 2011 winner, Skynet, used path-finding to prevent becoming surrounded. The second place in 2011, UAlbertaBot, used heuristic search algorithms to guide its build order for an economic advantage. Other examples are bots such as Weber, Mateas and Jhala, which used particle models for state estimation.Finally, Synnaeve and Bessire implemented a plan recognition algorithm, as well as a Bayesian model for unit control. Although various techniques were employed, a recurring theme for the first place winner of 2010 and 2011 is the importance of path-finding. On the other hand, bots that chose to invest in strategy development were met with little success. A likely reason for this is that the strategy among bots was simplistic, and that the bots did not significantly respond to each others' strategy. For example, professional StarCraft player, Oriol Vinyals, found the Skynet bot to be excellent at micro and macro, but exceptionally poor at strategy. This seems to indicate that, among bot tournaments, strategy is not a major factor, as even a bot which excels at strategy (compared to other bots) is still poor at strategy compared to humans. By having a strong economy and unit management, the strategy of other bots were trumped.

Starting from 2012, AIIDE allowed bots to save data between matches. In the previous years, strategic bots had not seen significant success. However, by allowing bots to retain data on strategies employed by

other bots, the ability to learn strategic behaviour against certain enemies became more viable. In total, 6 out of the 10 entrants to the 2012 competition saved data between matches. It is evident that the bots had difficulties with learning, however, shown by the win rates of the top 3 bots against one: Skynet, UAlbertaBot and AIUR. In a total of 30 matches, Skynet beat UAlbertaBot 26 times, UAlbertaBot beat AIUR 29 times, and AIUR beat Skynet 19 times. These win-rates follow a rock-paper-scissors pattern, indicating that each of the bots utilised a specific tactic which was effective against certain other bots, and that no bot was able to adjust their strategy enough based on the saved data to break the rock-paper-scissors pattern.

In the AIIDE 2013 tournament, UAlbertaBot took first place with a win-rate of 84.49%, with the second place Skynet having only a 66.26% win-rate. The key addition to UAlbertaBot that year was SparCraft, a combat simulator responsible for simulating fights between UAlbertaBot's units and the known enemy units, producing an estimate of how the fight would end. Using this estimate, UAlbertaBot was able to intelligently pick fights even with just a single combat unit. Although a number of bots did use combat simulators prior to this, including UAlbertaBot itself, the accuracy of SparCraft was significantly greater than the combat simulators of the years prior. As a result, UAlbertaBot was able to make better decisions with its army and act aggressively from the start. Primarily, the lesson that can be learned from this is the importance of a combat simulator. Since the addition of SparCraft to UAlbertaBot, combat simulators have become a standard in StarCraft bots, with many bots incorporating the open source SparCraft in their design.

Prior to AIIDE 2014, the most successful StarCraft bots had been bots using the Protoss race, winning first place every year. In 2013 in particular, the top 4 bots were all Protoss. However, in AIIDE 2014, a number of new Terran entrants succeeded in taking over the top spots. The most successful bots employed various different strategies. In first place, ICELab utilised Bunkers for an early defensive, working up to a mid-game strategy relying on Siege Tanks. In second place, XIMP constructed a large number of static defences while it quickly advanced its technology in order to produce Carriers, the strongest Protoss unit. In third place, LetaBot used early game Bunker rushes, in which it would construct Bunkers within the opponents base. Overall, both aggressive and defensive strategies saw success.

In 2015, AIIDE featured more competitors than ever before. Additionally, this was the first year that Zerg bots saw success, taking the first, second and third place in the tournament. The winner of the tournament, Tscmoo, implemented a total of 15 different strategies and used the saving of data in between matches to select its strategy based on the opposing bot. However, in second place, ZZZKBot achieved a win-rate only 0.68% lower than Tscmoo by using only a single strategy, namely the extremely aggressive 4-pool strategy. UAlbertaBot, which had previously played exclusively Protoss, had been updated to play as Random, playing a different race each match, making it the first Random bot to compete in a major StarCraft tournament. Another notable result was the performance of AIUR, which increased its win-rate from 63% to 73% over the course of the tournament through learning, clearly showing the effect that learning behaviours can have on the game performance of a StarCraft bot.

Finally, in 2016 a new Terran bot won the tournament: IronBot. This was followed by the Zerg bots ZZZKBot and Tscmoo. Of note is that in the top 10 out of the 21 participants, there was an equal representation of all 3 races, excluding UAlbertaBot which played Random. ZZZKBot had continued to use only the 4-pool strategy. Although it only used a single strategy, the level of execution at which it performed this strategy resulted in the majority of bots being unable to deal with the strategy regardless.

3.1.2. CIG

The Computational Intelligence in Games (CIG) tournament has been held annually starting from 2011. Unlike AIIDE, the maps that will be played on in CIG are not announced in advance, meaning that developers should be prepared to play on a wide variety of maps and utilise map-independent strategies. CIG is typically held around 3 months before AIIDE, and in many years the participants have largely been similar to those of AIIDE in that same year. As such, each year of CIG will be discussed in lesser detail than AIIDE.

In CIG 2011, only four bots were submitted to CIG that were not submitted to AIIDE. Among these four bots, 'Xelnaga' placed third by using a Dark Templar rush strategy. Dark Templars are permanently invisible, requiring detection to reveal them. Many bots were unprepared for this and frequently lost to the strategy. However, Xelnaga's win rate was half that of the second place bot, UAlbertaBot, indicating that there was still a significant gap between the it and the top two bots.

Identical to AIIDE, in 2012 CIG supported the use of saving data. However, due to technical problems, data was not shared among the 6 PC's responsible for running the matches, significantly reducing the ability of the bots to learn. However, the win-rates showed did not differ greatly from AIIDE, which indicates that there had been no bot to implement a learning behaviour which significantly impacted game performance.

An additional observation is the game performance of BTHAI, a Terran bot. BTHAI was nearly unchanged from its 2011 submission, but had dropped from a win-rate of 57.5% to only 19.9%. Based on this it is safe to conclude that the majority of the participating bots had significantly improved.

The CIG 2013 competition faced technical difficulties, resulting in the data saving function being disabled. In spite of this, the results resembled that of AIIDE, with exception of UAlbertaBot having a win-rate of only 67.4%. This may imply that UAlbertaBot was the bot that utilised the data saving function the most, or alternatively that UAlbertaBot's base strategy was poor.

The maps used in CIG 2014 was increased to 20 different maps, significantly more than AIIDE's 10 maps. However, the win-rates of the bots, as well as the eventual winners of the tournament, remained similar to AIIDE, with ICEBot, XIMP and LetaBot taking first, second and third place respectively.

In 2015, the main difference with the results from AIIDE was the convincing tournament victory by ZZZK-Bot, as opposed to Tscmoo, the winner of AIIDE. The main reason for this is the difference in the number of matches played, with 20788 matches played in AIIDE and 2730 matches played in CIG. Consequently, Tc-smoo had significantly less time to benefit from being able to learn which strategy among its 15 strategies it should use against each specific opponent.

Finally, in the CIG 2016 tournament a different format was used. First, the 16 participants played a total of 1500 games each in a Round Robin format. After this, the learning data of each bot from the first phase was erased, and the top 8 bots played an additional 700 games each. Of note was that Tscmoo used the Terran race for this tournament, as opposed to Zerg which it used in AIIDE 2015 and 2016. Iron won the first phase of the tournament, but in the second phase of the tournament Tscmoo, Iron and LetaBot placed first, second and third respectively.

3.1.3. SSCAIT

The Student StarCraft AI Tournament (SSCAIT) features the highest number of participants among the three major StarCraft AI competitions, with a total of 45 participants in the 2016 tournament. Unlike CIG and AIIDE, which take place for a few weeks each year, SSCAIT runs throughout the entire year. During this time, bots continuously play games against each other on in a ladder system, where their performance is measured using Elo [16]. Unlike AIIDE and CIG, all games are live streamed to online streaming services such as Twitch and SmashCast at speeds which humans can watch. As a result, significantly fewer matches are played in the same span of time. However, viewers are able to vote on bots that they are interested in seeing, which provides the opportunity for developers to test their bot against specific enemy bots. Game replays can also be downloaded right away, allowing for the games to be analysed in greater detail.

Once per year, SSCAIT replaces the ladder with a Round Robin tournament, in which each bot plays an equal number of games against each other bot. Bots are separated into two categories: student division and mixed division. Only bots developed by students can enter the student division, while any bot is allowed to participate in the mixed division. The student bot with the highest win-rate in the Round Robin is considered the winner of the student division. After the Round Robin tournament has completed, the highest ranking bots 16 bots proceed to an elimination bracket, and the winner of this elimination bracket is the final winner of SSCAIT and the winner of the mixed division.

In the majority of years since SSCAIT launched in 2011, the results of SSCAIT have been similar to that of AIIDE and CIG, with two major exceptions. In SSCAIT 2013, Krasi0 won the mixed division, as opposed to UAbertaBot which won AIIDE in that year. Krasi0 has thus far remained closed-source, and as such has never competed in AIIDE or CIG. Krasi0 also placed second in SSCAIT 2016 and has held the highest Elo on SSCAIT for a large portion of 2016 and 2017.

In addition to this, the SSCAIT 2016 tournament was won by LetaBot, making it the only time that a bot belonging to the student category won the mixed division. In that year, LetaBot featured a number of hard-coded strategies against specific opponents. In particular, it used an SCV rush in the finals against Krasi0, which attacks using the Terran worker units right at the start of the match. Krasi0 was not prepared for this strategy, allowing LetaBot to win 3-1.

3.2. Agent-Based Artificial Intelligence in Games

Video games is a medium which frequently lends itself well to artificial intelligence. In single player games, the player is frequently matched against a variety of computer-controlled opponents. For example, in the First-Person Shooter (FPS) game 'DOOM', developed by 'id Software', the player is assaulted by various enemies which it must defeat. Each enemy has its own behavioural patterns, and acts as an individual agent. A

number of games capitalise on this even further, such as the ‘Monster Hunter’ game franchise, developed by ‘Capcom’. In Monster Hunter games, players roam the world to find and defeat large monsters. Each of these monsters possesses its own territory, and will even fight other monsters in order to defend it. These agents interact not just with the player, but also the world and other agents.

Artificial intelligence in StarCraft is different, however. Unlike DOOM and Monster Hunter, artificial intelligence in StarCraft is capable of the same actions that humans are, and are therefore expected to play similarly. In recent times, a popular genre in which this same situation holds is the Multi-player Online Battle Arena (MOBA) genre. In these games, players are part of a team, and must pick from a pool of characters to play a match with. Most commonly, teams consist of five players. The development of bots for MOBA’s is similar to the development of bots for a Real-Time Strategy (RTS) game such as StarCraft. This is because the MOBA genre originates from a custom map in WarCraft III called ‘Defense of the Ancients’. WarCraft III is another RTS game developed by Blizzard Entertainment, and as such many of the controls in modern MOBA games are similar to that of RTS games. Unlike RTS games, however, there is no base building aspect, and the player typically only controls a single character instead of an entire army. Other challenges are added to MOBA’s however, such as item building (players generally use resources earned to purchase items to enhance their character), laning (the map is typically split into 3 sections which the team must use to accrue resources) and team fights between up to ten players [37].

MOBA games such as Dota 2 and Heroes of the Storm allow players to play matches against bots, and even with bot team-mates. Many players play matches against bots in order to become comfortable with the game, as many MOBA games feature steep learning curves. Further research into this has been performed in [38], in which a bot was used to specifically tutor the player during gameplay in League of Legends, another MOBA game. The presence of a helpful team-mate that provides tips and advice for the player was shown to have a positive effect on the performance of players. In addition, MOBA games tend to attract social problems such as cyberbullying [28], in which higher skilled players will bully newcomers or lesser skilled players. This is another factor which increases the demand for playing the game with computer-controlled opponents, as many players use this in order to escape the cyberbullying and play in a stress-free environment.

A team in a MOBA game may be composed of a mix of human and computer-controlled characters, and as such MOBA bots lend itself well to agent-based design [15]. All bots should not act as a single entity, as they should be able to acting normally even when there are humans present on the team. For example, in [27] an agent-based design is used to create bots for Heroes of Newerth, another MOBA game. In this work a case-based reasoning design is used to replace the traditional rule-based design. Cooperation between the individual agents was achieved by using a ‘coalition’ protocol, where the agents would decide on a group-effort to focus on, such as attacking a particular building on the map.

For MOBA’s, the best performing bot is developed by OpenAI. In 2017, a bot developed by OpenAI managed to beat professional players in 1v1 matches in Dota 2. However, these 1-on-1 matches used exclusively 1 of the 113 heroes available at the time, and typical Dota 2 matches are played in a 5v5 format. For 2018, OpenAI has moved its focus to playing 5v5 games instead [33]. Using reinforcement learning and neural networks, OpenAI has created a team of bots capable of beating semi-professional teams. Each hero is controlled by a separate bot, and there are no communication channels between the bots. Instead, the bots use a parameter nicknamed “team spirit”, which varies from 0 to 1, in order to determine how much each bot should focus on its individual reward versus the average of the team’s reward.

RTS games have seen a great deal of previous research as well. However, the majority of this research has been limited to StarCraft. In August 2017, Blizzard Entertainment released the StarCraft II API, allowing for the creation of bots for StarCraft II, but at this time the results attained in StarCraft II are less than those in StarCraft.

Among multi-agent or cognitive bots, there are however no comparisons to GOAL. One cognitive bot that has been created for StarCraft is Soar-SC [40], which uses the Soar Cognitive Architecture developed at the University of Michigan. However, Soar-SC does not use a multi-agent design, and instead perceives and plays as a single entity. Soar-SC is also unable to defeat the built-in AI for StarCraft on its highest difficulty setting. A number of bots use a design similar to a multi-agent system, such as Iron [14], which utilises a hybrid system that combines multi-agent paradigms with global strategies and algorithms. Unlike GOAL, agents are not completely independent. OpprimoBot [17], formerly known as BTHAI, uses a complete multi-agent system, but it has not achieved major results. In the 2016/2017 SSCAIT Round Robin, it placed 37th out of the 45 contestants.

4

GOAL and Connector Basics

Over the course of this thesis a cognitive multi-agent system, ForceBot, was developed using GOAL. However, GOAL on its own is not able to interact with StarCraft using the Brood War Application Layer Interface (BWAPI). To do so it requires a connector to link it. To this end, a StarCraft-GOAL Connector, usually referred to as just 'connector', has been developed by Harm Griffioen, Danny Plenge and Vincent Koeman of the Delft University of Technology. Over the course of this thesis this connector will be utilised in order to develop a GOAL bot for StarCraft. Furthermore, one of the aims of this thesis is to evaluate the effectiveness of GOAL in the complex and agent-rich environment of StarCraft. Observations, improvements and suggestions will be made on the features of GOAL or the connector over the course of the thesis. Therefore, this chapter aims to explore GOAL and the connector in order to provide context for the work that will be performed over the course of this thesis. In Section 4.1 the GOAL programming language is explained, followed by an overview of the StarCraft-GOAL Connector in Section 4.2.

4.1. GOAL

GOAL is a rule-based programming language for programming cognitive agents that interact with an environment and with each other. Information about the environment is received through percepts, and agents can act upon the environment using actions. Agents are part of a multi-agent system and can exchange information between each other through messages. Agents maintain a cognitive state that consists of knowledge, beliefs and goals of the agent which are represented in some knowledge representation (KR) language, this cognitive state is displayed in Figure 4.1. Agents are autonomous decision-making agents that derive their choice of action from their beliefs and goals.

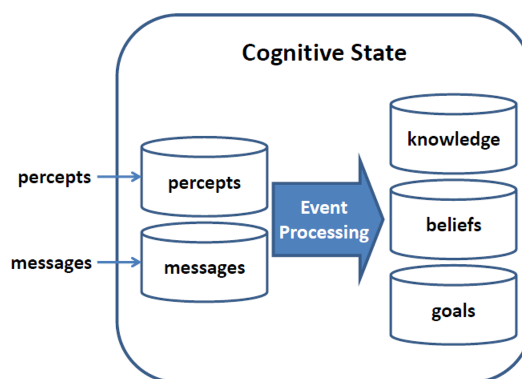


Figure 4.1: Cognitive State

Agents operate via cycles. At the start of each cycle, agents receive percepts from the environment, as well as any messages directed towards them by agents. The agent will then process its 'event module', in which the programmer should handle the agents' percept and message handling. Afterwards, the agent will process its 'main module', in which actions (if any) should be taken to advance towards some goal.

GOAL has been used in various environments prior to StarCraft. For example, the team-based Blocks World environment, called Blocks World 4 Teams (BW4T) [22]. In this environment, coloured blocks are spread out across numerous rooms, and a team of agents must hand in a specific sequence of coloured blocks at a designated drop-off location. In order to complete the task as fast as possible, the agents must communicate with one another. By sharing information, they can quickly discover the contents of each room. By sharing intentions, they can avoid getting in each others way.

Other environments which GOAL supports include environments applicable to real-life, such as an elevator simulation, in which agent-controlled elevators must transport passengers across numerous floors as quickly as possible. Additional GOAL environments for real-life games such as the Hanoi Tower Game and Tic-Tac-Toe also exist.

Of particular interest is the Unreal Tournament 2004 environment. Unreal Tournament 2004, commonly referred to as simply UT2004, is a First-Person Shooter (FPS) game developed by Epic Games. Thus far, it is the only other video game environment supported by GOAL. However, this video game environment failed to show the capabilities of GOAL. This is because the implementation does not support more than 10 agents [19] and offers a very restricted set of actions that agents can perform [26], limited mostly to actions like "go to". This is the result of middleware software handling actions such as shooting and path planning, which limits the degree of complexity of the game.

In comparison, StarCraft does not suffer from these problems. The challenges presented by aspects of the game such as micro and macro, building planning and resource management are not lost in abstraction. StarCraft also presents itself as an environment with a potential for more than a hundred agents. Consequently, StarCraft challenges GOAL both in in computational performance and complexity.

4.1.1. GOAL Project Structure

There are four different files types used in GOAL projects:

- `mas2g` – The project file, which contains agent definitions, the environment to use, etc.
- `pl` – Knowledge files, which are Prolog files containing knowledge speicifications for an agent.
- `act2g` – Action specification files, used to define the actions that an agent is able to perform.
- `mod2g` – Module files, in which the decision making of your bot will primarily occur in.

In this section all four file types will be discussed in detail in order to provide a foundation for understanding how GOAL operates.

Multi-Agent System file (.mas2g)

A multi-agent system file is the main project file. It includes information such as which environment to use, arguments for the environment to use, as well as the agent definitions and agent launch policies.

The environment connector is used by GOAL to retrieve and act upon the environment which the agent is designed to operate within. In the case of StarCraft, this is the StarCraft-GOAL Connector [24], which is explored in detail in Section 4.2. The arguments supplied to the environment are different for each environment. In the case of the StarCraft connector, they consist of options to setup the automatic startup of StarCraft matches, as well as enabling certain in-game cheat commands for testing purposes. These cheat commands are disabled in standard matches.

```
use "starcraft-connector.jar" as environment with ...

define drone as agent {
   use DroneInit as init module.
   use DroneEvent as event module.
   use Drone as main module.
}

launchpolicy {
   when type = zergDrone launch drone.
}
```

The main files that are to be used by agents are also specified within the `.mas2g` file. Each agent may have an init, event and main module. All modules, including the main module, are optional.

Init Module
: Will run upon creation of the agent, before any other modules. The init module should be used for sections of the code which only need to run once, such as processing percepts which are only received once.

Event Module
: Will run before the main module. The event module should handle the processing of information such as percepts and messages.

Main Module
: Will run last. Decisions should be placed within this module.

Finally, the *launchpolicy* dictates which agent type is launched. Whenever a new entity is added to the environment, the connector will attempt to bind an agent to that entity. It uses the launch policies to determine which agent type to bind. If no policy is declared for the entity type, no new agent is launched.

Knowledge file (.pl)

Knowledge files, indicated by the `.pl` extension, are Prolog files containing knowledge specifications for an agent. Prolog is a logic programming language [41] often used for artificial intelligence [39]. It is primarily a declarative language, in which the program logic is expressed in terms of facts and rules. A typical knowledge file will start by declaring the dynamic knowledge by means of the `:- dynamic` directive. Any number of dynamic predicates may be specified, and all dynamic predicates used by the agent must be declared in one of its knowledge files for the program to compile.

```
:- dynamic
   self/2,
   status/6,
   enemy/7
.
```

Each predicate is assigned an arity, indicated by the `/N` component at the end, where *N* is a natural number. The arity denotes the number of arguments which the predicate contains. For example, `self/2` contains the <Id> and <Type> of a unit, making it a 2-arity predicate. The variable type of each argument does not matter, only that the arity matches that of the predicate declaration. It is also possible to define rules or facts within the knowledge file. The example below can be called using `distance(0, 0, 10, 0, Distance)` which will return `true` with `Distance = 10`.

```
% Calculate distance between two points
distance(X1, Y1, X2, Y2, Distance) :- Distance is sqrt( (X2 - X1)**2 + (Y2 -
Y1)**2 ).
```

Prolog is well-suited for recursive functions in particular. It does this via the use of Prolog's rule ordering and backtracking. Prolog will always attempt to apply the rules in the same order that they are defined in within the knowledge file. If a rule fails (i.e., does not return true), it will attempt to backtrack and find another rule that it can fulfil. For example, the code below will traverse a list and return true if it finds an instance of 'Zergling' before it finds an instance of 'Hydralisk'.

```
% True if we find 'Zergling'
findZergling(['Zergling' | _]).
% False if we find 'Hydralisk'
findZergling(['Hydralisk' | _]) :- !, false.
% Continue the recursive call
findZergling([_ | Tail]) :- findZergling(Tail).
```

The first element of a list is retrieved using the `[Head | Tail]` notation. If one of these parts is not of importance, the value can be ignored by replacing it with an underscore (`_`). In the first rule, if 'Zergling' is encountered, then the rule will return true as there are no other conditions to fulfil. If the first rule can not be fulfilled, i.e. the head element is not 'Zergling', then it will backtrack out of this rule and attempt to find another rule to unify with, leading it to the second rule. The second rule applies if 'Hydralisk' is the

head element. In this case, the rule will return false. However, it will not backtrack out of this rule, as the exclamation mark (!) is called a 'cut' and prevents backtracking. If neither of the first two rules can be fulfilled, it will move on to the third rule, which recursively calls the function on the tail of the list. This will continue until one of the first two rules can be fulfilled, or until the list is empty. If the list is empty, the third rule can no longer be applied. As there are no other rules which it can attempt to apply, the call will fail and return false.

Action Specification file (.act2g)

The action specification file is used to define the actions that an agent is able to perform, and the pre-conditions and post-condition updates that will be used. For example, the below code sample can be used to begin gathering minerals within StarCraft.

```
define gather(Id) with
   pre { mineralField(Id), idle }
   post { not(idle), gathering }
```

For the action to be allowed to be performed, the agent must believe that a `mineralField/5` with the given `Id` exists, and that the agent believes itself to be currently idle. If these pre-conditions are met, the environment will attempt to execute the action, and the post-condition updates will be applied. In this case, the `idle` belief is deleted as a result of `not(idle)` being inserted into the belief base. Furthermore, the `gathering` belief is inserted.

Module file (.mod2g)

Module files are where the decision making of your bot will primarily occur in. A module file starts by importing any required GOAL files, as well as specifying the ordering and exit logic of the module if required.

```
use SomeActionFile as actionspec.
use SomeKnowledgeFile as knowledge.
use AnotherModule as module.
order = linear.
exit = never.

module Module { ...  }
```

The 'order' of the module dictates the order in which the rules within the module is traversed. A *linear* order will traverse the module from start to finish, and terminate once a rule has been found that it is able to make true. A *linearall* order will traverse the module in the same order, except it will not terminate after one successful rule. Aside from *linear* and *linearall*, *random*, *randomall*, *linearrandom* and *linearallrandom* also exist. By default, modules specified as 'main module' in the `.mas2g` file will use *linear* ordering (as you can not perform more than one action), while 'event module' files will use *linearall* ordering.

The 'exit' of the module dictates when the module will be exited from. If the 'main module' is exited from, the agent will terminate. The available exit conditions are *always*, *noaction*, *nogoals* and *never*. By default all modules use *always* as the exit condition, with exception of the main module, which uses *never*.

The module itself consists of rules which use percepts, beliefs and goals. In the below example of a module, the module first inserts any `mineralField/5` percepts which it receives as a belief in its knowledge base. After that, it will gather minerals if it has the goal `gatherMinals`.

```
module GatherModule {
   forall percept(mineralField(Id, _, _, _, _)) do insert(mineralField(Id)).
   if goal(gatherMinerals), bel(mineralField(Id)) then gather(Id).
}
```

4.1.2. Messaging

Agents in GOAL can communicate with one another via sending messages. By doing so, agents are able to coordinate and organise themselves. Communication is done via the `send` action, which receivers can then process using the `sent` operator. An example of this shown below.

```
if true then allother.send(helloWorld).
if (SomeAgent).sent(helloWorld) then (SomeAgent).send(hello).
```

When sending messages you can specify the receiver. There are the basic targets of `all` and `allother`, the latter of which will send it to all agents except itself. Additionally, an agents name can be given as well, for example: `(manager).send(...)`. Finally, communication channels can be used to spread messages to a group of agents. These communication channels follow a subscription model, where individual agents may subscribe or unsubscribe to a communication channel at will. This is done using the `subscribe` and `unsubscribe` actions respectively.

Finally, messages can be given a context by supplying a 'mood'. There are three different moods that can be given, *indicative* (`:`), *declarative* (`?`) and *interrogative* (`!`). In order to apply a mood to a message, the respective symbol must be appended to the `send` action, as well as the receiving `sent` operator. In this manner, it is possible to distinguish the intent of messages using the mood, which also allows for the distinguishing of identical messages based on the mood with which it was sent. If no mood is supplied, the mood will default to an indicative mood.

```
if true then subscribe(drones).
if true then (drones).send?(hello).
if (SomeAgent).sent?(helloWorld) then (SomeAgent).send:(hello).
```

When messages are received, they are inserted into the mailbox of the receiver. The message will remain within the mailbox for a single cycle, and is removed automatically afterwards. If multiple identical messages are received at the same time by the same sender, any duplicate messages will automatically be filtered out.

4.1.3. IDE & Tools

This section introduces the IDE and tools offered by GOAL. Over the course of this thesis these will be evaluated. The GOAL IDE comes in the form of a plug-in for Eclipse. This plug-in provides the basics of modern IDE's such as creating, renaming, moving and deleting files or projects, as well as syntax highlighting, automatic code completion and automatic building.

Tools for debugging are provided by the plug-in. Standard for this are error and warning messages generated by the automatic building. Projects may also be ran in 'debug mode' which can be used to inspect the state of individual agents. This is done by allowing the user to observe the beliefs, goals, percepts and messages received by agents during runtime, interact with the agent through an interactive console, and use a source-level (stepping) debugger to inspect an agents behaviour in detail. The interactive console can be used to, for example, delete a belief during runtime in order to observe the reaction of the agent. Breakpoints can be set in the code of an agent, in order to observe an agents state at a specific code execution point.

Additionally, logging can be enabled within the plug-in preferences, with options for the type of information that is logged. This log is then written to file in an XML format. Finally, the IDE provides a profiling tool, which will record the execution time on activities such as executing individual modules, rules, actions, etc. Using the information generated by the profiling tool, it is possible to identify bottlenecks within the code.

4.2. StarCraft-GOAL Connector

In order for GOAL to interact with the StarCraft game, a connector which links the data from BWAPI with the Environment Interface Standard (EIS) utilised by GOAL is required. EIS is a Java-based interface standard for connecting agents within GOAL to entities in an environment. EIS allows for GOAL to be used in any environment that is compliant with EIS. This creates a structure for connecting GOAL to StarCraft that is represented in Figure 4.2.

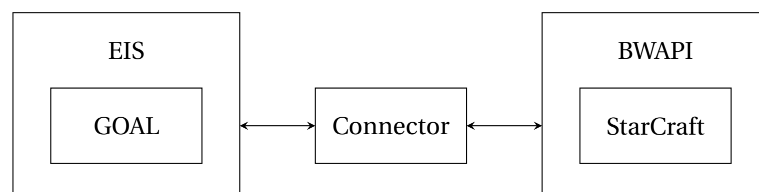


Figure 4.2: GOAL-Connector-StarCraft structure

In order for GOAL to interact with StarCraft, Harm Griffioen, Danny Plenge and Vincent Koeman of the Delft University of Technology have developed the StarCraft-GOAL Connector. This connector is responsible for the communication between GOAL and BWAPI. This is done by mapping the units and buildings in StarCraft to entities in the connector. These entities are EIS-compliant, thus allowing GOAL to interact with the StarCraft environment via the StarCraft-GOAL Connector. This relationship between units, agents and entities is shown in Figure 4.3.

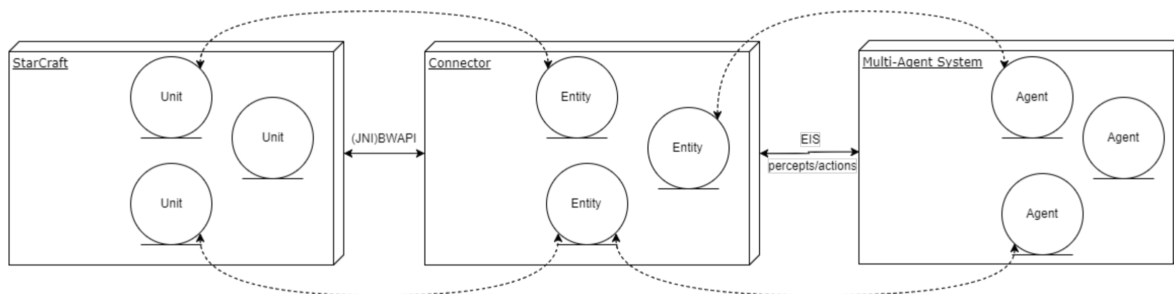


Figure 4.3: Diagram of the relationships between units, entities and agents

The connector was developed a year before the start of ForceBot's development, and before then had only been used in small testing setups. Therefore, the connector was still immature when development on ForceBot began. As a result, over the course of ForceBot's development the connector saw many changes, ranging from adding/removing information to percepts, adding new functionality, or simply fixing bugs. In this section the initial state of the connector will be outlined. The later development chapters will then outline any notable changes that were made to the connector from this state onward.

The role of the StarCraft connector is to abstract the environment of StarCraft into one which GOAL can effectively interact with. This means that units in the game must be automatically assigned agents to control them. These agents must then receive percepts which tell them what is happening within the game, as well as be able to perform actions with which they can influence the unit to which they are connected (and thus, the game). The design of this environment was guided by two conflicting objectives [26]:

1. The environment should facilitate multi-agent systems that operate at a level of abstraction that is as high as possible.
2. The environment should facilitate multi-agent system implementations with as many different strategies as possible.

This design ideology does not intend to be as precise as C++ or Java agents, however should be sufficient to play the game effectively. Primarily, the abstraction aims to automatically handle the low-level operations, namely the life-cycle of each agent:

- Assign each unit an agent based on the `mas2g` file definitions.
- Retrieve information from BWAPI and convert them to percepts.
- Convert actions performed by agents into command calls in BWAPI.
- Upon the death of a unit, terminate the agent connected to that unit.

These low-level operations are not important to the design of individual bots and are therefore performed automatically by the connector. High-level decision making such as where to place buildings, when to expand, or what to attack, is still in the hands of the programmer.

The environment is designed to support the large number of units that can be present during StarCraft games. It is not uncommon for this number to exceed 100 units. This, too, is a task which is made more manageable by abstraction. By simplifying the information which the agents receive it becomes possible to increase computational performance. Care must be taken, however, to ensure that vital information is not lost in this process. The exact importance of information is often difficult to define, however a programmer should not feel limited in the strategies, responses or actions that it can perform as a result of information that was lost due to abstraction.

Finally, GOAL provides the ability to launch independent agents that are not connected to the environment. These agents follow a different life-cycle to that of normal agents: since they are not connected to a unit within StarCraft like normal agents, they will not receive percepts. However, these agents can still communicate with other agents. Furthermore, while normal agents are terminated when the unit which they are connected to no longer exists, agents that are not connected to any unit will never be terminated. This makes these agents well-suited for performing the role of a manager, as they can act as a central point for communication that is guaranteed to exist.

4.2.1. Connector Outline

In order to clearly explain the changes made to the connector over the course of this thesis, a brief overview of all percept types present in the connector during the start of the thesis is provided in this section. For each percept type, its purpose is explained and its parameters are listed. All percept types belong to one of four categories: global static percepts, global dynamic percepts, generic unit percepts and unit-specific percepts. Global percepts are the same for all units on a single frame, while generic unit percepts and unit-specific percepts are unique to each agent.

4.2.2. Global Static Percepts

Global static percept types are received by all agents. Percepts of this type are perceived only once, generally during the agents' first cycle. Global static percepts typically contain information regarding the map or other information that will not change over the course of the match.

base(<X>, <Y>, <IsStart>, <RegionID>)
: This percept type is used to inform agents of all base locations on the map. The <IsStart> parameter is used to indicate whether the base location is a starting location.

chokepoint(<X>, <Y>)
: This percept type is used to indicate where on the map a chokepoint is located.

enemyRace(<Race>)
: This percept type informs the agent what the race is of the enemy player. The <Race> parameter may be 'unknown' if the player started as a random race and has not been seen yet.

map(<Width>, <Height>)
: This percept type provides the height and width of the map on which the match is being played on.

winner
: This percept type is received at the end of the match, indicating that the player has won the match.

4.2.3. Global Dynamic Percepts

Global dynamic percepts are perceived by all agents. These percepts provide information about other units or the game state, meaning that they are identical for every agent in a certain frame.

attacking(<AttackerID>, <TargetID>)
: This percept type is used to indicate which enemy units are attacking which friendly units.

enemy(<Type>, <ID>, <Health>, <Shield>, <Conditions>, <X>, <Y>)
: This percept type provides information such as unit type, health and location of all visible enemy units on the map.

friendly(<Type>, <ID>, <Conditions>)
: This percept type provides information regarding all friendly units. This percept does not contain as much information as the `enemy` percept – the <Health>, <Shield>, <X> and <Y> parameters are not included. Units instead receive the omitted information using the `status` percept. It is assumed that the omitted information is not necessarily of value to all other agents, making it more efficient to share this information on-demand through messaging between agents.

gamespeed(<Speed>)
: This percept type indicates the speed at which the game is currently running.

resources(<Minerals>, <Gas>, <UsedSupply>, <TotalSupply>)
This percept type indicates the current number of minerals and vespene gas, as well as both the current and maximum supply.

4.2.4. Generic Unit Percepts

Generic unit percepts are received by all agents. However, the information received through these percepts is specific to the agent that received it, meaning that generic unit percepts are not identical for every agent in a certain frame.

self(<ID>, <Type>, <MaxHealth>, <MaxShields>, <MaxEnergy>)
This percept type tells the agent what its unit type and ID is, as well as its maximum stats. Perceived only when the agent is created.

status(<Health>, <ShielD>, <Energy>, <ConD>, <X>, <Y>)
This percept type is used to inform the agent of its current stats, condition and location.

4.2.5. Unit-Specific Percepts

Unit-specific percepts are similar to generic unit percepts, in that they are received on a per-agent basis. However, instead of being perceived by every agent, they are only perceived by agents connected to certain unit types or under certain circumstances.

constructionSite(<X>, <Y>, <InPylonRange>)
Perceived only by workers. This percept type indicates all explored and non-obstructed locations on which new buildings may be placed. The <InPylonRange> parameter is only used by Protoss, as their buildings must be powered by nearby Pylons before placement of most of their buildings is allowed.

defensiveMatrix(<Health>)
Perceived only by a unit which has a Defensive Matrix placed on it. This percept type tells an agent how much health the Defensive Matrix placed on the unit has left.

mineralField(<ID>, <Resources>, <ResourceGroup>, <X>, <Y>)
Perceived only by workers. This percept type gives the agent information about all mineral fields currently visible on the map.

queueSize(<Size>)
Perceived only by buildings which are able to train units. This percept type indicates how many units are in queue for training.

rallyPoint(<X>, <Y>)
Perceived only by a building that has placed a rally point. This percept type indicates where the rally point belonging to this unit is located. Newly trained units will automatically move to the rally point of the building that trained them.

rallyUnit(<UnitID>)
Perceived only by a building that has placed a rally point. Similar to the `rallyPoint` percept type, this percept type instead indicates to which unit the rally point belonging to this unit is bound. Newly trained units will automatically move towards that units location.

researching(<TechType>)
Perceived only by a building that is researching a tech type. This percept type indicates which tech type is currently being researched by this unit.

spaceProvided(<CurrentSpace, MaximumSpace>)
Perceived only by units which may hold other units. This percept type gives the amount of space currently being occupied, as well as the maximum amount of space that this unit can hold.

unitLoaded(<UnitID>)
Perceived only by units which may hold other units. This percept type provides the ID's of any units being held by the unit in question.

upgrading(<Upgrade>)
Perceived only by a building which is perform an upgrades. Similar to the `upgrading` percept, this percept type indicates the upgrade which is currently being performed by this unit.

vespeneGeyser(<GeyserID>, <Resources>, <X>, <Y>, <RegionID>)
Perceived only by workers. This percept type gives the agent information about all vespene geysers currently visible on the map.

workerActivity(<Activity>)
Perceived only by workers. This percept type displays the current activity of the agent. These activities are one of 'gatheringGas', 'gatheringMinerals', 'constructing' and 'idling'.

In addition to percept types, the StarCraft-GOAL Connector also features various actions. However, actions did not undergo significant changes over the course of ForceBot's development, and as such have been left out of the connector outline. A complete list of actions can be found in the StarCraft Environment Manual for the connector [24]. The reason for the relatively few changes to actions is that there are a limited number of commands that can be performed in the game, making it simple to include an action for each command. Therefore, no new actions needed to be made, as every command already had a corresponding action. Instead, most changes to actions were in order to resolve bugs or improve debugging.

5

Milestone 1: MAS Project

The first milestone covers the duration of the Multi-Agent Systems (MAS) Project 2017, spanning from April until the end of June. The MAS Project is a course hosted by the University of Delft in which teams of bachelor students compete to develop bots using GOAL. This was the first year in which student teams developed StarCraft bots, in previous years the teams developed Unreal Tournament bots. As development on ForceBot started at almost the same time, the MAS Project was a good means of comparing ForceBot's game performance to other, similar bots.

This chapter will first discuss the changes that occurred to the StarCraft-GOAL connector over the course of this milestone in Section 5.1. The MAS Project was the first time that the connector was used by users other than the developers. Therefore, many of the changes are focused on usability and missing information, as the developers have not yet had an opportunity to see ordinary users interact with the system. The development of ForceBot has been split into 3 versions, covered in Section 5.2, 5.3 and 5.4. For each section, the focus will be on the improvements made to ForceBot in order to best the other GOAL bots while moving to the next version. The game performance of ForceBot was measured using 3 tournaments that took place over the course of the MAS Project, which are analysed in Section 5.3.4 and 5.4.1. Finally, in Section 5.5 the lessons learned and progress made on ForceBot over the course of this milestone is summarised.

5.1. StarCraft-GOAL Connector Development

As a result of the MAS Project being active, changes to the connector during the course of this milestone time were made with backwards compatibility in mind. In particular, this holds for all 2.X versions of the connector, which were the versions that will be used by the student teams. For example, when it was decided to add information regarding which region a unit is located in, a new `unitRegion/2` percept type was added, rather than adding that information to existing percept types. This is because adding or removing a parameter in a GOAL percept type will require all code using that percept type to be adjusted. By adding the information as a new percept type instead, backwards compatibility is maintained, making it easier to update to newer versions of the connector for student teams.

Once the MAS Project concluded, there was no longer a need for backwards compatability. At that point the existing percept types were changed instead, producing version 3.0 of the connector, which is no longer backwards compatible with 2.X versions. The changes to percept types which will be discussed in this chapter are the non-backwards compatible changes.

Furthermore, in version 3.0 of the connector, the parameters of many percept types were adjusted to be more logical and consistent across the different percepts types. The primary purpose of these changes is to streamline and simplify the connector for use by users. A number of examples of these changes are:

- The `self` percept type began with <ID>, followed by <Type>, where as the `enemy` percept type used the reverse order.
- The connector distinguished between <ResourceGroup> and <RegionId> for a number of percept types, namely `base`, `region`, `mineralField` and `vespeneGeyser`. As all maps used only have a single <ResourceGroup> for every <RegionId>, the distinction serves no purpose.

- For some percept types a <RegionId> is given along with an `(<X>,<Y>)` location, while others do not.

These sort of changes that do not affect the actual information being made available, but only the manner in which they are presented, will not be discussed in detail in this chapter.

5.1.1. Global Static Percepts

chokepoint(<X1>, <Y1>, <X2>, <Y2>, <RegionID1>, <RegionID2>)
Initially only possessing the centre `(<X>, <Y>)` positions, chokepoints have been altered in order to indicate both endpoints of the chokepoint. These points can be used to determine properties such as the size, centre and orientation of the chokepoint. Additionally, it now gives the regions which it connects via the <RegionID1> and <RegionID2> parameters.

region(<RegionId>, <CenterX>, <CenterY>, <Height>, <ConnectedRegionsList>)
This percept type has been added to provide agents with more information about the layout of the map. The height of a region is given as well, as it was impossible to exploit terrain height advantages with the amount of information available previously.

5.1.2. Global Dynamic Percepts

enemy(<UnitID>, <Type>, <Health>, <Shields>, <Energy>, <Conditions>, <X>, <Y>, <RegionID>)
The `enemy` percept type was missing the <Energy> parameter, one of the three resources used by units. By including information about the enemy energy, judgements can be made on how threatening support units which require energy are.
Furthermore, the <RegionId> parameter has been added as it is important be aware of which region enemies are located in. This is particularly important due to terrain features. Units which have `(<X>, <Y>)` locations that are close to each other, may still be far away in terms of path distance, as non-flying units can only move between regions using chokepoints.

gameframe(<Frame>)
The `gameframe` percept type has been added in order to replace the `gamespeed` percept type. Every 50 frames an agent receives this percept, indicating that 50 frames have passed.
The intent was to use `gamespeed` in combination with the real-time runtime to determine the in-game time. However, this approach does not work when the gamespeed is set to 0, which means the gamespeed becomes as fast as the computer is able to manage, making the actual speed unknown. The `gameframe` percept type will provide the current frame according to BWAPI, making it always accurate to the actual elapsed game time.

nuke(<X>, <Y>)
This percept type was added shortly after the start of the MAS Project, as there was no way to spot nukes before the inclusion of this percept type.

winner(<IsWinner>)
As this percept was only received when the game was won, it was not possible to determine when you had lost the game. This is problematic when using automated testing or learning behaviour, as you cannot detect losses. In order to correct this, the percept has been changed to always be sent when the game ends and instead use a boolean <IsWinner> parameter to indicate the winner.

5.1.3. Generic Unit Percepts

self(<UnitID>, <Type>)
The parameters <MaxHealth>, <MaxShields> and <MaxEnergy> were removed. These are in fact static values that can be determined using the <Type> parameter. Rather than including this information in the connector, a user is able to include this information directly into the agent's knowledge base when desired.

status(<Health>, <Shield>, <Energy>, <Conditions>, <X>, <Y>, <RegionId>)
The <RegionId> parameter has been added as it is important for agents to be aware of which region it is located in. This is particularly important due to terrain features. Units which have `(<X>, <Y>)` locations that are close to each other, may still be far away in terms of path distance, as non-flying units can only move between regions using chokepoints.

5.1.4. Unit-Specific Percepts

constructionSite(<X>, <Y>, <RegionId>, <InPylonRange/OnCreep>)

This percept type has been altered to include a <RegionId> parameter in which the given construction site is located. Without this information an agent can not know for certain whether a construction site was located inside or outside the base.
Furthermore, the <OnCreep> variable was added for Zerg bots, which require creep to construct most buildings on. This requirement is similar to Protoss requiring a nearby Pylon, and this information not being available from the start was an oversight.

mineralField(<MineralId>, <Resources>, <X>, <Y>, <RegionID>)

The `mineralField` percept type was changed to include a <RegionID> parameter rather than a <ResourceGroup> parameter. This is because all maps used only have a single <ResourceGroup> for every <RegionId>.

queueSize(<Size>)

This percept type has been expanded so that the <Size> parameter will give the number of Larvae for Hatcheries, Lairs and Hives. Knowing how many Larvae these buildings possessed was impossible before this change.

rallyPoint(<X>, <Y>)

This percept type has been removed. Generally speaking, any rally point order which a new unit may have is almost instantly cleared once the agent for that unit becomes active. Furthermore, if need be, the rally point can be tracked by the agents themselves, rather than having StarCraft 'enforce' this.

rallyUnit(<UnitId>)

This percept has been removed for the same reason as the `rallyPoint` percept type.

spaceProvided(<CurrentSpace, MaximumSpace>)

This percept type has been removed. The maximum space is known in advance depending on the unit type, and the current space can be kept track of through the `unitLoaded` percept.

unitLoaded(<UnitId>)

The <Type> parameter of this percept type has been removed. The <Type> of the loaded unit can be determined using the `friendly` percept type, making the inclusion of this parameter redundant.

upgrading(<Upgrade>)

This percept type has been removed. The distinction between researches and upgrades is unclear to players and bot developers, as it is only clearly present within the internal code of StarCraft itself. To make things simpler for users, these two percept types were unified into a single `researching` percept type.

vespeneGeyser(<GeyserId>, <Resources>, <X>, <Y>, <RegionID>)

The `vespeneGeyser` percept type was changed to include a <RegionID> parameter rather than a <ResourceGroup> parameter. This is because all maps used only have a single <ResourceGroup> for every <RegionId>.

workerActivity(<Activity>)

This percept has been removed. It was used to display activities such as 'gatheringMinerals' or 'idling'. However, these activities were added as flags to the <Conditions> parameter in the `status` percept type. As such, this percept type is no longer needed in order to determine the activity of a worker.

5.1.5. Action Changes

Overall, connector actions saw few changes over the course of the MAS Project. Almost all required functionality was already present in the existing actions. However, two changes did occur as a result of the MAS Project, these two are outlined below.

cancel

The `cancel` action can now be called with an optional <ID> argument. The `cancel` action is used

to interrupt an in-progress morph or construction. However, units which are morphing or under construction have no agent attached to them. Therefore, they can not be cancelled. This issue has been resolved by letting the `cancel` action be executed by other agents instead. By having other agents use the `cancel` action with an additional <ID> argument, they are able to perform the cancel action for other units.

setRallyPoint

This action has been removed. As discussed for the `rallyPoint` percept type, rally points have been found to have no real use for GOAL agents, as on start-up they will immediately override the rally point move order with their own actions.

5.1.6. Connector Change Overview

The changes that have been made to the connector over the course of the MAS Project primarily belong to two categories. The first category is changes to address information that was missing. For example, the <OnCreep> parameter in the `constructionSite` percept type or the missing `nuke` percept type.

Secondly, the level of abstraction has been adjusted to be simpler for the programmer in a number of areas. For example, the `constructionSite` percept type did not have a <RegionId> parameter previously, as it was assumed that the programmer could determine this information through other available information. However, in practice this was found to be more difficult than anticipated. Instead, this information will now be provided directly.

Lastly, redundant information was condensed or removed. For example, the `workerActivity` percept type provided the same information as the <Conditions> parameter included in the `status` percept type. All in all, the changes are aimed towards usability, as this is the first time that the connector has been used by people besides the developers of the connector.

5.2. Design and Development of ForceBot: V1 - Prototype

The first version of ForceBot was a prototype version, developed in order to determine approaches to a number of basic steps such as worker management, building construction and attacking. Being a prototype version, ForceBot V1's code was not extended upon in order to create later versions of ForceBot.

ForceBot V1 uses a strategy, known as '4-pool', which is the most aggressive strategy in StarCraft. This strategy was chosen as it is a simple yet powerful strategy, which aims to end the game in a single attack. This reduces the complexity of the game, allowing for the bot to be fairly simple in design: the starting building of the Zerg, the Hatchery, would decide when and which buildings to construct or unit to train, effectively serving as a manager. The reason that a dedicated manager agent was not used for this task is that these do not receive percepts from StarCraft. It is possible to have other agents send the manager agent the information, however for the scope of this version, using the Hatchery as a manager instead was a simpler approach. Each agent determines on its own how to achieve this task. Drones, the worker unit for Zerg, decide independently where to harvest minerals and construct buildings. Collection of vespene gas is not needed for a 4-pool strategy, so this aspect was left out. Combat units, in this case limited to Zerglings, attacked as soon as they were created, but decided independently what to attack.

ForceBot V1 tried to use something called the 'Extractor trick'. This trick involves starting construction on a building (specifically, an Extractor) in order to reduce the current supply. By freeing up supply, you can then train additional units. Once these units begin training, you may cancel the building being constructed, which will push the player over the supply limit, thus allowing for more units to be trained than normally possible. However, it was not possible to successfully perform this trick at the time due to limitations involving the connector. Since agents are attached to units or buildings upon their completion, it is not possible to perform the `cancel` action during construction. This functionality has since been added via the `cancel(<ID>)` action, detailed in Section 5.1.5.

Although the strategy that this version used is simple, it was capable of beating the default bots that were included in StarCraft: Brood War, which were not designed to be able to handle such early attacks. However, there are numerous drawbacks to the 4-pool strategy. Precisely because this strategy aims to end games with a single attack, it will often lose otherwise as a result of falling behind in economy. Furthermore, the simple strategy fails to show off the capabilities of the connector or the language. Because of that, this strategy will not be relied upon in future versions of ForceBot.

5.3. Design and Development of ForceBot: V2 - Beginning

The second version of ForceBot was developed in order to compete in the MAS Project tournaments and play against the bots developed by the Bachelor student teams. As strategies that aim to end the game early can result in one-dimensional bots, one design goal of ForceBot V2 was to utilise strategies that focus on the later stages of the game. For this reason, focus was placed on ensuring that an economic advantage is gained on the opponent.

5.3.1. Designing a Manager

One problem that had already shown itself in ForceBot V1 is the difficulty of managing resources: many agents in the game can spend resources, but all of them share the same resource pool. If each individual agent were to try and spend resources independently, they will only get in each others' way. Organisation is required in order to avoid problems like these. In ForceBot V1, the starting building was assigned to this task. However, using the starting building is not feasible for a bot with a scope greater than that of ForceBot V1. This is because the manager would be lost when the starting building is destroyed. Therefore, the task of manager was moved to a dedicated manager agent. As this manager agent is not connected to any unit, it will never be terminated. This manager agent is responsible for managing resources, determining which units and building to construct and ensuring that the build order is correctly executed. The approach of using a manager agent did create some problems of its own, however, as manager agents do not receive percepts like agents do.

Within GOAL, managers are agents operating outside of the environment and are not connected to any unit. Since they operate outside of the environment, these manager agents do not receive percepts from the environment. In order for managers to be aware of the state of the game, other agents must send messages containing this information to the manager agents. Initially this task was assigned to the starting building, the Hatchery. The first Hatchery is present at the start of each game and is your most important building, making it suitable for the task. This solution comes with the same problems, albeit less severe, as using the Hatchery itself as a manager, however. The Hatchery may be destroyed, which will cause its agent to terminate. The Hatchery may also be morphed into a Lair or a Hive. When a unit is morphed, the existing agent is replaced with a new agent corresponding to the new unit type. If for any reason the first Hatchery's agent ceases to function, the manager would no longer receive information.

An obvious solution to this problem is to assign a different agent to the task of forwarding information when the agent currently tasked with it dies. However, gaps in data will appear as a result of the time it takes to assign a new agent this task. For this version of ForceBot, this problem was circumvented by ensuring that the main Hatchery does not morph into a Lair or Hive. Furthermore, if the first Hatchery is destroyed, the match is often already lost, making this situation relatively unimportant. Regardless, this approach is less than ideal, as it reduces the flexibility of the available build orders by enforcing the construction of a second Hatchery before a Lair can be constructed, and although difficult, there are matches which can be won even after losing the first Hatchery. For future versions of ForceBot, a better solution is thus required.

5.3.2. Spending Resources

When the manager decides that it wants to spend resources on, for example, training a new unit, it first needs to send a message to an agent capable of training that unit, and that agent must then perform the `train` action. This means that resources are not spent instaneously. Therefore, it is necessary for the manager to account for resources that it is planning to spend. In order to address this problem a belief was added that is used to store the amount of resources that are slated to be spent. Whenever the manager wants to spend resources, it 'reserves' the required resources. Resources are 'freed' when it receives a message for one of the following events:

- A new building begins construction
- A Hatchery, Lair or Hive trains a new unit
- A building starts a research

This version of ForceBot is, however, prone to reserving resources for training new units while it has no Larvae available. New units are trained by Larvae, which are periodically spawned at Hatcheries, Lairs and Hives. New units cannot be trained while there are no Larvae available. Because ForceBot V2 is not aware of the amount of Larvae available at each Hatchery, it frequently plans for new units to be trained while no Larvae are available. This can cause resources to go unused for long periods of time. A goal for the next version

is to integrate the `queueSize` percept type into the decision making of the manager. The `queueSize` percept type, which Hatcheries receive, indicates the number of Larvae which they possess. By taking the number of Larvae available into account, ForceBot will instead be able to use its excess resources on other tasks.

5.3.3. Unit Deaths

When a unit dies while it is assigned to a certain task, that task should be given to another agent, or alternatively it should notify the manager that it cannot complete the task. However, when a unit dies, its agent is immediately terminated. It cannot inform other agents. Additionally, there is no way to know in advance when an agent will be terminated.

In order to tackle this problem, a 'shutdown module' would be a solution. A shutdown module would be executed when an agent is terminated. However, such a solution required significant changes in the GOAL language and was thus not a feasible solution in the short term. Instead, this problem was solved through the use of the `friendly` percept type, which can be used to determine when a friendly unit dies. This is not a perfect solution, however, due to the unstable nature of the communication with the manager that was explained previously. The death of a unit will not be forwarded to the manager correctly when it occurs while the agent tasked with forwarding this information has recently disappeared and has not yet been re-assigned.

5.3.4. Tournament Results

During the MAS Project, periodic tournaments took place. For each tournament, student teams were free to submit their bots. The tournaments followed a round robin format, in which every bot plays one game against every other bot on each available map. For the first tournament, only the map 'Circuit Breaker' was used. The results of this tournament can be seen in Table 5.1.

ForceBot's game performance in the first tournament was good, ranking 3rd place with a 73% winrate. ForceBot followed a simple strategy of constructing static defences in order to protect itself from early-game threats while amassing a large army inside of its base. Upon reaching a pre-determined army size, it attacked the enemy base. Although the strategy performed well within the tournament, there are many points for improvement. The games which ForceBot lost were most frequently caused by three issues.

Name	Games	Wins	Losses	Winrate
Group8	14	11	3	79%
NothingVenturedNothingGained	13	10	3	77%
ForceBot	**15**	**11**	**4**	**73%**
WeJustSpamSiegeTanks	13	9	4	69%
ATLEASTYOUTRIED	14	9	5	64%
WeJustSpamScouts	14	9	5	64%
AyyyLmaoSmallHope	13	8	5	62%
UndefinedBehaviour	14	8	6	57%
ProtossFTW	14	8	6	57%
iliketrains	15	7	8	47%
lllllllllIl	12	5	7	42%
GekkeBoys63	10	4	6	40%
ninejas	12	3	9	25%
TrolololoV3	11	2	9	18%
CoolMinecraftMod	11	0	11	0%
WifiParticileReceiver	13	0	13	0%

Table 5.1: Results of the first MAS tournament

First, this version of ForceBot critically fails if its build order is interrupted. This happens when a tech building such as a Spawning Pool is destroyed, or when a Drone responsible for constructing a building is killed. This will cause production to stop, leading to certain defeat. In particular, the deaths of Drones responsible for constructing a building were often the result of the Drone attempting to place the building towards the exit of the base, away from its defences. Future versions of ForceBot will need to assign these tasks to other Drones, as well as give preference to constructing non-defensive buildings away from the entrance to the base. Second, this version of Forcebot does not scout in advance. When it decides to attack, it

first needs to find the enemy base. This leads to its army becoming spread out and disorganised when the scouting takes some time. Third, due to a bug in the attack code, there were a number of games in which ForceBot did not attack at all. Such cases may result in a loss by time-out, despite possessing a superior army.

The next tournament was 3 days after the first. The map used for this tournament was 'Destination', and the results can be seen in Table 5.2. As there was not much time between the tournaments, many changes were not yet ready. Because of this, the same version was submitted once more. The results were significantly worse than the first tournament, with a winrate of 49%. Two reasons can be primarily attributed to this: first, the version submitted to this tournament was identical to the last, while most other bots had adjusted their strategies. Secondly, the first tournament was played only on a large map, where as the second was played on both a large and a small map. Early rushing bots gain an advantage on smaller maps. This meant that ForceBot, which did not use such a strategy at this time, was at a disadvantage compared to the previous tournament. In particular, its defensive buildings often completed too late to deal with highly aggressive enemies on the smaller map.

Name	Games	Wins	Losses	Winrate
DebuggingProbeLame	46	41	5	89%
WeGainedAndWeWantMore	45	39	6	87%
42	42	34	8	81%
Group8	42	31	11	74%
LagIRL	42	29	13	69%
NULL	43	28	15	65%
ThoMAS2G	41	26	15	63%
andyforpreezy	43	27	16	63%
AyyyLmaoChoking	42	25	17	60%
httpgooglMPVVkw	46	25	21	54%
CheesyMarines	45	24	21	53%
ForceBot	**45**	**22**	**23**	**49%**
Emmentaler	46	22	24	48%
TobysApprentices69	45	21	24	47%
mAkEtErRaNgReAtAgAiN	43	19	24	44%
MarineNotGoodToGO	39	17	22	44%
Kruidenkaas	37	15	22	41%
GratisBier	43	17	26	40%
HydraArmy	40	13	27	33%
TerranItUpANewHope	39	9	30	23%
Ninejas	44	10	34	23%
Snor	43	9	34	21%
TobySnippers	45	9	36	20%
ProtossOP	38	0	38	0%

Table 5.2: Results of the second MAS tournament

Based on the findings in this version of ForceBot, the following points for improvements need to be addressed for the following version of ForceBot, as their underlying issues were responsibile for a number of losses:

- Ensure that ForceBot always attacks once the required conditions are met.
- Ensure that the build order is always correctly executed.
- Transfer the tasks of a dead unit to another unit.
- Utilise `queueSize` when training new units.
- Improve defence, primarily on smaller maps.
- Scout the enemy's base in advance.

5.4. Design and Development of ForceBot: V3 - MAS Project

The primary goal of the third version was to address the findings of the second version, and generally improve ForceBot's game performance in the last MAS Project tournament as much as possible by prioritising minor changes that were expected to yield good results.

The first issue to be resolved was the issue where if units that were important to the build order were killed, the build order would halt. This was resolved by having the main Hatchery notify the manager of any friendly unit or building which is destroyed. The manager will then determine if the destroyed unit or building was important for the build order and act accordingly. If the destroyed unit was in the process of performing a task, that task is given to another unit.

The second issue was that of defence construction. The second tournament had shown that the construction was too late for the small map. To resolve this, ForceBot was changed to construct defences earlier depending on the size of the map. Additionally, ForceBot was adjusted to construct defensive buildings towards the entrance of the base, while tech buildings are placed away from it. This is to prevent enemy units from being able to circumvent ForceBot's defences.

Finally, the use of Overlords was improved. Previously, the location of the enemy base was not scouted in advance. In this version, Overlords are used to scout the map, as well as keep watch over base locations to spot enemy expansions. When attacking, expansions are prioritised over the main enemy base, as these are typically less defended. In order to keep the army from spreading out or becoming disorganised, the manager will inform all combat units which base they should attack, so that they will all attack together.

With these changes, all findings from the previous version were addressed with exception of utilising `queueSize` percepts when training new units, as this issue was difficult to resolve. Similar to the problem of spending resources, Larvae do not begin morphing immediately. The manager would need to take into account how many Larvae are slated to be used by each Hatchery, which made the issue difficult to solve. Therefore, the issue was not addressed, and other changes which were expected to yield a greater benefit were prioritsed instead. One of these changes involved the agent types used by each building type. In this version of ForceBot, each building type received its own unique agent type. Previously, a generic 'building' agent type was used for all buildings other than the Hatchery, but this was expanded in order to allow the buildings to perform more detailed reasoning about their new, specific tasks. For example, Extractors will now tell nearby Drones to harvest gas from it, while the Spawning Pool will determine when to start performing its researches.

5.4.1. Tournament Results

The third tournament was played using both maps previously used, 'Circuit Breaker' and 'Destination', meaning that each bot played twice against every other bot. The results of this tournament can be seen in Table 5.3. ForceBot finished the third MAS tournament with a win-rate of 83%, making it the best bot participating in the tournament. In particular, its heavy defences managed to prevent almost all losses to aggressive strategies, even on the smaller maps. The majority of the bots in the tournament were not designed to handle a situation in which they were not able to break through a defensive line. These bots would continually use small armies to attack ForceBot's defences, rather than assemble a larger army capable of destroying the defences.

Name	Games	Wins	Losses	Winrate
ForceBot	**41**	**34**	**7**	**83%**
100KGCOCAINE	41	33	8	80%
ZergRushIsForLosers	40	28	12	70%
ThoMAS2G	39	26	13	67%
TheSenate	41	27	14	66%
jaedong	42	27	15	64%
HappyWithOneWin	42	25	17	60%
life	42	24	18	57%
Group8	41	23	18	56%
WeJustTryMore	41	22	19	54%
ClickBait	38	19	19	50%
JamesBot	42	21	21	50%
AyyyLmaoFinalCountdown	40	19	21	48%
JeroenNietDown	41	19	22	46%
TobyIsBack	40	15	25	38%
EarthLings	38	13	25	34%
PerfecteKrul	39	13	26	33%
Munt	41	13	28	32%
TerranKillerSquad	39	12	27	31%
ZergCheese	40	12	28	30%
KAPOWRUSH	41	11	30	27%
zergminators	41	9	32	22%

Table 5.3: Results of the third MAS tournament

The games did reveal one weakness of the defence with regards to the placement of defensive buildings. The StarCraft-GOAL connector utilises construction sites in order to provide the worker units with places to construct buildings. However, these construction sites were difficult to work with. Construction sites utilised a 3x4 tile size, equivalent to the largest building in StarCraft. However, most buildings are smaller, such as 2x2 or 3x2. The large tile size of construction sites leads to a large amount of wasted space, where a small 2x2 building will be considered as block a large 3x4 tile. In particular, Vespene Geysers end up blocking construction sites in a large radius around it, even though there are valid locations for buildings. Furthermore, as a Zerg there is the added problem of requiring creep to be present. The connector, however, regarded the entire construction site invalid even if only a small corner is not covered in creep. As such, despite attempts to improve ForceBot's building placement, there were still many problems that need to be resolved in this area in order to account for the large construction sites.

This issue with placing constructions had a significant impact during two matches in the third tournament. In these games, ForceBot played against a Protoss bot which utilised an early Zealot rush. The Protoss bot would send out Zealots to attack the enemy base until it wins. ForceBot is weaker against rushes on smaller maps, as the enemy can reach its base faster. In spite of this, it lost on the larger map, but won on the smaller map. This was due to the problems surrounding construction sites. On the larger map, ForceBot's starting location was positioned in such a way that a Vespene Geyser blocked a large portion of the construction sites towards the entrance of the base. As such, ForceBot could not place its defensive buildings at ideal locations, and instead placed them to the side of the base, away from the entrance of the base. As a result of this, the poorly placed defences were not in range to attack the Zealots while they destroyed ForceBot's base. The small map did not have such a problem, however. ForceBot successfully constructed a wall of defences near the entrance of its base and survived the Zealot rush without problems.

Another problem that revealed itself in the tournament was that ForceBot had issues dealing with other defensive bots. As ForceBot did not expand, nor attack the enemy until it had created a large army, other bots were able to freely expand and gain a much stronger economy than ForceBot. Additionally, as Overlords stopped scouting in the later stages of the game, ForceBot frequently had troubles with locating enemy bases that are constructed at later stages of the game. As a result, the majority of ForceBot's losses in this tournament were against defensive bots.

These defensive bots also showcase a bigger problem: ForceBot is unable to assess when it is able to win. ForceBot lacks a means of comparing the strength of its own army to that of the opponent. As a result, it does

not immediately attack when it is able to beat the enemy, and does not retreat when it is in a losing position. This is a major problem which traditional bots address via the use of a combat simulator, which is able to estimate the winner of a fight between two armies.

Based on the findings in this version of ForceBot, the following points for improvements need to be addressed for the following version of ForceBot, as their underlying issues were responsibile for a number of losses:

- Addition of a combat simulator.
- Address the messaging problems of managers.
- Addition of expansion behaviour to avoid losing the economic advantage against defensive bots.
- Further improve defensive building placement to handle obstructions such as Vespene Geyesers.
- Scout bases at all stages of the game.
- Utilise `queueSize/1` when training new units.

Additionally, although the problems related to messaging percepts to the manager did not cause any losses in this tournament, it is an issue that should be addressed in a future version.

5.5. Conclusion

Over the course of this milestone, the development on the StarCraft-GOAL Connector had been directed primarily towards usability, as this was the first time that the connector was used by users outside of the developers of the connector. Furthermore, a number of percept types received additional parameters in order to provide agents with more information. The missing information was primarily oversights or information that had not seemed useful at earlier points in time.

A great deal of the development on ForceBot in this chapter lies in the fundamental design. ForceBot V2 added a manager agent which operates outside of the StarCraft environment. As such, it cannot be terminated, but it also does not receive percepts from the environment. This poses a problem, as other agents must message the manager with information instead. However, this results in either the risk of information being lost, or creating additional overhead by having multiple agents send information to the manager. During this milestone this issue has been resolved by using the starting building, a Hatchery, as the unit responsible for this task. However, this risks information being lost when the Hatchery is killed. As such, in the next milestone, a better solution should be created to deal with this problem.

In the MAS Project, ForceBot ranked #1 in the final periodic tournament. Although this was a good result, with the completion of the MAS Project, ForceBot's goal changed to competing in AIIDE. As the level of competition is much higher in AIIDE, there was still a need to improve upon ForceBot's design in subsequent versions. A major point for improvement lies in the addition of a combat simulator. As ForceBot lacked the means to estimate the winner of a fight, the behaviour of its army lacked decision making as a result. ForceBot attacks when a pre-determined amount of supply has been reached instead of when it believes that it can win a fight. One of the goals of the next milestone was to add a combat simulator to ForceBot to allow it to perform decision making while controlling its army.

Moving forward, a number of the issues that appeared are ideally addressed using improvements to the connector. For example, the placement of buildings, particularly static defenses, was poor as a result of the 3x4 tile size that the connector uses for the `constructionSite` percept type. Ideally, this tile size is made smaller in future versions of the connector. Additionally, the problem of a manager can be addressed by adding a dedicated manager entity within the connector, which is not connected to a unit but still allows for the agent connected to that entity to receive percepts like normal agents. Issues such as these are more easily resolved by using the connector, as these problems are caused by the level of abstraction being less than ideal in these areas.

6
Milestone 2: AIIDE

In the previous chapter, ForceBot was successful at besting the GOAL bots created by the teams for the MAS Project. In this chapter, the improvements made to ForceBot in order to successfully compete in the AIIDE tournament are discussed. The AIIDE StarCraft tournament is the largest StarCraft: Brood War AI tournament of the year, held over the course of September and October 2018. The strongest bots are present in this tournament. In the last chapter, it was concluded that the tactic which ForceBot uses does not work well against opponents that do not strictly use rush strategies. The bots that are competing in AIIDE use a variety of tactics, with many of them being able to learn which tactic is effective against which opponent. Hence, ForceBot needed to be improved in order to be able to deal with the level of opponents that are found in AIIDE. As such, one of the objectives was to improve the strategic components by being able to not only deal with a larger variety of tactics but also to be able to employ multiple tactics. If ForceBot only uses a single tactic, a learning bot will be able to easily determine a counter-strategy.

This chapter will discuss the development to the StarCraft-GOAL connector and ForceBot leading up to AIIDE, as well as the results of AIIDE. First, the changes that were made to the connector will be discussed in Section 6.1. These changes primarily alter the way in which percepts are received, and how managers are handled. Next, the further development of ForceBot is detailed in Sections 6.2 to 6.4. In addition to strategic improvements, a major part of ForceBot's development over the course of this milestone will be aimed towards developing a combat simulator which ForceBot now uses to predict the outcome of battles in order to be able to make an informed decision on whether it should or should not fight. A dedicated manager entity was also be added in order to resolve the manager-related problems discussed in the previous chapter related to the manager. Next, the results of AIIDE will be analysed in Section 6.6. Finally, Section 6.7 will summarise the progress and lessons learned during this milestone.

6.1. StarCraft-GOAL Connector Development

In the previous milestone, the connector was updated with backwards compatibility in mind as a result of the ongoing MAS Project. From this milestone onward, the MAS Project has ended, making ForceBot the only user of the connector. As such, the changes that have been made to the connector after the first milestone are based on ForceBot's development. For ForceBot, the most important aspect of the connector that must be improved is the handling of managers. As a result, the connectors development over the course of this milestone will focus on improving the way in which managers are supported by the connector.

6.1.1. Managers

Prior to version 3.0 of the connector, the creation of manager entities was not supported by the connector. It was possible to launch agents separate from the environment, which can act as managers, however this is a standard feature of GOAL, and not part of the connector. These agents are launched separate from the StarCraft environment, and as a result they launch immediately, before the StarCraft game has started. Additionally, these managers will not receive any percepts, as they are not connected to a unit. In order for managers to be aware of the state of the game, other agents must send it this information. However, this creates a number of issues.

First, not all unit types are privy to all information. For example, non-worker units do not receive information regarding resource locations. In order for a manager to receive all information, multiple agents must forward information. Second, if all agents forward all their information, a considerable overhead would be placed on GOAL. Ideally, only a single agent is responsible for this task in order to minimize the overhead. However, with only a single agent tasked with forwarding percepts, then in the event that the agent tasked with forwarding information is terminated, then a new agent must be assigned that task. In the time it takes to assign a new agent to this task, information may be lost. The way in which this can occur can be seen in Figure 6.1, in which the information that occurred between agent A's last cycle and the cycle before agent B registers agent A's death is lost. This loss of information can be avoided by having the newly assigned agent send all of its knowledge, but this creates the problem of ensuring that information isn't erroneously processed twice.

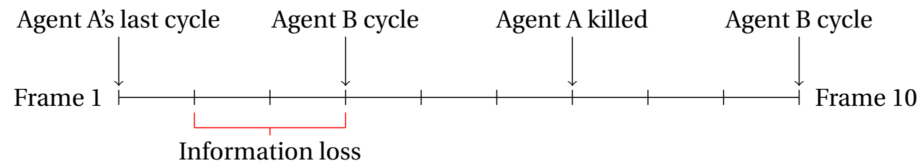


Figure 6.1: Loss of information if only a single agent is used for forwarding

The third issue is that, as the agents run in different threads, the manager agent will also frequently receive information about the same thing in different points in time (e.g. varying different states of the same enemy unit). Only the most recent state is relevant, however. Determining the most recent state would require a `gameframe` to be included with each message and stored with each percept so that they may be compared. The final issue is that agents and managers run in separate threads, meaning that they can start cycles independently from one another. As a result, the manager agent will occasionally start a new cycle while messages are still being sent, giving an incomplete or inconsistent picture of the current state.

The majority of the student teams participating in the MAS Project resolved the issue of the 'messaging unit' dying by utilising the main base building as the messenger. This building is among your starting units, meaning it is present from the start of any game. Although having this building destroyed will result in the manager no longer receiving any information, a game in which this building is lost is frequently already lost. Within the scope of the MAS Project, most teams accepted this pitfall for the sake of simplicity. This solution does create a problem for Zerg players, however, as morphing your main base from a Hatchery to a Lair will 'kill' the existing agent, replacing the Hatchery agent with a new Lair agent. Being unable to morph the main base into a Lair will hamper the build order for Zerg players. This situation was partially resolved by at least one of the MAS Project teams, who created a system of multiple 'messaging units' that will automatically take over when one died. However, this creates the problem of information loss which was explained by Figure 6.1, making the solution less than perfect.

In order to address all of the above issues, a new, optional manager entity was added to the connector in version 3.0. This manager entity receives all global percept types and is able to function independently without requiring other agents to inform it of the game state. This new entity simplifies the problem of creating a reliable manager agent within GOAL-based StarCraft bots, and ensures that the manager agent always possesses the most recent and fully complete game state.

When the manager entity is enabled, all agents will stop receiving all global static and global dynamic percepts. This is because agents do not necessarily use all percepts. Instead, the manager will serve as a filter, only providing information to agents that are actually interested in it. In this way, the total amount of information that is received by all agents is reduced, improving the computational performance of both the connector as well as the agents. However, this solution is not ideal. This is because agents in GOAL run all of their code each cycle, and only receive new messages/percepts in between cycles. As the manager becomes larger and more complex, cycles take a longer amount of time to process. In the case of ForceBot, manager cycles in the later stages of a game will often take more than 2 seconds as a result of the increasing number of agents and percepts. As a result, the manager entity will be working with outdated information towards the end of its cycle. To avoid the responsiveness of the manager becoming a bottleneck for the bot, the connector has been expanded in version 3.2 to allow for the creation of multiple of these dedicated manager entities. This change allows bots to split the load between multiple managers by having multiple managers making short cycles instead of a single manager making long cycles. This ensures that each manager main-

tains proper responsiveness for the tasks which it is responsible for.

Version 3.2 also added a new action which allows for managers to be started on-the-fly. This is a feature suggested during the MAS Project, as it allows for the dynamic creation of managers responsible for, for example, managing a group of units. Overall, the manager entities are an important milestone in the connector's development, they prevent the issues that occur when requiring other agents to transmit the game state to managers.

6.1.2. Subscribing to Percepts

Up to version 3.1 of the connector, the percept types that agents receive were fixed based on the unit-type of the unit to which the agent is connected. This lead to situations where, if a percept type has any conceivable purpose for a unit-type, it had to be included in the percept types which agents for such a unit-type received – regardless of whether they were actually used or not. As this is a costly process, the idea of a manager entity was introduced, as discussed in Section 6.1.1.

However, this approach turned out to be less than ideal. Though the manager was indeed able to intelligently decide what information to share and in an efficient manner, receiving and processing all percepts also placed a great load on the manager itself. This is problematic, because in this setup the agents are reliant on the manager in order to perceive the state of the game. Therefore, if the manager runs slowly, the agents will also be slow to receive the state of the game.

In order to address this problem, the feature to 'subscribe' to percept types was added. Instead of the received percept types being fixed within the connector, the bot may provide a list in the `mas2g` file of the bot on start-up. This list dictates which unit-types/managers receive which global percepts types – generic unit and unit-specific percepts are always received. This approach allows developers to only enable those percepts that they will actually use in their agents and thus reduce the load on the system, without requiring the manager to act as a filter. To illustrate this feature, the code snippet below shows the list of percept types that are used for Drones and Overlords by ForceBot. At the start of the list the unit-type is indicated, followed by each percept type that this unit-type wants to receive.

```
...
[zergDrone, chokepoint, gameframe],
[zergOverlord, enemyPlayer, map, region, enemy, gameframe],
...
```

For ForceBot, Drones use 1 out of the 5 global static percepts, and 1 out of the 11 global dynamic percepts that a Drone receives by default. Overlords use 3 out of 5 global static percepts, and 2 out of 7 global dynamic percepts that an Overlord receives by default. The new subscription model also allows agents to subscribe to percepts that they did not receive by default, such as receiving `mineralField` percepts as a non-worker agent. The need to subscribe to percepts that were not received by default is unlikely to be needed, however, as agents received every percept type that had a conceivable use for them.

To test the impact of this new subscription model, a number of games were played with identical setups. In each game, ForceBot played against the standard StarCraft Protoss AI, utilising the same strategy. Ten games were played with all percepts enabled, and ten using subscriptions. From the results in Table 6.1 it can be seen that number of percepts received each cycle is reduced by 30% to over 90% for each agent.

Unit Type	Total P/C	Required P/C	Reduction
Building	10.06	0.28	97.2%
Drone	29.04	0.77	97.3%
Hydralisk	10.21	6.13	40.0%
Larva	10.08	0.53	94.7%
Lurker	13.14	6.83	48.0%
Overlord	17.7	9.88	44.2%
Zergling	30.85	20.4	33.9%
Average	18.42	4.87	73.6%

Table 6.1: Number of percepts received per cycle

Furthermore, in order to remove the opponent from the measurement, Table 6.2 shows the number of percepts received per cycle when the `enemy` percept type is ignored. In this case, an even larger decrease

is visible. The difference between these tables also indicates that enemy percepts are the majority of all percepts received. Due to the number of units alive at once in a StarCraft match increasing as the game progresses, this difference will become larger over time. Therefore, the reduction in percepts received for units that do not need to receive enemy percepts, such as Larva and buildings, can be expected to become even greater over time.

Unit Type	Total P/C	Required P/C	Reduction
Building	1.31	0.28	78.6%
Drone	18.03	0.77	95.7%
Hydralisk	2.99	1.6	46.5%
Larva	3.68	0.53	85.6%
Lurker	1.73	0.83	52.0%
Overlord	5.03	0.82	83.7%
Zergling	7.95	0.84	89.4%
Average	8.59	1.13	86.8%

Table 6.2: Number of percepts received per cycle, ignoring enemy percepts

Additionally, agents in GOAL are only active as long as they are receiving percepts or messages, and will automatically sleep otherwise. By disabling almost all percepts for agents attached to buildings, the total number of cycles for these agents was reduced by roughly 98%, as the agents were able to sleep for long periods of time.

6.1.3. Percept Changes

The changes made to the connector are primarily aimed at how percepts are received by agents. Through the addition of built-in dedicated manager agents, as well as percept subscriptions, the managing of receiving percepts becomes easier to work with as well as more efficient and intuitive. Every agent can now receive the percepts that it is interested in directly from the connector, rather than needing to receive them from another agent.

The most important changes that were made to percepts revolved around the change of percepts to a subscription-based model. Notably, this led to a number of unit-specific percepts becoming global dynamic percepts. This involved the following percepts: constructionSite, mineralField and vespeneGeyser. All of these percepts were previously only available to worker units, however with the subscription-based model the choice of limiting this to only workers is left to the developer of the bot in question.

Finally, two new percepts were added: order and underConstruction. The information that these new percepts provide was previously difficult and imprecise to track. By providing direct percepts, this became simpler and always precise.

6.1.4. Global Static Percepts

base(<IsStart>, <Minerals>, <Gas>, <X>, <Y>, <RegionId>)

The <Minerals> and <Gas> parameters have been added to the base percept type, which show the total number of resources available at a base at the start of the game. Using this information, a bot can determine optimal bases to expand to.

enemyPlayer(<Name>, <Race>)

This percept types replaces enemyRace. In addition to providing the enemy's race, this percept type also provides the enemy's name. The enemy name can be used to allow the bot to learn effective strategies against a specific opponent.

map(<Name>, <Width>, <Height>)

The <Name> parameter has been added to the map percept type. As with the enemyPlayer percept type, this can also be used for learning. It can also be used to define behaviour specific to specific maps.

6.1.5. Global Dynamic Percepts

enemy(<UnitId>, <Type>, <Health>, <Shields>, <Energy>, <Conditions>, <Orientation>, <X>, <Y>, <RegionId>)

The <Orientation> parameter has been added to this percept type, indicating which direction the enemy is currently facing. For computational performance optimisations, <Orientation> is given in steps of 45 degrees. Orientation can be used to determine, for example, whether the enemy is currently fleeing or chasing.

friendly(<UnitId>, <Type>)
The <Conditions> parameter has been removed from the `friendly` percept type. The <Conditions> parameter was a list of status flags, such as 'idle' and 'underAttack'. The information that this parameter provided was rarely used, and as <UnitId> does not change, and <Type> rarely changes, its removal means that the `friendly` percept for a friendly unit will typically remain the same until that units death. As percepts which remain the same are more efficient to update, the removal of the <Condition> parameter makes this percept more efficient.

In the event that an agent is interested in more than just the type of a friendly unit, this can be transmitted through a message instead. The exception to this is while a unit is still under construction, as that unit will not yet have an agent attached to it in such a situation. In order to be able to receive important information about a friendly unit under these conditions, a new `underConstruction` percept was added. In all other situations, any important information about friendly units can be transmitted using messages instead.

gameframe(<Frame>)
The values associated with the <Frame> parameter of the `gameframe` percept type have been changed. Previously, this percept was received once every 50 frames. However, usage of the percept showed that a higher update rate is preferable. As such, `gameframe` has been changed to update every frame. Although this does have an impact on computational performance, as any agent will only receive a single `gameframe` percept per cycle, the impact on computational performance is negligible.

mineralField(<MineralId>, <Resources>, <X>, <Y>, <RegionId>)
The values associated with the <Resources> parameter of the `mineralField` percept type have been changed. The <Resources> parameter will now be rounded to multiples of 100. This is to prevent superfluous updates to `mineralField` percepts, as the precision of the <Resources> parameter is not that important.

underConstruction(<UnitId>, <Vitality>, <X>, <Y>, <RegionId>)
This new percept type provides information about units or buildings that are under construction. As units or buildings which are under construction do not have an agent attached to them, and thus cannot send messages to other agents, the information contained in this percept type was not previously available.

vespeneGeyser(<GeyserId>, <Resources>, <X>, <Y>, <RegionId>)
The values associated with the <Resources> parameter of the `vespeneGeyser` percept type have been changed. As with the `mineralField` percept type, the <Resource> parameter will now be rounded to multiples of 100.

6.1.6. Generic Unit Percepts

status(<Health>, <Shield>, <Energy>, <Conditions>, <Orientation>, <X>, <Y>, <RegionId>)
The <Orientation> parameter has been added to this percept type, indicating which direction the in which the unit is currently facing. The primary usage for this is in hit-and-run strategies, where a unit will attempt to run away while their weapons are cooling down. In this event, the <Orientation> parameter may be used to determine in which direction the unit should run away. As with the <Orientation> parameter in the `enemy` percept type, the angle is split into steps of 45 degrees.

order(<Primary>, <TargetUnit>, <TargetX>, <TargetY>, <Secondary>)
The `order` percept type has been added to address the difficulty of not always being certain whether an action was, and still is, being executed correctly. As StarCraft can cancel the current order on a unit (e.g. in case sight of the target is lost or the path to the target is blocked), this percept type can be used to detect when this happens so that the agent may act accordingly. Alternatively, it may be used to determine when the current order has been successfully completed.

6.1.7. Action Changes

debugdraw

Debugdraw is an action that has been added to assist in debugging. When an agent performs this action, text is drawn either above themselves (for units) or at the top-left of the screen (for managers). An optional <UnitID> argument can also be supplied to draw the text above another unit instead. The programmer can use this action in order to display debug information in real-time, as an alternative to logging. This feature is particularly useful when debugging issues or analysing behaviour on non-local environments such as SSCAIT, where logging is not available.

6.2. Design and Development of ForceBot: V4 - Manager

This version of ForceBot was developed using version 3.0 of the connector. Therefore, the primary objective of ForceBot V4 is to utilise the manager entity which was added to version 3.0 of the connector. In this version of the connector, the manager will receive all global percepts, while agents stop receiving global percepts. This means that the flow of global information between agents and manager has reversed, with now the manager being in charge of sending information to the units. This change in design structure called for changes in all agents, a number of which will be outlined below.

This version also marks a notable change in the design philosophy of ForceBot. Previously, it utilised a swarm-based control structure in all parts of its code. Agents would receive a task to complete, such as constructing a particular building or attacking a certain location, and each agent would determine an optimal approach to achieve that task according to its own beliefs and knowledge. However, this design created problems when managing the economic parts of ForceBot, in which agents will occasionally obstruct one another if not properly coordinated. For example, workers may decide to construct a building at the same location. This results in a deadlock, with neither worker being able to place the building, as both agents are blocking each other from placing the building. This also applies to resources, where uncoordinated resource management will lead to multiple agents attempting to use the same resources. Although there are swarm-based solutions to this problem, it is easier to utilise a central approach when possible. Furthermore, as the manager is privy to more information, it is able to make a better choice than individual units are able to. This is particularly important for major decisions such as the placement of buildings, where a poorly placed defensive structure can lead to a loss. Based on these observations, it was concluded that a swarm-based control structure is not desirable for these parts of StarCraft.

This is not true for all parts of StarCraft. Decisions related to combat (e.g. deciding which unit to attack) remain decentralised within ForceBot. This is because combat units must deal with a large number of enemies that are all moving around constantly, in addition to itself. It is an environment that requires quick response and highly unit-specific code. Coordinating this from the manager slows down the response time of the units, as agents will need to send their own status to the manager before it can make a decision. Therefore, the swarm-based control structure is still in use for combat related decisions.

Buildings

Buildings were an agent that was able to be assigned the role of forwarding information to the manager. The introduction of the manager entity meant that this task was no longer needed. Furthermore, up to this point, buildings themselves were tasked with determining when to begin performing a research. However, as upgrades are costly, the decision of when to spend resources on them is not a simple decision. Although it would be possible to transfer the information required to make an informed decision from the manager to building agents, it is simpler and more direct to let the manager make such decisions. Furthermore, as the manager was already in charge of all other means of spending resources as well, this means that the manager now has complete control and overview of economic decision making. This allows it to make the most informed decisions. These changes combined led to the agent code for buildings becoming very minimal, only needing to research and morph when told so by the manager, as all decision-making had been moved to the manager.

Unit training

Another objective for V4 of ForceBot relates unit training. As Zerg, all units are trained using Larvae, which periodically spawn at Hatcheries, Lairs and Hives. Previously, these buildings used the 'train' action to morph Larvae into the desired unit. Other races also train units exclusively using the 'train' action, making the approach intuitive at a glance. However, the approach has a number of problems.

First, Hatcheries cannot train units while it is morphing to a Lair or Hive, as units that are morphing

do not have an agent attached to them. This means that morphing to Lair is a fairly costly move that will hamper the build order, as its Larvae become temporarily unusable. Second, Hatcheries cannot train more than 1 unit per cycle/frame, while they may have up to 3 Larvae. For each Larvae, it needs to receive the order, execute it, and detect the successful start of training, before it can receive a new order. In the later stages of the game, this need for multiple cycles per order slows down the training process.

In order to address, this the manager has been changed to not send train orders to Hatcheries, but to the Larvae themselves. As Larvae are available even while Hatcheries are morphing, and a training order can be sent to any number of Larvae, both issues present when using Hatcheries are resolved. Furthermore, it simplifies the code as Hatcheries are removed from the training process.

Drones

In previous versions, Drones made many decisions on their own, such as where to place buildings. However, this leads to coordination issues when two Drones decide on the same place. In order to resolve such conflicts, the manager will now be tasked with determining where to place buildings. As building placement is important, the manager uses various information, such as the locations of chokepoints and enemy bases. By using a central approach, the information required to make such decisions does not need to be shared, making the approach more efficient.

This version also introduces a number of mineral gathering optimisations. The first optimisation is mineral locking, where workers are forced to stay at a specific mineral field. By default, workers will automatically move to another mineral field when another worker is already harvesting from the mineral field that they are trying to harvest from. However, moving to a different mineral field wastes a greater amount of time than waiting for the current mineral field to become available once more. The mineral locking technique requires an inhuman amount of actions per second for even just a handful of workers, making this technique is only available to bots. In order to reduce the CPU load, mineral locking is disabled after 7 minutes of in-game time. In addition to mineral locking, Drones are now also automatically assigned to the closest mineral patches first. Furthermore, the manager will assign Drones to specific mineral fields, ensuring that the Drones are split evenly across the mineral fields. These changes allow Drones to gather minerals faster than if the Drones are left unattended.

Combat units

In this version of ForceBot, combat units no longer receive `enemy` percepts. This is because in this version of the connector, percept subscription has not yet been implemented, and all global percepts are received solely by the manager entity. This resulted in a refactor to make the units receive this information through messages instead. Communication channels have been opened in order to allow a large number of agents to receive the information they desire from the manager. Separate channels exist for enemy air and enemy ground units. This approach allows each agent to decide which channel it is interested in and only receive information important to it. For example, Zerglings, who cannot attack air, will only subscribe to the enemy ground unit channel. This means that combat units do not need to check if they are able to attack the enemy – any enemy which they receive is a valid target.

Strategy

The use of multiple strategies is another addition to this version of ForceBot. Earlier versions would always construct a large amount of static defences, and then train an army of Mutalisks from behind the safety of its defences. While this tactic is effective against bots that only try to rush down ForceBot, it is a poor strategy otherwise. Any bot that focuses on its own economy will frequently overwhelm ForceBot, as ForceBot made no attempts to disrupt the opponent.

Instead of the overly defensive play, ForceBot V4 will use different tactics depending on the enemy race. Against Zerg, it builds only a few early defences to counter enemy rush attempts, while quickly assembling an army of Mutalisks to win the game with itself. Against Protoss and Terran, it builds almost no defences, instead opting for a fast expansion and building an army of Hydralisks and Lurkers.

This selection of strategies mimics the popular strategies in high-level human play, in which Zerg versus Zerg games are often won by whichever player trains Mutalisks first. This is due to Zerg lacking the means of fighting back against Mutalisks outside of training Mutalisks itself, making it important for players to be the first to train Mutalisks. Finally, in Zerg versus Protoss/Terran games, the Zerg player frequently wins by taking control of numerous bases in order to establish an economic advantage. Once

an economic lead has been established, the Zerg uses its wealth of resources to wear down and outlast the opponent.

Finally, a small tournament of ForceBot against the final MAS project bots was held. Overall, the game performance of ForceBot was notably worse compared to results of the third MAS tournament. This is primarily because ForceBot's tactic against Terran/Protoss was now to assume that they will not use rush tactics. However, the vast majority of MAS project bots are highly rush bots, as such the win rate against MAS project bots dropped from 83% to around 50%.

Overall, this result is not unexpected. This version of ForceBot has been designed to compete against SSCAIT/AIIDE bots. Unlike MAS project bots, SSCAIT/AIIDE bots are less likely to solely use rush tactics. These results make it clear that for future versions, ForceBot has to be capable of adapting to the enemy's strategy and dynamically determine how aggressive or defensive it should play.

6.3. Design and Development of ForceBot: V5 - Combat Simulator

This version of ForceBot was developed using version 3.1 of the connector. Previous versions of ForceBot use a simple strategy of building a single, large army. Once a certain fixed size is reached, it attacks with everything it has. However, since it is unable to predict which players has the stronger army, it does not know whether it will win this. Predicting the outcome of a fight between armies is the task of a combat simulator. This information can then be used to determine when a bot should fight and when it should retreat. Without a combat simulator, a bot is unable to make effective use of its army. For this reason, in order to further improve the attacking strategy of ForceBot, the next objective in ForceBot's development is to add a combat simulator.

There are a number of combat simulation libraries available, such as SparCraft [9]. However, these existing libraries are typically written in C++. It is possible to use these in GOAL, this would have to be done via actions within the connector, using arguments to tell the combat simulator the condition of the units. The result of this simulation would then be returned through a percept on the next cycle. This is not how actions within GOAL are typically used, as actions should perform commands on the environment, not mimic a function call. For this reason, as well as in order to further challenge GOAL, I decided to design and create a combat simulator within GOAL. If the performance of this combat simulator is insufficient, an external library for this task can be added to GOAL at a later stage.

6.3.1. Combat Simulator: Goals and Challenges

A combat simulator implementation has to be both fast and accurate. Speed is required, because a slow simulator will result in slow reactions to the constantly changing conditions of a fight. A combat simulator which lacks accuracy will either try to take a fight it cannot win, or refuse to attack when it is able to win. This creates a balancing act, in which an algorithm is required which balances speed and accuracy. Accurately simulating combat in a real game is complicated by many factors, determining the outcome is not straight forward. In order to perform a simulation accurately, a number of aspects are to be taken into account:

(a) The distinction between ground and air units, and the inability of certain units to attack one or the other.

(b) Attack cooldowns.

(c) Unit abilities. In StarCraft, certain units may deal damage through spells, heal other units or even act as suicide bombers.

(d) Upgrades which will affect the fight. Particularly, armour/attack upgrades will affect the damage taken/dealt by units.

(e) The time it takes for a unit to get in range to attack

(f) Targeting priorities such as 'nearest first', or 'lowest health first'.

(g) Collision amongst units, preventing some units from effectively attacking due to being blocked by allies.

(h) Micro management such as retreating with low health units and hit-and-run tactics.

Existing combat simulators typically function by simulating the next few seconds worth of game frames in a simulation that abstracts the actual game. These abstracted fights take many of the above aspects into account. However, not all of these factors are equally important. For example, SparCraft does not take into account unit collision (g) or micro management (h), as these are features that are computationally expensive to support. For the same reason, these features will also be excluded from the combat simulator for ForceBot.

Since Prolog, the language in which the combat simulator for GOAL will be programmed in, does not operate as fast as C++, a greater level of abstraction will be applied than that is used for C++ combat simulators. The combat simulator will include factors (a) and (c). This allows for a simple approach that sums the stats of all friendly and enemy units and determine the outcome assuming both sides continued to attack without stopping. In this approach, both armies are abstracted into a single ‘unit’ with the combined stats of each unit within that army.

The abstraction results in a number of factors becoming difficult to implement. As the armies are abstracted to act as a single ‘unit’, the notions of individual unit movement, attack cooldowns and targeting priorities are lost within the abstraction and cannot be accurately modelled. For this reason, factors (b), (e) and (f) will not be included. Finally, factor (d) will not be included as the connector does not provide a means of determining the upgrade levels of enemy units. The reason for this is that JNIBWAPI, the interface which the StarCraft-GOAL Connector uses to interact with BWAPI, does not support this correctly. ForceBot is able to track its own upgrade levels, however if only its own upgrade levels are taken into account then the simulation may determine a victory based on an upgrade advantage that it does not possess. For this reason, the combat simulator will simply assume that no player has performed and upgrades.

6.3.2. Combat Simulator: Version 1

In the first version of the combat simulator, each agent performs its own simulation. In order to make an accurate simulation, a number of changes needed to be made:

- Agents must inform other agents of each others location and health. This also means that the manager must inform agents when a friendly unit has died, so that they can delete the knowledge for that unit.
- The manager needed to send all information regarding enemy units to combat units, instead of only those that it can attack. For example, Zerglings cannot make accurate combat simulations if they do not know of nearby enemy flying units, even if they cannot attack them.
- Out-of-sight enemies must be remembered, so as to not run back towards a stronger enemy army the moment sight of it is lost.

Each agent will sum up the stats of nearby enemy and friendly units to predict the outcome of the fight. The simulation makes a number of distinctions, such as between units that can damage flying units and those that cannot. In total, the simulation has 24 parameters: both the health and damage for both friendly and enemy units for Ground-to-Ground, Ground-to-Air, Ground-to-Both, Air-to-Air, Air-to-Ground and Air-to-Both units. ‘Both’ refers to units which can attack both ground and flying units. This means that the simulation accurately accounts for targeting priorities so that flying units will not be intimidated by units that cannot threaten it. Note that the simulation does not account for unit locations (beyond the initial ‘nearby’ check) nor mid-simulation deaths.

This approach aimed to create a highly accurate simulation, however that aim resulted in a number of problems. For one, the definition of ‘nearby enemy and friendly units’ is complex. Some units are able to attack from very long range (e.g. Siege Tanks), while others need to be right next to the enemy (e.g. Zerglings). This can lead to erroneous judgements, such as a single Zergling believing it can win against a Zealot, when the Zerglings allies are still a significant distance away. This particular problem can be improved over time, but the real problem of this approach was the computational performance requirements. The approach requires every agent to run its own combat simulation, and every agent to be kept up-to-date of the state of every other friendly unit. In this way, the cost of messaging increases quadratically as more agents are added to the game. This problematic nature led to the idea of individual simulations being discarded, as the approach becomes too costly when the number of units exceeds a certain threshold as a result of the quadratic scaling.

6.3.3. Combat Simulator: Version 2

As the approach of individual combat simulations proved to scale too poorly, in version 2 the simulation has been moved to the manager. Instead of having each unit simulate its own fight, the manager will now

simulate a fight for all units on the map. However, if all units on the map are considered as being part of the same fight, the combat simulation will predict that it will win a fight inside the opponents base thanks to the help of friendly units still located in its own base. In order to address this problem, a combat simulation will be performed for each region on the map separately. This is more efficient than centring a simulation around a unit, as each map only features around 15 to 25 regions.

By using a region-based approach, units near the opponents base will be scared and back-off, as the combat simulation for that region will conclude that they will lose if they try to advance on the enemy base. On the other hand, units in ForceBot's base will advance towards the front-lines, as combat simulations in those regions will tell them they are safe. This will lead to the army naturally converging towards the same location, i.e. near the enemy's forces. In addition, by making the simulator region-based, the frequently-changing `(X, Y)` coordinates of units are no longer needed, as only the less-frequently changing `RegionId` is required, reducing the number of updates that are required for the combat simulator.

The simulation itself has also changed. The number of parameters in the simulation has been reduced from 24 to 10 – namely friendly and enemy values for Ground Health, Air Health, Ground Damage, Air Damage and Both Damage (i.e. units which may attack both ground and air units). The combat simulation will simulate a fight between the chosen region and the region that leads to the enemy's base. If the simulation determines a loss, it will order a retreat towards its own base. If the simulation determines a win, it will advance towards the enemy's base. This approach means that the number of combat simulations per cycle is at most equal to the number of regions found on the map. By limiting the maximum number of simulations per cycle, the problem of exponential scaling that was present in the previous version is removed.

6.3.4. Combat Simulator: Version 3

With the basic functionality of the combat simulator operational, the next step was to improve the accuracy of its predictions, as well as improve the way in which the outcome was handled. To this end, a 'commitment' threshold was introduced in order to improve the decision making ability of the combat simulator. A commitment threshold serves to ensure that the bot sticks to its current decision for longer. This is important, because disengaging from a fight can be a costly manoeuvre. While running away, the enemy can freely attack your units. Without a commitment threshold, ForceBot would choose to attack based on a 1% advantage, and then retreat based on a 1% disadvantage. Each time this happens, a costly retreat is initiated. By using a commitment threshold, the bot will be more reluctant to retreat once it has initiated a fight.

ForceBot will consider a simulated fight to be a win if it is won while keeping at least 30% of its own units alive. Once the fight starts, the combat manager will not call for retreat until the combat simulation indicates that at least 30% of the enemy units will survive. When the combat simulation believes a fight in the current region will be won, but a fight in the next region (towards the enemy base) will not be won, it will hold and defend its current region.

Furthermore, the simulation was changed so that, instead of using only the stats of units in the current region and the next in the path, it will account for all adjacent regions. This helps address for units that are not in a straight path towards the enemy base, but are still in close proximity of the combat area. Furthermore, there was an issue with units located in 'region 0'. Not all positions on the map belong to a specific region, causing units to occasionally belong to region 0. This is most prominently the case for air units that are flying over otherwise unwalkable ground terrain. This has been changed so that region 0 acts as if it is adjacent to every region. This is because air units will otherwise confuse the simulator by moving between regions not connected by ground and therefore not adjacent to one another. Friendly units circumvent this issue by using the last non-zero region they were in. This solution was not used for enemy units as it is possible that the first time they are seen is in region 0.

Finally, a number of changes have been made to address some finer unit balances within the combat simulator. Initially, all units used their exact in-game stats within the simulation. A 5 Damage Per Second (DPS) Zergling does exactly 5 DPS in the simulator. However, this leads to inaccuracies for a number of units. For example, Siege Tanks are often underestimated, as the simulation does not account for splash damage or their very long range. Units with such traits have had their DPS manually increased in order to more accurately simulate their actual contribution in a fight.

Terran Siege Tank

The damage of Siege Tanks has been roughly tripled in order to account for its massive range and splash damage.

Terran Firebat

The damage of Firebats has been doubled in order to account for its splash damage.

Terran Bunker
Bunkers do not actually attack themselves. Instead, units may enter a bunker. Units inside bunkers can then attack freely while being safe from harm until the bunker is destroyed. As the combat simulator only accounts for units that can attack, a bunker is ignored. To remedy this, Bunkers have been changed to deal damage as if it is fully loaded. This may mean that an empty bunker is considered as dangerous, but as GOAL does not provide a means of determining how full an enemy bunker is, it is the safest option.

Zerg Zergling
The damage of Zerglings has been halved in order to make for more accurate simulations. Although Zerglings have high DPS, they require melee range and will often times block each other when in large numbers. Especially in large numbers, Zerglings under-perform compared to their stats.

Zerg Lurker
The damage of Lurkers has been doubled in order to account for its splash damage.

Zerg Scourge
As Scourges attack via a high-damage suicide attack, their attack damage is many times higher than any other unit. As the simulator does not account for the fact that they will die after attacking once, their high damage causes the simulation to become highly inaccurate. In order to make their threat assessment more accurate, their damage has been lowered greatly.

Workers
Workers have been removed from the combat simulation. This change means that a small army of Zerglings will not be intimidated by an army of workers. Even though the workers may retaliate and kill the Zerglings, the economic damage dealt to the opponent by killing a number of its workers will often outweigh the loss of units.

Spell Casters
There are a number of spell caster units in the game which cannot attack. Most commonly used among such units are Medics and High Templars, but also Queens, Defilers and Science Vessels. These units have been given damage values estimated to be proportional to the threat posed by their abilities.

The approach to a number of these problems differs from other combat simulators:

- The splash damage of a Siege Tank, Firebat and Lurker is typically simulated by increasing the damage based on the opposing army size. Due to abstracting all units as a single entity, this approach is difficult to use for Forcebot.
- Bunkers face the same issues in other combat simulators, with similar solutions of assuming it is loaded with units. This is because determining the amount of units inside a bunker can only be done by counting the number of attacks being fired from it. This is not done by any bots with exception of less than a handful, as the benefit is not that big compared to simply assuming that it is full at all times.
- Melee range units such as Zerglings are partially addressed by simulating movement. Since all units are abstracted into a single entity in ForceBot's combat simulator, this approach is not usable for ForceBot. Additionally, there is presently no combat simulator which simulates collision due to the computational performance requirements of doing so. As unit collision is a major factor to the effectiveness of melee units such as Zerglings, other combat simulators instead decrease the damage of melee units based on the amount of melee units present, as larger numbers of melee units are more prone to obstruct one another.
- Suicide units such as Scourges are accurately simulated, with the unit being removed from the simulation upon completing its attack.
- Worker units are typically only included when they are attacking, and are otherwise ignored. This would be ideal for ForceBot as well, but the benefit is minor compared to the difficulty of implementing this behaviour. As such, this feature can be implemented at a later date.

- The approach to spell casters differ depending on the unit. Defensive spell casters such as Medics are typically simulated exactly as they behave, however for offensive spell casters such as High Templars, similar estimated damage values are used to determine their value in a fight.

6.4. Design and Development of ForceBot: V6 - Subscribe

The sixth version marks another major update to the functionality of the connector. This version uses version 3.2 of the connector, allowing for the specification of which agent types receive which types of percepts through percept subscriptions. This provides a simple and efficient means of receiving only the percept types that are needed for every agent, as any other percepts are simply turned off. All agents will now receive only the global percepts types that are required by their implementation, instead of it being pre-determined which percept types they will receive and requiring the use of messaging to 'receive' percepts types outside of that. As a result of this change, the manager no longer need to supply agents with percepts. Each agent will receive the percepts it requires directly from the connector.

In addition, version 3.2 of the connector allows for the specification of the number of manager entities. Each manager can be set up to receive different percepts. Using this feature, another issue is addressed in this version. With a single manager, each component of the manager can only run sequentially. If this manager starts to slow down as a result of being put under a heavy load, every component of the manager will slow down. This is problematic, as it is important for the manager to maintain responsiveness, especially for tasks such as combat-related decision making.

To address this, the existing manager will be split into multiple, separate managers. Each manager will be in charge of separate tasks and operate independently as much as possible. The objective is that the managers should not need to communicate much with one another. These objectives resulted in the creation of the following three managers:

Construction Manager
: The construction manager, nicknamed 'BuildMind', is responsible for placing buildings. This manager has been created because the `constructionSite` percept type is the most costly percept type to update due to how many percepts of this type are perceived every cycle. However, although this percept type is costly to update due to the quantity, the information is not often required. Therefore, this manager will only become active when a building needs to be placed, and will otherwise sleep.

Combat Manager
: The combat manager, nicknamed 'BattleMind', is responsible for tracking both friendly and enemy combat units, running the combat simulation, and determining where and when to attack or retreat. As managing the army is a critical task, this manager exists in order to prioritise combat-related decisions above other tasks.

General Manager
: The general manager, nicknamed 'OverMind', is responsible for all tasks not covered by other managers. This includes tasks such as managing worker units, training new units, determining when and which buildings to construct (but not where), determining which upgrades to research, etc.

These three managers are similar to the way in which many existing bots are split. In [32], 5 out of the 7 bots analysed perform decision making for economy and combat separately. Furthermore, 6 out of the 7 bots also have a separate construction planner or manager.

6.5. Design and Development of ForceBot: V7 - AIIDE

The seventh version is the version submitted to the AIIDE tournament. The main focus points of this version were computational performance improvements, robustness and fixing bugs. As many different scenarios will be encountered within a tournament setting, it is important that the bot is robust enough to handle these without problems, and that no bugs occur, which could lead to losses. For this reason, major changes will be avoided, as these may add unexpected behaviours. However, the handling of enemy percepts will receive major changes in order to address a major computational performance pitfall.

6.5.1. Enemy Targeting

In earlier versions of ForceBot, enemy percepts were stored in the belief base. Once stored, agents utilise the stored beliefs to find the closest enemy unit according to its targeting priorities. For example, a Mutalisk prioritises units that can attack air units over units that cannot.

In the new version, enemy percepts are not stored in the belief base. Instead, agents iterate over all enemy percepts. Upon finding a valid target, it stores that target as a belief. It will then continue to iterate over all enemy percepts, and whenever it finds an enemy which is closer than the existing target or of higher priority, it will replace the earlier stored belief with this newly found target. This new approach means that only as many inserts/deletes are required as it takes until it iterates over the closest enemy. Additionally, as long as the existing target remains closest, no further insert/deletes are required in future cycles.

As the old approach inserts every enemy percept as a belief, the new approach is equally performant in the worst case, which is when the iteration order results in every subsequent enemy in the iteration being of higher priority. Typically, however, the new approach will result in fewer inserts/deletes. Although this new approach did result in computational performance improvements, the design is not optimal due to deficiencies in the language features of GOAL. This issue is explained and discussed in detail in Section 9.4.1.

6.5.2. Robustness

Regarding robustness, the primary focus is to ensure that tactics are executed correctly and consistently. ForceBot's actions should remain consistent and stable even in the various situations that might occur within tournament games. In order to test how ForceBot responds to strange situations, ForceBot was submitted to the Student StarCraft Artificial Intelligence Tournament (SSCAIT). SSCAIT is a stream which runs all day and features a large number of opponents that will also participate in AIIDE. The games that play out there provide a clear image of how games in AIIDE will play out and what sort of situations ForceBot will encounter.

For example, one game showed ForceBot losing its main base. In this game, ForceBot still controlled other bases, however it was unable to reconstruct its lost buildings correctly as a result of there being no fall-back when a construction site in the main base is not available. Upon discovering this behaviour, a fall-back was added where buildings are placed outside of the main base in case no valid construction sites can be found within the main base. As a result of this, ForceBot has become able to reconstruct itself from any location. The addition of this fall-back also revealed that there had been another issue with building placement, as ForceBot began placing buildings outside of its main base even while it still controlled the main base. This indicates that are cases in which ForceBot runs out of space to place buildings within the main base. The addition of the fall-back ended up solving multiple issues at once.

SSCAIT results also showed ForceBot to have difficulties at later stages of the game as a result of the computational load increasing due to the number of units, and therefore the number of agents, increasing. Although attempts were made to improve computational performance, the issue was not resolved before the AIIDE deadline. Instead, the build orders and strategies that ForceBot uses have been adjusted to make its playstyle more aggressive. ForceBot will aim to end games before the computational load becomes a major burden and a liability.

Finally, responses to certain enemy unit types has been added. For example, ForceBot will now switch to Mutalisks upon encountering enemy Siege Tanks, as Mutalisks can deal with Siege Tanks effectively. Through such responses, the responsiveness of ForceBot towards the actions of the opponent has been improved.

6.6. AIIDE Results

ForceBot placed 27th on AIIDE, out of 28th competitors. The complete results can be found in Table 6.3. The poor results of ForceBot are not unexpected, previous AIIDE competitions frequently saw new competitors performing poorly in the first year of their entry. It is difficult for new competitors to be versatile and efficient enough to deal with the large variety of strategies employed by other bots. However, estimations based on its game performance in SSCAIT gave a higher expected ranking. Upon inspection of the replays made available after the completion of the tournament, it was found that the primary cause for its unexpectedly low game performance was due to a critical bug that prevented it from gathering vespene gas. This meant that for the entirety of AIIDE, ForceBot was restricted to only creating Zerglings, the only combat unit which did not cost any vespene gas. As all of ForceBot's strategies involve the eventual use of units which require vespene gas, this bug greatly obstructed its ability to function as intended within AIIDE. In spite of this, ForceBot did adjust to its abundance of minerals and lack of vespene gas correctly by producing large amounts of Zerglings, allowing it to achieve its final winrate of 17.97%.

Bot	Games	Win	Loss	Win%	AvgTime	Game Timeout	Crash	Frame Timeout
ZZZKBot	2966	2465	501	83.11	8:00	3	4	0
PurpleWave	2963	2440	523	82.35	13:27	15	25	0
Iron	2965	2417	548	81.52	14:19	117	83	0
cpac	2963	2104	859	71.01	9:45	5	3	0
Microwave	2962	2099	863	70.86	11:34	14	22	0
CherryPi	2966	2049	917	69.08	9:50	7	12	27
McRave	2964	1988	976	67.07	14:35	32	14	0
Arrakhammer	2963	1954	1009	65.95	11:37	11	14	1
Tyr	2966	1955	1011	65.91	13:09	18	13	0
Steamhammer	2964	1901	1063	64.14	10:32	11	4	0
AILien	2966	1729	1237	58.29	13:04	13	216	34
LetaBot	2955	1682	1273	56.92	16:48	119	34	0
Ximp	2962	1605	1357	54.19	18:46	42	205	14
UAlbertaBot	2968	1585	1383	53.40	10:54	59	74	0
Aiur	2965	1496	1469	50.46	13:51	68	53	0
IceBot	2955	1348	1607	45.62	17:16	134	24	0
Skynet	2958	1295	1663	43.78	11:40	29	4	0
KillAll	2965	1276	1689	43.04	10:56	22	4	120
MegaBot	2802	1200	1602	42.83	12:21	52	413	25
Xelnaga	2962	1099	1863	37.10	15:19	121	147	0
Overkill	2958	967	1991	32.69	18:00	128	18	1
Juno	2962	876	2086	29.57	14:07	174	16	0
GarmBot	2961	802	2159	27.09	15:21	40	8	0
Myscbot	2964	769	2195	25.94	13:41	75	4	6
HannesBredberg	2964	630	2334	21.26	14:09	62	8	1
Sling	2963	625	2338	21.09	16:21	147	64	0
ForceBot	**2960**	**532**	**2428**	**17.97**	**15:10**	**167**	**9**	**0**
Ziabot	2964	510	2454	17.21	10:08	25	67	0
Total	41398	41398	41398	N/A	13:23	855	1562	229

Table 6.3: AIIDE 2017 results

In order to properly explain the cause of this bug, prior context for the versioning of the Brood War Application Programming Interface (BWAPI) is required. In recent years, the stable versions of BWAPI have been 3.7.4, 3.7.5, 4.1.2 and 4.2.0, all of which are supported on SSCAIT. However, AIIDE does not support BWAPI 3.7.5, which is the version used by the GOAL environment. A few days prior to the submission deadline, the StarCraft-GOAL connector was adjusted to also function with BWAPI 3.7.4. Afterwards, each map that would be played on in AIIDE was tested to ensure that there were no problems with using BWAPI 3.7.4. At this stage of the bots development, however, it typically ended the games rather quickly against the built-in AI. As a result of this, a critical bug which did not reveal itself until later the stages of the game was not discovered. Upon completion of the Extractor, the general manager is meant to assign Drones to harvest vespene gas. However, when using BWAPI 3.7.4 the morphing of a Drone to an Extractor is handled differently, leading to the Extractor not being connected to an agent. As a result, the general manager is not notified about the location of the Extractor and no Drones are tasked with collecting vespene gas. Due to this bug, the results of AIIDE cannot be used as an accurate indicator for ForceBot's game performance against other bots.

6.7. Conclusion

During this milestone the flow of information within the connector has changed a great deal. Prior to version 3.0 of the connector, agents did not receive percepts, which complicated their usage as a result of agents being required to send this. However, since agents can be killed, this communication is inherently unstable. In order to address this problem, version 3.0 of the connector allowed for the creation of a manager entity that was able to receive percepts. However, when enabled, the agents no longer received global percept types. This meant that the flow of information between managers and agents had reversed, with managers forwarding percepts to the agents. This approach was less than ideal, as it meant that the manager was a limiting factor

on the speed at which agents could perceive the game state. This problem was solved by allowing agents and managers to 'subscribe' to percepts, allowing them to dictate which percept types they wanted to receive. This solution meant that the flow of information could now be removed, as both the agents and managers were able to receive exactly the percept types they were interested in. Furthermore, unused percept types could be disabled in order to improve computational performance.

In ForceBot V4 there was a movement away from a swarm-based control structure. The decision making regarding when to perform researches was moved to the general manager. As a result, structures no longer performed any decision making, and would only act as told to by the manager. Drones were also changed to perform fewer decision making, primarily in order to prevent Drones from making decisions that conflict with those of other Drones. Finally, this is also the first version of ForceBot to feature the use of multiple strategies.

A combat simulator was added in ForceBot V5. This combat simulator will simulate fights between all units in a region and any regions adjacent to it. Using the combat simulator, ForceBot can predict the outcome of battles in order to intelligently decide whether units in a specific region should attack, hold position or retreat.

In ForceBot V6 the manager was split into three in order to improve responsiveness of the manager by allowing the three separate managers to make shorter cycles, as well as run parallel to one another. In addition, the subscription model for percepts was first used in this version.

Finally, ForceBot V7 focused on the computational performance and robustness of ForceBot, in order to improve its consistency within AIIDE. However, as a result of a major bug that was not noticed in advance, the results of AIIDE do not allow for a conclusion to be made regarding the game performance of ForceBot. This also makes it difficult to draw any conclusions regarding what areas of ForceBot should be worked on next, as the games played in AIIDE do not properly showcase ForceBot's capabilities. Moving forward, a more reliable source of data for ForceBot's game performance will need to be established.

One of the research questions raised at the beginning of this thesis is with regards to the computational performance demands of GOAL. In particular, the question was raised as to whether the 1-on-1 mapping of agents to buildings and units was feasible and useful for an environment such as StarCraft. A preliminary answer to this question can be made based on the work performed during this milestone. As of ForceBot V4, structures no longer perform any decision making, as this has been centralised to occur within the general manager. Based on this it can be concluded that, at this time, a complete 1-on-1 mapping of agents to units and buildings is not useful. Buildings which are only able to perform researches and upgrades, as well as train units, do not perform any decision making and therefore have no need for an agent.

7
Milestone 3: SSCAIT

The third milestone is the annual Student StarCraft Artificial Intelligence Tournament (SSCAIT) tournament, held starting from December 20th, 2017. In addition to the 24/7 stream, SSCAIT also hosts a tournament once a year. This tournament has a special student category, in which bots created by single students are ranked in order to determine the best student bot. ForceBot will be competing in the SSCAIT tournament and qualifies for the student category.

The previous milestone did not yield good results. Due to a critical bug affecting ForceBot, it was not able to perform to the best of its capabilities in AIIDE. As a result, the games that were played in AIIDE do not provide a good indication on what areas of ForceBot should be improved upon next. In order to determine ForceBot's actual game performance, Section 7.1 introduces a local setup for testing ForceBot against existing StarCraft bots of various ratings. Next, the changes made to the StarCraft-GOAL Connector over the course of this milestone are discussed in Section 7.3. The changes to the connector are primarily aimed at improving the computational performance of ForceBot, as well as resolving ambiguity between losing sight of enemy units, and enemy units dying. Section 7.4 discusses the changes made to ForceBot over the course of this milestone, using the local testing environment in order to determine the effectiveness of changes made. Following this, the results of SSCAIT and the game performance of ForceBot are analysed in Section 7.5. Finally, Section 7.6 will summarise the progress made and lessons learned during the third milestone.

7.1. Internal Testing

At this stage of ForceBot's development, besting the default bots included in StarCraft itself has become a simple task. In order to continue to improve ForceBot, a means of measuring its game performance in a local environment is required. The best means of doing so is by observing ForceBot play against bots that are better than the default AI.

7.2. Internal Testing Setup

In order to measure ForceBot's game performance more accurately between tournament milestones, a selection of other StarCraft bots will be assembled in order to play against ForceBot. These bots can be acquired in three ways. First, a number of StarCraft bots are open-source, allowing easy access to these bots. Secondly, all bots who enter into AIIDE are required to release the source code for the version submitted to AIIDE [7], meaning that any of the AIIDE contestants are available as well, even if these bots are otherwise closed-source. Lastly, the binary files of any bot present on SSCAIT may be freely downloaded from the SSCAIT website [42]. Tournaments between the selected bots would take place using a fork of the StarCraft AI Tournament Manager created by Dave Churchill, which is used in AIIDE 2017. The fork itself was modified by Vincent Koeman for use in the MAS Project 2017 [8]. The same maps that were used in AIIDE 2017 will also be used:

- Benzene
- Heartbreak Ridge
- Destination

- Aztec
- Tau Cross
- Empire of the Sun
- Andromeda
- Circuit Breaker
- Fortress
- Python

One game will be played on each map, for a total of 10 games against each opponent. Finally, the selection of bots has been assembled with a number of objectives in mind:

1. Each of the three races should be equally represented.
2. Three 'tiers' of bots will be selected, roughly equivalent to easy, medium and hard opponents. The intent of this is to ensure that any changes made for the sake of beating more difficult bots does not result in a negative effect on easier opponents. Furthermore, it gives a clear target to aim for, e.g. obtaining a win-rate of over 50% against all medium opponents.
3. The bots should each perform different strategies. By ensuring that the opponents employ a variety of strategies. This is to best avoid the problem of 'overfitting' on the chosen bots and losing game performance against bots using strategies that are not present in this internal testing.
4. Bots with extremely niche strategies will be avoided, also called 'cheese' strategies by StarCraft players. A select few bots employ only cheesy strategies. Beating these bots will often require a specific type of counter play. Although it is important for ForceBot to be capable of handling such strategies, these bots do not serve as a good baseline for game performance measurement, as they provide little challenge once a counter strategy has been learned.

Using these criteria, a total of 9 bots were selected in order to test ForceBot's game performance. These bots will be used throughout the remainder of this thesis. Furthermore, they will not be updated, in order to provide a static point for comparison between the different versions of ForceBot. The following bots have been selected:

Easy Opponents

The 'easy' opponents were selected based on previous games on SSCAIT. ForceBot was known to perform well against the selected bots. This meant that these bots would serve as a good baseline to determine that changes made did not negatively impact its game performance against existing bots.

- **Lukas Moravec** – This Protoss bot can select from both early-game tactics to overwhelm the opponent, as well as more late-game oriented approaches of using Reavers to strengthen its army.
- **KaonBot** – A basic Terran bot that only uses Marines. It is fairly aggressive with expansions, often giving it an economic advantage over its opponents to outweigh its simple attack force.
- **Aurelien Lermant** – A macro-orientated Zerg bot that will quickly try to expand across the map. This bot also participated in AIIDE under the name 'GarmBot', ranking 23rd out of the 28 participants.

Medium Opponents

These opponents were selected based on ForceBot having a number of difficulties with them. Although far from impossible to defeat, these bots generally held the upper hand, improving to the point that these bots can be reliably beat is one of the main goals of these internal tests.

- **Gaoyuan Chen** – A Protoss bot that specializes in early and mid-game strategies, primarily using Zealots and Dragoons.

- **Sparks** – Unlike most Terran bots, which focus on biological units such as Marines, Sparks focuses on mechanical units. By using Sparks as a testing bot the effectiveness against a less commonly used archetype can be measured.

- **zLyfe** – A highly aggressive Zerg bot that only uses Zerglings, a fairly common strategy among Zerg bots.

Hard Opponents

'Hard' opponents were selected based on their excellent execution of their respective strategies. These bots will be challenging for ForceBot to beat reliably, however, a great deal of information can be learned from these difficult opponents.

- **Antiga** – A Protoss bot that uses a variety of strategies and is frequently seen using a number of the less commonly used Protoss units. In order to best Antiga, dealing with and responding to unorthodox units is important.

- **Simon Prins** – A defensive Terran bot that focuses on Marines & Medics. In order to beat this bot, ForceBot will need to learn to break strong defensive lines.

- **NLPRBot** – A strong and versatile Zerg bot that can utilize both early and mid game strategies. NLPRBot uses many of the same tactics as ForceBot itself does, but currently performs both its macro and micro better. As NLPRBot uses the same race, it is a bot from which ForceBot may learn a great deal.

7.2.1. Internal Testing Results

The first internal test tournament was held using the version submitted to AIIDE. This is both to establish a baseline for performance moving forward, but also to measure the game performance of the AIIDE version of ForceBot without the Extractor bug that prevented it from functioning as intended. The results of this tournament can be seen in Table 7.1, with the race of the opponent indicated before the name of each bot.

Opponent	Test #1 AIIDE
(P) Lukas Moravec	60%
(T) KaonBot	50%
(Z) Aurelien Lermant	100%
(P) Gaoyuan Chen	20%
(T) Sparks	40%
(Z) zLyfe	40%
(P) Antiga	10%
(T) Simon Prins	0%
(Z) NLPRBot	50%
Average	41.1%

Table 7.1: Win-rate against internal test bots

ForceBot's win-rate against Aurelien Lermant is an impressive 100%. This is impressive, as the win-rate of ForceBot against Aurelien Lermant in AIIDE is only 22.7%, in which it participated under the name 'GarmBot' and ranked 23rd out of the 28 participants. However, no other bot selected for the internal tests was present in AIIDE. Therefore, this is not enough data to estimate ForceBot's actual ranking in AIIDE if it had functioned as intended, but this does clearly demonstrate the extent to which ForceBot's game performance suffered in AIIDE.

The high win-rate against NLPRBot, relative to the other 'hard' bots, is an interesting result. The main cause for this is that NLPRBot primarily uses two different strategies: an early offence using Zerglings, or defensive play leading into quick Mutalisks. The AIIDE version of ForceBot only uses a single strategy: defensive play leading into quick Mutalisks. The outcome of each game was decided by the strategy that NLPRBot used: ForceBot would beat the early offence using Zerglings, but lost whenever NLPRBot utilised the same Mutalisk-based strategy as ForceBot itself used. The primary reason ForceBot would lose when both bots used the same strategy is that ForceBot played more defensively, spending a significant amount of resources on defences, despite NLPRBot showing no signs of aggression. This allowed NLPRBot to be the first to train

Mutalisks. Moving forward, ForceBot should be able to estimate the number of defences required based on the enemies combat potential.

The other problem is that NLPRBot's Mutalisk micro was superior to that of ForceBot. High-level Mutalisk micro requires a significant amount of actions per minute, however. ForceBot's computational performance does not currently allow such levels of micro. This makes it difficult to compete against the Mutalisks of other Zerg bots. To resolve this problem, either the computational performance will need to be improved, or a different strategy is required.

Additionally, the low win-rate against most Protoss bots was the result of ForceBot frequently being unable to defend itself from the first Zealot rush of the opponents. ForceBot will need to become better at defending itself from Zealot rushes moving forward. Finally, ForceBot could not break the defence of Simon Prins, losing every game. Against highly defensive opponents, a Zerg player should take control of the map to gain a resource advantage. In the future, ForceBot will need to become capable of responding in such a manner to defensive play by the opponent.

7.3. StarCraft-GOAL Connector Development

At this point in time, the connector had grown stable, with most of the work being directed towards computational performance improvements that did not affect users of the connector. One major change did happen to the connector, however. Previously, the `enemy` percept type would only give the position of all visible enemies on the map. If the bot wanted to store information regarding units that have disappeared from sight, this had to be explicitly stored as beliefs. Storing information from units that are out of sight is important for a number of things, such as accurate combat simulations. If enemy units are not remembered, the combat simulation will determine that the enemy has no defences as soon as the enemy units enter the Fog of War.

However, leaving the task of remembering enemies that have been lost sight of to the programmer was problematic in several aspects. If all enemy units ever seen are remembered, the combat simulation remember units that have already died, which would lead to erroneous combat predictions. This means that a mechanism for detecting when a unit has died is required. The simplest approach for this is by determining whether a unit should be visible, but is not. If this is the case, that unit must have died or disappeared. However, this approach is costly, as it means line-of-sight checks are required for every friendly unit to every enemy unit. Furthermore, this has a potential for false positives, as the unit may have simply moved to a different location while within the Fog of War.

Traditional bots utilise native events within BWAPI for this: `UnitHide` and `UnitDestroy`. These events allow for an enemy unit dying and disappearing to be distinguished from one another, as well as not requiring line-of-sight checks. In order for GOAL to provide the same level of detail and performance, the `enemy` percept type was extended with an an additional <Gameframe> parameter. This parameter contains the frame that the enemy was last seen, which is the current frame for visible enemies. Instead of the `enemy` percept being removed upon sight of the enemy unit being lost, it is only removed when BWAPI fires a `UnitDestroy` event for that unit. This means that an enemy unit disappearing and an enemy unit dying can not be distinguished from one another, as well as no longer requiring line of sight checks.

There does exist one problem to this change, however. That is, the `UnitDestroy` event within BWAPI does not fire when a unit dies within Fog of War. However, this issue is largely circumvented by ensuring that units do not try to attack units which are not currently visible. Furthermore, this problem also exists for traditional bots.

7.4. Design and Development of ForceBot: V8 - SSCAIT

The first task for this development cycle was integrating the changes to the `enemy` percept type into ForceBot. Primarily, this means that there is no longer a need to perform out-of-sight checks. Up till now, the combat simulator would store the `(X, Y` locations of every enemy unit, as well as a boolean indicating whether the unit was visible or not. Units which were not visible would have their `(X, Y)` locations compared against the sight range of ForceBot's units. As removing dead units was now a task of the connector, both these checks as well as the `(X, Y)` information was no longer needed for the combat simulator, resulting in a significant computational performance increase.

Afterwards, work on ForceBot's development primarily aimed at making ForceBot more flexible in its behaviour. For example, the number of static defences were previously a hard-coded amount. Instead of this, it would now use the results of the combat simulator to estimate an appropriate number. In order to have this process be more accurate, Overlords will now position themselves above the enemy main, as well as near

the entrance to ForceBot's natural. This will allow the Overlords to see the number of enemy units, so that the combat simulator can provide better estimates. When the enemy becomes able to attack flying units, the Overlords will stop this behaviour and retreat, so as to avoid dying.

In previous versions, vespene geysers would always have the maximum number of Drones assigned to it. As this would frequently lead to excessive amounts of vespene gas compared to Minerals, the allocation of Drones to vespene geysers became dynamically determined based on whether vespene gas is currently in demand. The change to vespene gas collection behaviour also allowed for new strategies: strategies that intentionally did not collect much vespene gas in favour of more minerals.

Finally, existing build orders suffered from problems in specificity. For example, if a build order was created which told ForceBot to construct a Spawning Pool and an Extractor once a total of 9 Drones have been created, it would always create an Extractor first, as it is cheaper to construct. The build order system was therefore altered so that ForceBot would use the exact order specified.

After these changes were made, the second internal test tournament was held on November 3, 2017. The results of this tournament can be seen in Table 7.2. The game performance of ForceBot is overall worse, primarily as a result of changes to build order and behaviour that had not yet been thoroughly tested. For example, ForceBot would now estimate an appropriate number of static defences based on the enemy army that it has seen. However, the estimate was not always safe enough, leading to losses against NLPRBot's Zergling rush. The AIIDE version did not lose to this, as it would always place the same number of static defences.

Opponent	Test #1 AIIDE	Test #2 Nov. 3
(P) Lukas Moravec	60%	40%
(T) KaonBot	50%	80%
(Z) Aurelien Lermant	100%	90%
(P) Gaoyuan Chen	20%	30%
(T) Sparks	40%	20%
(Z) zLyfe	40%	10%
(P) Antiga	10%	10%
(T) Simon Prins	0%	0%
(Z) NLPRBot	50%	30%
Average	41.1%	34.4%

Table 7.2: Win-rate against internal test bots

A number of the new build orders did not perform as well as expected. When ForceBot ignored vespene gas collection to prioritise minerals, it would often ignore vespene gas for too long, resulting it in falling behind significantly on technology. Furthermore, it did not produce enough Hatcheries using the additional mineral income, causing it to be unable to spend all of its resources. In order to address this, Hatcheries, as well as a number of other decisions to spend resources, would now be based on the amount of workers available. These decisions are as follows:

- When to construct additional Hatcheries for more Larvae. The previous version of ForceBot would do this only when ForceBot had a surplus of resources, however in this case the Hatchery will be completed late, causing it to float a surplus of resources for some time before. By taking into account the number of Drones and the resource demands of the current strategy, the required number of Hatcheries to needed to continuously train units can be constructed before ForceBot begins to float resources.

- When to expand. The previous version did this periodically based on its current strategy. ForceBot V8 will attempt to expand if it wants to construct an additional Hatchery and ForceBot believes it is safe to expand.

- When to switch to a different strategy. This is important, as switching to an expensive strategy when there are insufficient workers will lead to a shortage of resources.

- The amount of Overlords, Lurkers and Ultralisks to train simultaneously. As sufficient supply is needed to train additional units, Overlords should be trained in advance based on the current income. Additionally, Lurkers and Ultralisks are expensive and strong units, but it is not desirable for the entire army

to consist of them. As such, the amount trained is now based on the current economical strength and existing number of these units.

7.4.1. Lurker Behaviour

Zerg is able to produce Lurkers, a unit which can burrow, making it immobile and invisible. While burrowed, it is able to attack enemy units. Lurkers are a powerful unit against Terran and Protoss when they do not yet have detection, particularly against bots which often respond poorly to invisible units. A number of ForceBots' strategies are focused on Lurkers as a result.

ForceBot used Lurkers aggressively, sending them out alone. The reason for this behaviour is because they are difficult to kill when burrowed unless the opponent has mobile detection. Furthermore, many Terran bots rely on Comsat stations, which have a limited amount of energy that they can use to detect Lurkers. By sending out individual Lurkers, the aim is to exhaust their energy in a cost-efficient way. This strategy saw success, but primarily in bots which responded poorly to Lurkers. Many stronger bots did not have a problem dealing with Lurkers without the support of ForceBots' main army.

In order to improve behaviour against stronger bots, Lurkers were changed to stay with the main army instead. This change led to improved game performance against bots capable of dealing with the previous behaviour, but reduced game performance against those who did not respond well. As both approaches have their own advantages, ForceBot was changed once more to send out Lurkers alone as long as the opponent does not possess mobile detection. Once the opponent possesses mobile detection, the Lurkers will stay with the army. Furthermore, as the effectiveness of Lurkers decreases at this stage, ForceBot will train less Lurkers once this happens.

7.4.2. Start-Up Difficulties

Stability is an important factor in tournament settings. Any time a bot crashes or fails to start, the result is a loss for that bot. Therefore, stability is once more a focus in the lead-up to the tournament. ForceBot had been suffering from inconsistencies in its start-up process, with sometimes not all initial agents becoming active due to ordering conflicts. The cause for this has been difficult to track down, as behaviour varied from machine to machine due to the parallel nature of the agents. However, this bug occurred most frequently on the SSCAIT server. In order to not negatively affect the results of the tournament, solving the issue became a priority. This bug was discovered to have two different causes:

The first cause was an inconsistency in start-up sequence. Sometimes, all managers started before agents, other times they did not. In order to address this problem, agents were made to send out their start-up messages continuously until the second gameframe is reached. The connector will pause StarCraft until a total of four actions have been made, corresponding to the first four `gather` commands of each of the four starting workers. Therefore, so long as each Drone continually sends out start-up messages, and does not perform more than one command each, the start-up process is guaranteed to succeed.

The second cause of the bug was the communication channels: on start-up, all manager agents will subscribe to the 'managers' channel. However, messages appeared to be getting stuck in the system. When this happens, the message will be sent, but not received. This problem has been resolved by having the agents send their messages not towards the 'managers' channel, but towards each manager directly. The problem was that if a communication channel is created by two agents or managers simultaneously, only one would succeed in creating and subscribing to the channel. This prevented the start-up messages from arriving at the agent or manager that failed, leading to the start-up problem. This problem has since been resolved in GOAL, so that units correctly subscribe when the action is performed simultaneously.

7.5. SSCAIT Results

For the SSCAIT 2017/18 tournament, a total of 78 bots competed against each other over the course of 25 days, starting on December 20th, 2017. The tournament was split into two phases. In phase one, each bot played twice against each other bot, for a total of 154 games each. In phase two, the top 16 ranked bots from phase one would face off against each other in an elimination bracket. The top 16 ranked bots from this round robin phase would then enter into a further elimination tournament [5]. The results of phase one can be seen in Table 7.3. As ForceBot placed 33rd in the first phase of the tournament, it did not participate in the second phase of the tournament.

SSCAIT further distinguishes participants as either 'mixed' or 'student' bots, indicating whether the bot has been created by students or not. Among the 78 participants, a total of 25 qualified as student bots. Out of

these 25, ForceBot ranked 7th. Overall, the results are satisfactory. While ForceBot might not have qualified within the top 16, it scored an above average result and was one of the stronger student bots. Furthermore, the cross-table results against each other bot [1], show that ForceBot was capable of beating a number of the top competitors on at least one occasion. ForceBot successfully took games from Bereaver, Steamhammer, CherryPi and Microwave, ranked #5, #6, #8 and #9 respectively.

Results also showed that ForceBot performed poorly against highly aggressive opponents, with ForceBot losing before it can create an army of its own. In some cases this occurred against bots ranked significantly lower than itself. This primarily occurred on larger maps, where ForceBot was more prone to use strategies that could not deal with early aggression. This is an area for improvement in the future.

Name	Wins	Losses	Winrate	Name	Wins	Losses	Winrate
Iron bot	144	10	93.5%	Yuanheng Zhu	77	77	50.0%
tscmoor	140	14	90.9%	WillBot	77	77	50.0%
McRave	139	15	90.3%	Gaoyuan Chen	77	77	50.0%
Marian Devecka	135	19	87.7%	MegaBot2017	76	78	49.4%
Bereaver	132	22	85.7%	Lukas Moravec	72	82	46.8%
Steamhammer	128	26	83.1%	Tomas Cere	71	84	45.8%
WuliBot	124	30	80.5%	Ecgberht	70	84	45.5%
CherryPi	124	30	80.5%	Aurelien Lermant	69	85	44.8%
Microwave	123	31	79.9%	Matej Istenik	69	85	44.8%
TyrProtoss	123	31	79.9%	KaonBot	67	87	43.5%
Neo Edmund Zerg	123	31	79.9%	Pineapple Cactus	66	88	42.9%
Tomas Vajda	118	36	76.6%	Roman Danielis	66	88	42.9%
Andrew Smith	115	39	74.7%	Niels Justesen	61	93	39.6%
Arrakhammer	115	39	74.7%	Bryan Weber	60	94	39.0%
AILien	109	45	70.8%	HOLD Z	60	94	39.0%
Martin Rooijackers	109	45	70.8%	AyyyLmao	54	100	35.1%
Andrey Kurdiumov	109	45	70.8%	NUS Bot	50	104	32.5%
ZurZurZur	108	46	70.1%	DAIDOES	50	104	32.5%
Black Crow	108	46	70.1%	Marek Kadek	49	105	31.8%
Dave Churchill	107	47	69.5%	OpprimoBot	49	106	31.6%
Sijia Xu	106	48	68.8%	Oleg Ostroumov	48	107	31.0%
KillAlll	105	49	68.2%	Marine Hell	47	107	30.5%
ICELab	105	49	68.2%	igjbot	45	110	29.0%
Flash	103	51	66.9%	Travis Shelton	44	110	28.6%
NLPRbot	102	52	66.2%	Kruecke	43	111	27.9%
Carsten Nielsen	102	53	65.8%	auxanic	40	114	26.0%
NiteKatT	100	54	64.9%	Korean	39	115	25.3%
Zia bot	98	56	63.6%	Goliat	31	123	20.1%
CasiaBot	97	57	63.0%	Bjorn P Mattsson	26	128	16.9%
Hannes Bredberg	93	61	60.4%	FTTankTER	21	133	13.6%
PurpleCheese	90	64	58.4%	Hao Pan	19	135	12.3%
Soeren Klett	89	65	57.8%	JEMMET	19	135	12.3%
ForceBot	**88**	**66**	**57.1%**	Blonws31	17	137	11.0%
Florian Richoux	87	67	56.5%	UC3ManoloBot	17	137	11.0%
Dawid Loranc	83	71	53.9%	100382319	17	137	11.0%
MadMixP	83	71	53.9%	Lluvatar	12	142	7.8%
Jakub Trancik	83	72	53.5%	Guillermo Agitaperas	11	143	7.1%
PeregrineBot	81	74	52.3%	Laura Martin Gallardo	10	144	6.5%
UPStarCraftAI 2016	79	75	51.3%	MorglozBot	7	148	4.5%

Table 7.3: Results of the SSCAIT Round Robin

7.5.1. Internal Test Results

The third test tournament was held using the version of ForceBot submitted to the SSCAIT 2017/18 tournament in order to compare them to previous versions in an identical environment. The results of this tourna-

ment can be seen in Table 7.4. The version showed significant improvements over the previous version. In particular, it was now able to beat Simon Prins, the only bot that had been undefeated prior to this.

Opponent	Test #1 AIIDE	Test #2 Nov. 3	Test #3 SSCAIT
(P) Lukas Moravec	60%	40%	60%
(T) KaonBot	50%	80%	80%
(Z) Aurelien Lermant	100%	90%	80%
(P) Gaoyuan Chen	20%	30%	50%
(T) Sparks	40%	20%	60%
(Z) zLyfe	40%	10%	70%
(P) Antiga	10%	10%	30%
(T) Simon Prins	0%	0%	20%
(Z) NLPRBot	50%	30%	20%
Average	41.1%	34.4%	52.2%

Table 7.4: Win-rate against internal test bots

The biggest issue that occurred during this tournament was that ForceBot failed to start in 11 out of 90 games, resulting in losses by default. For example, all losses against Aurelien Lermant were due to start-up failures. When the games that were lost as a result of a crash are not counted, ForceBot's average win-rate was 59.5%. The cause of this is due to a change in the start-up code performed in Section 7.4.2. In previous versions, agents would execute their start-up code while the `gameframe` percept was lower than 100. In this version, the start-up code was only executed when `gameframe` was exactly 1. However, in some games during the internal test tournament, the gameframe on start-up was 0. This problem did not occur in SSCAIT, likely due to the difference in computer specifications.

Aside from the start-up problems, the performance against each bot was good, with a win-rate of 50% or greater against all but the 'hard' bots. Most of ForceBot's losses were due to a poor choice of build order against the build order of the opponent. This was particularly frequent against Protoss, where ForceBot would not have enough defences to protect itself from a Zealot rush. This problem has been noted before, and the SSCAIT version of ForceBot has strategies that can effectively deal with this. However, not all build orders are capable of defending against it. Another example is the games in which ForceBot beat Simon Prins. In all games where ForceBot won, it did so as a result of choosing its most economy-heavy build order, allowing it to gain an economic lead over Simon Prins' highly defensive strategy.

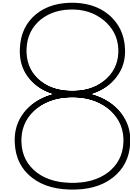
Moving forward, there are two approaches that can be used to improve ForceBot in these situations. First, learning behaviour can be utilised to improve the selection of build order. By using past games as reference, ForceBot can make an informed decision on which strategy it should use against its current opponent. Second, ForceBot's ability to adjust its strategy based on information about the enemy can be improved, by for example focusing heavily on economy when detecting defensive behaviour from the opponent.

7.6. Conclusion

In this milestone, test tournaments were held using a local computer setup to serve as an additional source of game performance data for ForceBot. In these internal test tournaments, ForceBot competes against 9 bots of various ratings. The first tournament was held using the AIIDE version. This gave an indication of the extent to which it affected ForceBot's game performance within AIIDE. Additionally, the internal test tournament provided for a good source of information for further development, as the games played at AIIDE were not representative of its actual capabilities. Another test tournament was held using a version in the middle of this milestone. The second test tournament showed a decrease in the game performance of ForceBot as a result of a number of new build orders and strategies that were not present before. According to these results, they were improved upon or removed for the final SSCAIT version.

The connector was updated so that `enemy` percepts did not disappear until the enemy died, rather than as long as there is vision of the enemy unit. This addressed a problem with the death of an enemy unit and an enemy unit moving into the Fog of War being indistinguishable from one another. Prior to this change, a performance-intensive and unreliable approach had to be used within GOAL, while the connector is able to perform this task efficiently by using built-in BWAPI events.

ForceBot V8 performed well in the SSCAIT 2017/18 tournament. It was able to rank in the upper half of bots, and 7th place amongst student bots. The results of the internal test tournament using the same version of ForceBot also showed a significant increase in game performance over the AIIDE version, with the win-rate increasing from 41.1% to 59.5%. Both the results of the SSCAIT tournament, as well as the internal test tournament using the same version, showed that a large part of the outcome of games was decided by the build order selected by ForceBot at the start of the game. Moving forward, future versions of ForceBot will need better responses to the opponents strategy in order to reduce the extent to which this is a limiting factor on ForceBot's game performance. Another approach is to utilise learning behaviour: by storing the outcome, build order and strategies used, ForceBot could utilise this data to make an informed decision on which build order and strategy it should use the next time it encounters the same opponent.



Milestone 4: SAIL

The previous chapter covered the development over the course of the last milestone in the development plan of ForceBot, as well as the results from the SSCAIT 2017/2018 Round Robin tournament. In SSCAIT, ForceBot lost against a number of bots ranked significantly lower, generally as a result of poor strategic responses or opening build orders that could not defend themselves sufficiently. In this chapter, the development of ForceBot after the SSCAIT Round Robin is detailed, which for a large part focuses on reducing the number of losses to lower ranked bots. Initially, only internal testing and SSCAIT ladder games was planned to take place during this time, however a new ladder, the StarCraft Artificial Intelligence League (SAIL) launched on March 20th, 2018, providing an additional source for testing the game performance of ForceBot.

Over the course of this final development stage, which took place primarily between January and March 2018, no changes have been made to the StarCraft-GOAL Connector, as there were no parts that required changes in order to improve the performance of ForceBot. This chapter begins with discussing the changes that occurred in Section 8.1, as well as the internal testing results of these changes. Afterwards, the results of SAIL are discussed in Section 8.2. Finally, the lessons learned and progress made on ForceBot over the course of this milestone is summarised in Section 8.3.

8.1. Design and Development of ForceBot: V9 - Final

Development of ForceBot V9 focused on the results of the SSCAIT 2017/2018 Round Robin tournament. In this tournament, a number of games against significantly lower ranked bots were lost as a result of ForceBot selecting strategies that were not safe against highly aggressive play. In particular, ForceBot would neglect defences on larger maps, on the assumption that the enemy is less likely to choose an aggressive strategy. In this final version, changes were made to reduce the number of losses to such strategies by utilising more safe strategies that are not as easily taken by surprise.

To this end, the build order against Protoss, which closely resembled the build orders used against Terran, has been changed to a new ‘10-Hatchery’ strategy, which is a safe strategy against common early aggression strategies by Protoss. Additionally, ForceBot will only utilise strategies which are safe from ‘4-Pool’ strategies against other Zerg players. Finally, a number of strategies which found little success against Terran within the SSCAIT Round Robin were removed entirely. Terran strategies will still often neglect defence, however, as highly aggressive Terran strategies are rarely encountered.

In order to streamline the execution of the build orders, ‘predictive building’ has also been added. What this means is that ForceBot will send Drones towards the location at which a new building should be constructed, even before all resources for that building have been collected. Using the current number of Drones and the distance to the construction site, it sends out Drones so that they will arrive at the destination around when the required resources will have been collected. This is particularly important when expanding, as Hatcheries can often be constructed 10 to 20 seconds earlier using predictive building, allowing early natural expansions to be better defended.

Finally, ForceBot will now utilise Guardians, a previously unused unit type. Guardians are late-game flying units which are particularly strong against Terran and in some cases Protoss. Although it is preferable for ForceBot to win its games before it reaches this stage, Guardians allow for ForceBot to break long-term stalemates, particularly against Terran bots which rely on Marines, which is the case for many Terran bots.

8.1.1. Enemy Strategy Detection

In StarCraft, players will continually attempt to counter each others strategy in order to gain an advantage. If players do not respond to the opponents strategy, they will quickly start to fall behind. This is less of a factor in games between bots, as many bots respond poorly to changes in strategy. As a result, ForceBot's responses to enemy strategies has thus far remained simplistic: it can adjust the combat units it produces depending on which enemy unit types it has spotted. It also estimates how many defences it should construct depending on how its own army compares to that of the opponent.

In this final version, one of the goals was to expand upon these responses to the opponent. In addition to responding to the enemy units it can see, it should also respond to the strategy used by the opponent. For example, as a Zerg player against an opposing Protoss player, the Protoss may choose to do a Fast Forge Expand (FFE), a strategy in which a Protoss quickly takes its natural expansion while defending itself with Photon Cannons. In such a case, the Zerg should respond either by playing very aggressively to defeat the opponent before the opponent can benefit from the resource advantages of its fast expansion, or rapidly expand itself to avoid falling behind in economic power. In the case of ForceBot, its strategy would always gravitate towards a 'balanced' playstyle, one which does not focus heavily on aggression nor economy. In many games a balanced playstyle is a good strategy, however against a Protoss FFE, a balanced playstyle will lead to a loss, as specific counter strategies are required.

Based on the games in SSCAIT, as well as those from internal testing games, ForceBot was made to recognise a number of strategies that the opponent may use:

- Protoss – Cannon rush
- Protoss – Carrier rush
- Protoss – Fast Forge Expansion
- Terran – Bio strategy
- Terran – Mech strategy
- Zerg – Mutalisk strategy
- All races – Defensive 'turtle' strategies

Upon detecting one of these strategies, ForceBot will respond by prioritising aggressive or economic play, as well as building counter units in advance. Additionally, when detecting a FFE or Carrier rush, ForceBot will also perform a 'runby', where it will ignore static defenses in order to deal economic damage by killing enemy workers. This is an effective strategy, as the Protoss player is unlikely to have Zealots to protect itself with. If the static defences can be circumvented, the Zerglings can freely attack the Probes. Additionally, in order to better detect certain strategies, particularly against Protoss players, ForceBot will now use Overlords to scout the enemies natural expansion, as well as keep watch on the entrance to ForceBot's main in order to detect cannon rushes. Finally, ForceBot will now utilise Scourge and Defiler units in order to combat enemy Carriers when the enemy is found to be using a Carrier rush strategy. These units were previously unused by ForceBot.

A fourth internal testing tournament was held on January 7th 2018, after enemy strategy detection was implemented. The results of this tournament can be seen in Table 8.1. Notably, ForceBot achieved an average win-rate of 50% against the 'hard' testing bots, clearly showcasing its improvement.

Opponent	Test #1 AIIDE	Test #2 Nov. 3	Test #3 SSCAIT	Test #4 Jan. 7
(P) Lukas Moravec	60%	40%	60%	80%
(T) KaonBot	50%	80%	80%	80%
(Z) Aurelien Lermant	100%	90%	80%	90%
(P) Gaoyuan Chen	20%	30%	50%	70%
(T) Sparks	40%	20%	60%	40%
(Z) zLyfe	40%	10%	70%	60%
(P) Antiga	10%	10%	30%	50%
(T) Simon Prins	0%	0%	20%	60%
(Z) NLPRBot	50%	30%	20%	40%
Average	41%	34%	52%	63%

Table 8.1: Winrates against internal testing bots

The strategy responses were highly effective, with the win-rate against Simon Prins increasing from 20% to 60% as a result of being able to detect its defensive playstyle. The strategy response to the Mech-based strategy of Sparks did not seem effective, however. Upon seeing the Mech-based strategy, ForceBot starts producing Mutalisks, as Vultures and Siege Tanks are unable to attack Mutalisks. However, Sparks responded in turn by producing large amounts of Goliaths, which are capable of effectively fighting against Mutalisks. In order to address this, an additional response to Goliaths was added to ForceBot.

8.1.2. Profiling

GOAL provides the option of profiling code. Enabling the option within the settings will make GOAL log the time it spends on each piece of code. This option has been frequently used over the course of ForceBot's development in order to ensure that all code is running efficiently and that no bottleneck forms within the bot as a result of poorly optimised code.

Through the use of profiling, it was shown that the manager code responsible for detecting certain 'events' has consistently been one of the slowest modules. This problem exists for both the general manager and the combat manager, although it is a bigger problem for the general manager. Between 8% to 10% of ForceBot's total CPU usage is used by the event detection of the general manager. This event module is responsible for detecting when a unit is created, morphed or killed. The reason for the high CPU usage is because the code needed to detect the death of a unit is as follows:

```
forall bel(friendly(Id, Type)), not(percept(friendly(Id, Type)) then {...}
```

The code must iterate over every `friendly` belief, and confirm whether it no longer perceives a percept for that friendly unit. Internally, GOAL separately queries the belief and percept base, meaning that this code will generate a query for every individual friendly unit, which frequently exceeds a hundred. The code to detect the creation of friendly units operates in a similar matter, making the detection of these events a costly process. Other bots are able to detect these events through native event handlers within BWAPI. However, as a result of the abstraction performed by GOAL and the connector, these events are inaccessible to GOAL users.

Part of this process can be simplified. GOAL agents have an initialisation module, which will run when the agent starts. Within this module, the agent is able to notify the relevant managers about its creation. Processing such messages is significantly faster than tracking the `friendly` percepts, as this only requires the manager to wait for individual messages.

The initialisation module can be used to efficiently track the creation of units, but a 'shutdown' module which executes when an agent is terminated did not exist. This problem is similar to the problem discussed in Section 5.3.3, where the problem of forwarding unit deaths to the manager was raised. Because agents may be terminated, the communication between agents and managers is unstable, and cannot be used to reliable inform the manager of events within the game. At the time, there was no solution to this problem that did not come with flaws. The proposed solution to this was a shutdown module that could be used to inform the manager of this event. However, the problem was instead resolved in a later version of the connector by allowing managers to receive percepts. This removed the need for communication between the manager and agents in order to detect the event.

Although this solution has worked successfuly, the computational demands of this solution are high, as

shown earlier. For this reason, following the completion of SSCAIT, the option to use a shutdown module was added to GOAL. Any agent can now have a shutdown module by adding it to the agent definition located within the mas2g file. With this new module, the example definition given in Section 4.1.1 can now be extended as follows:

```
define drone as agent {
   use DroneInit as init module.
   use DroneEvent as event module.
   use Drone as main module.
   use DroneShutdown as shutdown module.
}
```

Using the shutdown module, each agent automatically sends a message to the managers, informing them that the agent has shut down. This means that, just like detecting the creation of friendly units using messages within the initialisation module, the managers can now detect deaths by waiting for death messages sent by agents which are in the process of terminating. The shutdown module can be used in the same manner in order to detect other events such as unit morphing.

The new design of event detection uses an event-driven design, as the managers only need to wait for event notification messages instead of actively detecting events. This allowed the code to run significantly faster, reducing the total CPU usage of the event detection of the general manager to less than 1% of ForceBot's total CPU usage. The average cycle time of the general manager was reduced by 54.4% as a result.

8.1.3. Drone Defence

The last major change to ForceBot over the course of the final version involves the drone defence. ForceBot has had problems with this in the past, as there is no organisation between the Drones, meaning that each Drone will only defend itself, and not each other. In order to resolve this, a fourth manager was added, the Drone defense manager, nicknamed 'DDMind'.

The Drone defense manager will communicate with all Drones and call for the assistance of nearby Drones when one of them is attacked. It can also call all Drones into action if need be. For example, if it finds that the opponent is attempting to do a cannon rush or other attempts to construct enemy buildings within ForceBot's base, the Drone defense manager will order all Drones to attack those buildings before they can complete construction, as it is difficult to destroy them after the buildings have completed construction.

Because Drones are not useful for defending beyond the earliest portion of the game, because of their low health and damage, the Drone defense manager will shut down after 5 minutes of in-game time, given that it is not currently defending. This is because ForceBot will have trained Zerglings or built static defences to protect itself with instead by that point in time. As the CPU load is low during the first 5 minutes, the Drone defense manager makes quick cycles, making the Drone defence highly responsive. Finally, the addition of the Drone defense manager means that the Drones themselves no longer need to receive the `enemy` percept type, as they are instead informed of any enemies to attack by the Drone defense manager. This means that Drones now only receive the `status`, `order` and `gameframe` percept types, for a maximum of 3 percepts per cycle, where as previously Drones could receive over a hundred `enemy` percepts in a single cycle during the later stages of the game. As ForceBot will have up to a maximum of 75 Drones, this reduction in percepts results in a significant performance increase.

The changes to the defensive behaviour of the Drones will also help in cases where ForceBot does not yet have defences available when the enemy attacks, further addressing the issue of dealing with highly aggressive bots encountered in the SSCAIT Round Robin.

8.2. SAIL

The StarCraft Artificial Intelligence League (SAIL) is a new StarCraft ladder which began operating on March 20th, 2018. The main goal of SAIL is to to encourage competitiveness and the increase of maximum bot strength. SAIL uses a ladder structure similar to SSCAIT, with games being continuously played all year round and ranking them using Elo ratings. Unlike SSCAIT, the games will not be streamed, meaning the games need not be limited to speeds at which a human is able to watch them. Therefore, many more games play out daily at SAIL than at SSCAIT, allowing for more data to be collected to improve the bots.

8.2.1. SAIL Startup Problems

When ForceBot was added to SAIL a problem was revealed, as SAIL could not start ForceBot successfully. The problem was revealed to be the number of files which it uses. Most StarCraft bots are only composed of a single `DLL` or `JAR` file alongside a configuration file in some cases. GOAL bots come packaged in a `JAR` file with GOAL, Prolog, the connector, and any other necessities. Upon launch, these required files are unpackaged so that the bot can run. Previously, the `JAR` would be unpackaged before being uploaded to SSCAIT, as we had presumed this to improve startup time. However, this caused problems with SAIL, which would transfer the bot directory between games. This issue has therefore been resolved by not unpackaging the `JAR` file in advance to reduce the total number of files that needs to be transferred.

Although the issue has been resolved, ideally the StarCraft GOAL packaging pipeline may be improved in the future so that the bot can be contained within a single JAR file without requiring any unpackaging like purely Java-based bots.

8.2.2. SAIL Results

ForceBot participated in SAIL for roughly a month, playing a total of 403 games. After a month, the computer responsible for running the ladder suffered technical issues and SAIL has remained inactive since. The results of SAIL are a good source of information as to ForceBot's game performance, as all of the games played within it use its final version. The results can be seen in Table 8.2. These results do not include bots with less than 200 games played. ForceBot ended ranked 42nd, with a win-rate of 48.88%, placing it in the middle of all bots competing.

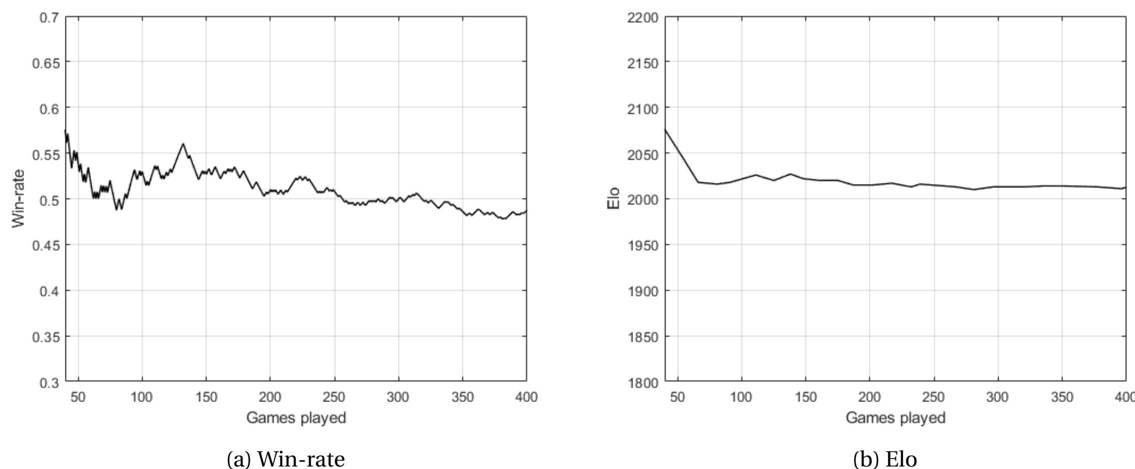
Name	# Games	Elo	Name	# Games	Elo
krasi0	495	2492	DawidLoranc	458	2005
Ironbot	485	2470	Stone	481	1997
MarianDevecka	463	2345	WillBot	521	1987
Bereaver	501	2307	MegaBot2017	442	1982
PurpleWave	456	2277	BananaBrain	469	1981
Steamhammer	466	2270	MiddleSchoolStrats	288	1981
tscmoop	468	2243	PeregrineBot	471	1970
NeoEdmundZerg	475	2237	FlorianRichoux	462	1953
CherryPi	482	2232	MadMixP	484	1950
Microwave	466	2231	GaoyuanChen	481	1950
Locutus	321	2226	PineappleCactus	490	1933
WuliBot	486	2219	UPStarCraftAI2016	451	1931
TyrProtoss	477	2214	AurelienLermant	472	1909
SimonPrins	446	2212	YuanhengZhu	488	1907
MartinRooijackers	455	2212	Xelnaga	456	1879
TomasVajda	459	2210	HannesBredberg	443	1876
Antiga	472	2203	SunggukCha	463	1875
tscmoo	489	2197	LukasMoravec	472	1872
Arrakhammer	467	2184	Zercgberht	471	1862
tscmoor	485	2180	RomanDanielis	489	1850
AndreyKurdiumov	464	2175	MatejIstenik	465	1847
AndrewSmith	479	2162	MadMixT	452	1846
BlackCrow	459	2155	Sling	509	1845
ChrisCoxe	476	2153	Sparks	474	1839
AILien	484	2140	MadMixZ	471	1839
Ecgberht	472	2134	NielsJustesen	315	1837
Flash	492	2131	TomasCere	485	1834
PurpleSwarm	451	2130	JakubTrancik	481	1826
Randomhammer	496	2130	NiteKatP	458	1810
NLPRbot	460	2125	KaonBot	460	1801
ZurZurZur	483	2109	NUSBot	500	1798
KillAlll	481	2093	Myscbot	473	1796
SijiaXu	479	2088	CruzBot	476	1783
DaveChurchill	474	2085	MarineHell	483	1728
CasiaBot	458	2067	MarekKadek	472	1728
tscmooz	446	2047	Alice	233	1718
ICELab	481	2041	DAIDOES	477	1717
NiteKatT	458	2041	HOLDZ	340	1687
CarstenNielsen	467	2026	JohanKayser	486	1670
SoerenKlett	455	2019	Korean	478	1668
PurpleSpirit	441	2014	BryanWeber	460	1636
ForceBot	**403**	**2014**	OpprimoBot	308	1599
Ziabot	478	2007	TravisShelton	536	1573

Table 8.2: Results of the SAIL ladder

The ranking of ForceBot is worse than its ranking in the SSCAIT Round Robin. There are a primarily two reasons for this. First, SSCAIT restricted bots participating in the Round Robin to one per author. Many authors responsible for some of the strongest bots have created multiple, meaning there is a larger number of strong bots participating in SAIL. In total, 12 bots participated in SAIL which ranked higher than ForceBot and did not participate in SSCAIT. Conversely, only 2 bots ranked higher than ForceBot in SSCAIT did not participate in SAIL. Secondly, the total number of games played in SAIL is higher. As ForceBot is not a learning bot, its win-rate gradually decreased as more games were played, giving an advantage to learning bots.

Therefore, the worse ranking is not indicative of a worse game performance. Instead, we look at the ranking of all of the bots present in both SSCAIT and SAIL, and analyse how they changed between SSCAIT and

SAIL. Among the 59 bots that were present in both SSCAIT and SAIL, ForceBot ranked 31st in SSCAIT and 30th in SAIL. Based on this it does not seem that the final version is significantly stronger than the version used in the SSCAIT Round Robin, even though the internal testing results would indicate otherwise. This may indicate that the internal testing was subject to a degree of over-fitting, reducing game performance against other bots. However, it seems more likely that this is a problem of sample size and learning bots gaining an advantage from a larger quantity of games being played. This can be seen in Figure 8.1a, which shows that the win-rate of ForceBot gradually decreased over time. In SSCAIT, 2 games were played against each opponent, while an average of 4.7 games were played against each opponent in SAIL.



(a) Win-rate (b) Elo

Figure 8.1: Win-rate and Elo of ForceBot over time in SAIL

The Elo of ForceBot on SAIL was consistent. In Figure 8.1b we can see that, following the first 60 games, the Elo rating of ForceBot remained mostly stationairy, hovering between 2010 and 2030 rating. With the average Elo on SAIL being 2000, the lowest rating being 1573 and the highest being 2492, this firmly places ForceBot slightly above average in the ranking.

Additionally, analyses were performed on ForceBot's game performance against different races and on different maps. These analyses used data collected by Jay Scott on map performance [34] and race matchups [35]. In the race matchup data, the game performance of each bot against each other race was measured. The results for ForceBot can be seen in Table 8.3. The win-rates against each race do not differ a great deal. The win-rate deviation of each race matchup compared to the average for that bot was calculated for each bot. ForceBot ranked as the 9th most consistent bot out of the 80 bots with more than 200 games present in SAIL. Based on this we can conclude that ForceBot is a highly consistent bot against each of the races.

ForceBot's poor performance against Terran, relative to the other matchups, is a notable result. The average Zerg bot in SAIL was found to have an advantage against Terran, with a win-rate of 55%. From this we can conclude that ForceBot's strategies against Terran has a lot of room for improvement.

Versus	Win-rate	Avg. deviation
Protoss	52%	3.12%
Terran	44%	4.88%
Zerg	50%	1.12%
Random	46%	2.88%
Average	48.88%	3.00%

Table 8.3: ForceBot's win-rate and average deviation against each race on SAIL

The win-rates of ForceBot on each map were also measured, the results for these are shown in Table 8.4. Again, the consistency of ForceBot was compared to that of other bots. This time, ForceBot ranked as the 45th most consistent bot out of 80 bots. This result is not significantly worse than average, however it can be concluded that ForceBot is not very consistent across all maps.

Map	Win-rate	Avg. deviation
Jade	58%	9.12%
Destination	57%	8.12%
Empire of the Sun	56%	7.12%
Benzene	55%	6.12%
La Manche	55%	6.12%
Roadrunner	55%	6.12%
Andromeda	52%	3.12%
Fighting Spirit	49%	0.12%
Circuit Breaker	45%	3.88%
Tau Cross	43%	5.88%
Python	42%	6.88%
Neo Moon Glaive	41%	7.88%
Heartbreak Ridge	40%	8.88%
Icarus	37%	11.88%
Average	48.88%	6.52%

Table 8.4: ForceBot's win-rates on each map on SAIL

The reason for this low consistency across maps is likely the result of how the combat simulator operates. In the case of Circuit Breaker, Python and Neo Moon Glaive, the center of the map is composed of a single, large region. The large center regions result in the combat simulator regarding units which are far distances apart as being in close proximity to one another, causing it to make poor predictions. The cause of this is that the combat simulator simulates battles for each region individually. For each region, a fight is simulated between all friendly and enemy units within that region and any regions adjacent to it. This is based on the assumption that units in adjacent regions are close to each other, however this is not always the case for these maps. Conversely, a map such as Jade splits the center of the map into multiple smaller regions, increasing the accuracy of the combat simulator.

The poor performance on Tau Cross, Heartbreak Ridge and Icarus is likely explained by the multiple routes that can be used to cross the center of the map. This means that ForceBot's and its opponents army can sometimes pass by each other without seeing one another. When the enemy army attacks is found, ForceBot often retreats through the opponents army, causing ForceBot to lose most of its army.

In both cases, the fault lies in the combat manager. In order to improve map consistensy, the combat manager will need to become more capable at handling differences in region sizes. It should also take into account the possibility of armies passing each other without seeing each other and avoid retreating through the opponents army.

8.2.3. Internal Testing Results

A fifth and sixth internal testing tournament were held in July 2018. The results of these tournaments can be seen in Table 8.5. The fifth tournament uses the same version of ForceBot as was used on SAIL. The sixth tournament uses an updated version with the bugs that were noted between March to July addressed, as well as addressing strategies and behaviour that were not performing well. One example of this is that ForceBot would prioritise Ultralisks and Guardians too much in the late game, leading to it not producing enough Hydralisks, causing an unbalanced army composition.

Opponent	Test #1 AIIDE	Test #2 Nov. 3	Test #3 SSCAIT	Test #4 Jan. 7	Test #5 SAIL	Test #6 Jul. 25
(P) Lukas Moravec	60%	40%	60%	80%	70%	100%
(T) KaonBot	50%	80%	80%	80%	90%	90%
(Z) Aurelien Lermant	100%	90%	80%	90%	90%	100%
(P) Gaoyuan Chen	20%	30%	50%	70%	60%	90%
(T) Sparks	40%	20%	60%	40%	70%	90%
(Z) zLyfe	40%	10%	70%	60%	90%	90%
(P) Antiga	10%	10%	30%	50%	80%	80%
(T) Simon Prins	0%	0%	20%	60%	80%	70%
(Z) NLPRBot	50%	30%	20%	40%	30%	50%
Average	41.1%	34.4%	52.2%	63.3%	73.3%	84.4%

Table 8.5: Win-rates against test bots over the course of all tests

The final performance is excellent, with ForceBot achieving a win-rate of 70% or greater over all bots except NLPRBot.

The most major bug to be addressed is related to the construction of Extractors in order to collect vespene gas, and occurred in 7 of the 24 losses by ForceBot in the fifth tournament. This bug ocurred if the construction of an Extractor, in addition to another building, were planned simultaneously when there were insufficient resources. This was intended behaviour from ForceBot's predictive building that was added to ForceBot V9. However, this could lead to the Drone attempting to start construction, but right before placement occurs, the amount of resources drops below the cost of constructing an Extractor due to the other building being placed. Normally, this will not cause a problem, as the building will simply be constructed once there are enough resources again. However, in some cases where this ocurred, the Drone in charge of constructing the Extractor did not attempt restart construction. When this happened, the Drone would idle for the rest of the game. Although the agent of the Drone was still active, as it continued writing to its log files, its actions were not performed. The exact cause and solution for this behaviour is still unknown – the bug was instead circumvented by ensuring that building construction does not plan the construction of Extractors ahead of time, so that the required amount of resources are always available.

8.3. Conclusion

The development of ForceBot over the course of this milestone primarily aimed at improving its ability to respond to the opponents strategy, as the results from SSCAIT showed that ForceBot lost a large number of its matches by selecting a build order and strategy at the start of the match that was ineffective against the enemy. As ForceBot is now more capable at responding to the opponents strategies, the choices made by ForceBot at the start of the match are now less of a deciding factor on the outcome of the match.

Another focus in this milestone was on addressing computational performance through profiling and using the new 'Shutdown module' feature. This feature has been added to GOAL in order to allow ForceBot to efficiently detect the deaths of friendly units. Prior to this, a query needed to be used which profiling showed to be highly inefficient. As a result of the shutdown module, the cost of detecting the deaths of friendly units was reduced from around 8 to 10% of ForceBot's total CPU usage, to less than 1%.

Over the course of this milestone, significant game performance improvements have been made to ForceBot. In internal test tournaments, ForceBot's win-rate increased from 52.2% during SSCAIT to 73.3% in the SAIL version. This increase in game performance was not clearly visible in the results of SAIL, however. The cause for this is likely the greater number of matches that were played, resulting in learning bots gaining an advantage. After SAIL, a number of further bugfixes and minor adjustments were made according to the results, leading to a final win-rate of 84.4% in internal test tournaments.

9

Development, Tools and Language

It is desirable for modern programming environments to provide features that assist in tasks such as writing code, refactoring, debugging and optimisation in order to produce a more efficient working environment. The aim of this chapter is to discuss the tools available to programmers utilising GOAL to develop multi-agent systems, particularly with regards to environments that feature a large number of agents, which GOAL has not been used in before. The effectiveness and ease of use of the tools provided will be analysed, to see whether they can be improved or expanded upon. Section 9.1 will discuss the capabilities of the IDE, particularly with regards to writing and modifying code, as well as refactoring. Section 9.2 will discuss the debugging capabilities, while Section 9.3 focuses on profiling and optimisation tools. Finally, Section 9.4 discusses GOAL as a programming language.

9.1. IDE

The GOAL Integrated Development Environment (IDE) functions as a plug-in to an existing IDE: Eclipse. Eclipse is an IDE primarily used for programming in Java, although support exists for many more languages. In particular, it has an extendible plug-in system that allows for plug-ins to be created that allow Eclipse to support any desired language. GOAL has made use of this feature through the creation of a plug-in that allows Eclipse to support GOAL. By utilising an existing IDE, many features are available for any programmer working with GOAL, such as syntax highlighting, on-the-fly building and error reporting. As a result, GOAL has few shortcomings in this department. Three issues of note were found: variable highlighting, refactoring and automatic building.

The GOAL plug-in has no support for variable highlighting, preventing one from seeing all uses of a variable in a section of code at a glance. This is typically not a major issue, most variables are only bound for a single statement. However, when nested statements are used, variables may be bound for several statements. In this case, it becomes difficult to keep track of variables.

The second issue found is the lack of refactoring support. Many modern IDE's can help in speeding up refactoring. When a variable, class or file is renamed or moved, every other reference to that object in all files is adjusted appropriately. In GOAL, this primarily concerns importing other modules. For example, importing a module within the same folder is done as follows:

```
use ModuleA as module.
```

However, if this module is moved to the parent directory, the line needs to be adjusted as follows:

```
use "../ModuleA.mod2g" as module.
```

Updating imports in numerous files is a tedious process, and the GOAL plug-in does not support automated refactoring. This is a particular problem in ForceBot, as it is composed of a much larger number of files than previous GOAL projects. In ForceBot there are a total of 722 imports. The most imported file, `OverMind.pl`, is imported in a total of 78 files. With such a large number of imports, a need for automation is created.

The third issue is related to the automatic building of the Eclipse plug-in. The plug-in performs an automatic build whenever a file is saved, allowing it to quickly report compilation errors and warnings. There

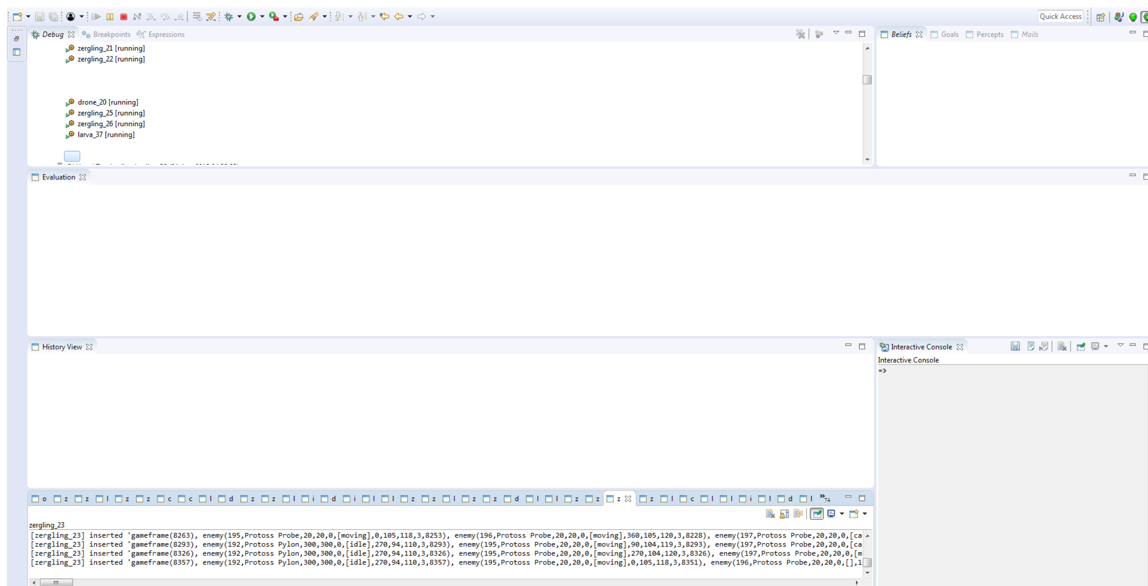


Figure 9.1: An example run of Debug Mode for ForceBot

is some room for improvement in this process: when a large number of files are saved simultaneously, the project is rebuilt once for every file saved. As Eclipse will block the use of the IDE when multiple builds jobs are in the queue, this will result in wasted time. In the case of ForceBot, each build requires 5-6 seconds, meaning the IDE will be unusable for roughly a minute if more than 10 files are saved at once. This is not a major issue, as it can be circumvented by temporarily disabling the automatic build or always saving immediately, but the problem should be easy to fix.

9.2. Debugging

The GOAL plug-in for Eclipse provides support for several types of debugging: debug mode, logging and profiling. In this section each of these three types will be analysed to determine their advantages and disadvantages. Suggestions for improvements will be made where possible, based on the usefulness of the debug type during ForceBot's development.

9.2.1. Debug Mode

A dedicated debugging mode is available when running agents [25]. When used, a large amount of real-time information and options are made available to the user, seen in Figure 9.1. The user can view and modify the running state of agents, and can use a stepping mode which allows for a detailed inspection of the agent state at a specific point in execution. When an agent is paused, its goals, beliefs, percepts and mails can be viewed. Breakpoints can be set at specific points in the code, allowing agents to be paused automatically upon reaching a certain line of code, or when a specific condition holds. An interactive console can be used to modify or interact with the knowledge and beliefs of an agent in real-time. Watch Expressions can be used in order to continuously evaluate specific expressions for all agents. Finally, every agent receives its own console which will log information according to the logging options specified within Eclipse's settings.

Using debug mode, the execution of agents can be precisely controlled, allowing developers to identify problematic code and observe behaviour surrounding lines of code. However, within the context of StarCraft, debug mode was found to be less useful. The large number of agents makes it difficult to extract useful information from debug mode. The list of agents becomes uninformative, as there can be over a dozen agents of the same type, each with the same name. It is impossible to tell which agent in the list corresponds to which unit within StarCraft. The individual agent consoles also work against the user when the number of agents is high: a tabbed list of agent consoles is generated which is difficult to navigate. For example, in the bottom of Figure 9.1, the agent consoles have shrunk to the extent that only the first letter of each agent is visible on each tab. Finally, although StarCraft can be paused, it does not do so automatically when agents are paused, which makes pausing agents more bothersome than it needs to be.

The issues present in debug mode are the result of the large number of agents present in StarCraft. The

design does not account for the possibility of potentially hundreds of agents being active at once. Although this issue may be resolved by creating a new design which allows for better navigation of a large number of concurrent agents, it can be questioned whether this will create an adequate solution. The debug mode provides a large amount of information for each agent, however in ForceBot's testing only a single agent was often times of interest. For example, when debugging Drone construction behaviour, only the Drone receiving the task is of interest. Another example is when you see a unit behaving strangely, only that unit is of interest.

Another issue associated with debug mode is that it is resource intensive. As debug mode is the only debugging approach which can be interacted with, and provides output in real-time, it is therefore also the most costly. Because of this, it may be necessary to look more towards other means of debugging when dealing with environments with large numbers of concurrent agents such as StarCraft. One of these other means is via logging to files.

Taking these problems into account, as well as how often times only a single agent is of interest, I will suggest a different approach from the existing debug mode. I believe that a debug tool which displays additional information depending on the currently selected unit(s) within StarCraft is a good approach. This could be done by either printing the information within StarCraft itself, or by having the Eclipse console automatically switch to display the logging of the currently selected unit(s). This approach allows developers to use the graphical environment of StarCraft to indicate to GOAL which agent they are interested in at that moment.

9.2.2. Logging

The GOAL plug-in provides the option to log desired information to files, allowing for post-match analysis of agents. This logging can be easily customised, allowing developers to change how, where and what is logged. Each agent will write its own log file. Logging to file is a feature that has been used extensively over the course of ForceBot's development, as it was the best source of insight for determining exactly what was going on inside the agent. Furthermore, this debug option does not use up many resources, making it appealing to keep the logging to files option enabled even when not aware of any bugs. If a bug is encountered during a match, the log can be used to try and determine the cause of the bug.

The log files themselves are formatted as an XML file, with several elements such as date and message (event). An example entry in the log file is shown below.

```
<record>
   <date>2017-12-08T16:00:51</date>
   <millis>1512745251609</millis>
   <sequence>81467</sequence>
   <logger>hydralisk_14_17-12-08_15.56.40.txt</logger>
   <level>WARNING</level>
   <class>goal.tools.logging.GOALLogger</class>
   <method>log</method>
   <thread>104</thread>
   <message>'region(5,32,12)' has been inserted into the belief base 'this'.
</message>
</record>
```

These log entries have proven themselves useful, however there is room for improvement. One of the primary issues encountered is bloat in the entries. Within the above example, only the **`message`** and **`millis`** element were typically of use, as they tell what the agent is doing and when. The only other useful element is the **`date`** element is occasionally useful when attempting to find roughly the same point in time in multiple log files, but otherwise merely a less precise **`millis`** element. The other element suffer from the following issues:

1. The **`sequence`** element can be used to determine the number of sequences that have taken place since the last logged entry. Although it is possible this can have some practical applications, such a situation never occurred during ForceBot's development.

2. The **`logger`** element more or less states the current file name. This can be used to distinguish agents when all agents log to a single file, but is otherwise useless.

3. The **`level`**, **`class`** and **`method`** element are always the values indicated in the above example and serve no purpose.

4. The **thread** element frequently changes, but does not seem to serve a purpose.

5. Although minor, the closing tag of the **message** element could be placed behind the message in order to save space.

Taking all of the above comments into account, ideally options can be added to determine which elements are logged. If such options were added, one could create the following entry format which, for the development of ForceBot, would be functionally near-identical, while demanding much less space:

```
<record>
   <millis>1512745251609</millis>
   <message>'region(5,32,12)' has been inserted into the belief base
'this'.</message>
</record>
```

A Comma-Separated Value (CSV) format option could also be given, in order to further reduce entries to a single line. These formats would make the log files more compact and readable, allowing developers to analyse the log files more quickly.

Finally, one more debugging option exists, although a little more difficult to find. By using the 'Run Configurations' native to any Eclipse execution, the console output can be logged to file. The majority of the messages found in the console are less-detailed versions of the entries found in the previous log files, for example:

```
[hydralisk_14] 'region(5,32,12)' has been inserted into the belief base 'this'.
```

Although this log file has the benefit of being much more compact than the previous, this default console contains the entries of all agents, making it difficult to track events from a single agent. Instead, this console is primarily useful because it logs the environment as well as the agents. For example, if an agent crashes, it is only logged to the default console, not to the log file of that specific agent. As agent logging is the vast majority of all logging within the default console, it can be difficult to find the desired log entry. With that in mind, a potential improvement to this log would be to remove the agent logging, or allocate a separate log file for the environment through the GOAL options, rather than the Eclipse options. Finally, the example below shows an error that occurred when an agent attempted to divide by zero. With the environment information from this log file, all information needed to debug successfully is available to developers.

```
WARNING: agent 'overmind' was forcefully terminated.  GOALActionFailedException
failed to evaluate condition of 'if bel(X is 1 / 0) then sleep(X)': failed
to query 'X is 1 / 0' on 'overmind:overmind:belief base'.:  swi prolog says
the query failed because computation failed:zero_divisor:  PrologException:
error(evaluation_error(zero_divisor), context('/'('/', 2), _202))
```

9.3. Profiling & Optimisation

Optimising code is an important task when developing a bot, particularly in a real-time game such as StarCraft. As computational power is limited, any optimisations to improve the speed of the program can play a big role in the overall effectiveness of the bot. To that end, GOAL provides profiling options, allowing developers to determine how much time agents spend on a module, condition or action. An example of the output is given below in Figure 9.1. Each profiling log is a tab-separated text file, in which the 'source' is the module, line or action being profiled. The output tells how many times each module, condition or action is executed and the amount of time spent on execution.

source	number of calls	total time	source info	this	parent
Module:DroneEvent	277	0.312002	line 7, position 7 in *<Path>*	*<Path>*	–
Module:DefendEvents	277	0.2496016	line 5, position 7 in *<Path>*	*<Path>*	*<Path>*
Module:UpdateStatus	19	0.0312002	line 4, position 7 in *<Path>*	*<Path>*	*<Path>*

Table 9.1: Example profiler output

The profiling output does perform its intended role, and has been used over the course of ForceBot's development to improve performance and identify bottlenecks within the code. However, I believe that there is room for improvement to the output format of the profiler. Currently, the output is written unordered to a `txt` file. In order to make sense of the output, it needs to be moved to spreadsheet software such as Microsoft Excel, which will place the tab-separated text lines into separate cells and allow for sorting on data such as 'number of calls' or 'total time'. To speed up this process, the file can be saved as a `csv` file with a delimiter specified at the start of the file. Additionally, the columns should be re-organised. The 'number of calls' and 'total time' columns should be at the front, as their fields are always the same length. On the other hand, 'source' and 'source info' columns have large individual differences in field length. Furthermore, the data provided by the 'this' and 'parent' columns are paths in the form of 'goal.tools.profiler.ProfileStatistic@1cb3aa3'. Similar to the debug logging, this data is not important and can be left out of the output. Combining all of these changes results in the output presented in Figure 9.2. By performing these changes there will be less need for the user to process the data before it can be analysed.

	A	B	C	D	E	F	G	H	I	J	K
1	source	number o	total time	source inf	this	parent					
2	Module:A	6	0.0312	line 11, pc	goal.tools	goal.tools.profiler.ProfileStatistic@5a6434					
3	Module:A	3	0	line 9, pos	goal.tools	goal.tools.profiler.ProfileStatistic@1739748					
4	Module:A	109	0.187201	line 16, pc	goal.tools	goal.tools.profiler.ProfileStatistic@33cebe					
5	Module:A	25	0.0468	line 7, pos	goal.tools	goal.tools.profiler.ProfileStatistic@1d2fe71					

	A	B	C	D	E	F	G	H	I	J	K
1	total time	number o	source inf	source							
2	0.187201	109	line 16, pc	Module:AssignTaskToDrone(Id, Agent, NewTask, NewTarget, BuildTarget)							
3	0.0468	25	line 7, pos	Module:AssignTaskToOverlord(Id, Agent, Task, X1, Y1, RegionId)							
4	0.0312	6	line 11, pc	Module:AssignBuildToDrone(Thing, RegionId)							
5	0	3	line 9, pos	Module:AssignExpandToDrone							

Figure 9.2: Suggested new layout for profiler output

Further improvements to the way in which the data is presented could be done by separating the data in advance. Currently, there are numerous sources, such as 'Module', 'IfThenRule', 'ForallDoRule' and more. If many or all sources are enabled then much of the information is listed twice, for example when the data is sorted by 'total time', the 'IfThenRule' with the highest 'total time' is likely positioned just below the module which it is located in. I believe that separating the module and rule information into separate sheets could make the data easier to process, as each spreadsheet will contain less repeated information.

Finally, a further improvement could be made by providing options to merge the results of identical agent types. Currently the profiler will create a separate profile results file for each agent. In StarCraft this results in a large number of files for each agent type. Depending on the activities performed by an agent, its environment, and the length of time it remained alive, the individual profile results can vary greatly. By merging the results of agents of the same type, the profile results will become more accurate.

9.4. GOAL Language

One of the goals of this thesis is to analyse the performance of GOAL in an environment with complexity on the level of StarCraft. ForceBot is the largest GOAL bot created to date, in an environment with significantly more agents than GOAL has been used for in the past. This section discusses difficulties encountered with GOAL as a language and makes suggestions on how these areas may be improved in the future. In addition, this section will cover features of the language which were not used throughout ForceBot's development.

9.4.1. Nesting of `not`

In Section 6.5.1 it is mentioned that the enemy targeting was redesigned. Rather than storing knowledge of enemies, the approach was changed to find the closest enemy directly from percepts. Whenever an enemy percept which is closer, or of higher priority, is found, it will store the new targets distance and priority. After iterating over all enemy percepts, the stored target is guaranteed to be the closest enemy with the highest priority.

However, the step of storing the distance and priority is one that should not be necessary. After all, the approach used before, of storing all enemy knowledge, skipped the step of storing the target separately. The reason that the new approach does require this step is due to a limitation of the GOAL language. Below is pseudo-code, with simplified beliefs, for making a unit target enemy ground units, used by the approach utilised before Section 6.5.1.

```
if bel(status(SelfX, SelfY)),
   bel(enemy(Type1, EnemyX1, EnemyY1), isGroundUnit(Type1),
   distance(SelfX, SelfY, EnemyX1, EnemyY1, Distance1)),
   not(bel(enemy(Type2, EnemyX2, EnemyY2), isGroundUnit(Type2),
   distance(SelfX, SelfY, EnemyX2, EnemyY2, Distance2),
   Distance2 < Distance1)) then
      attack(EnemyX1, EnemyY1).
```

This pseudo-code first determines its own position. Once it has done so, it selects an enemy and sees if it fulfils the targeting requirement of `isGroundUnit`. If it does, it determines the distance to this enemy. Afterwards it iterates over all other enemies and ensures that no enemy which fulfils the targeting requirement is closer than the already selected enemy. Once it has confirmed this to be true, the unit is ordered to attack and move towards the location of the selected enemy. This code ensures the closest valid target is selected to attack and does not require the inserting or deleting of any distance/priority beliefs.

In Section 6.5.1 this code was altered. The enemy percepts would no longer be stored as beliefs. This results in **`bel`**`(enemy)` being replaced with **`percept`**`(enemy)`. If we do so, then the pseudo-code would have to be changed to the following:

```
if bel(status(SelfX, SelfY)), percept(enemy(Type1, EnemyX1, EnemyY1)),
   bel(isGroundUnit(Type1), distance(SelfX, SelfY, EnemyX1, EnemyY1, Distance1)),
   not(percept(enemy(Type2, EnemyX2, EnemyY2)),
   bel(isGroundUnit(Type2), distance(SelfX, SelfY, EnemyX2, EnemyY2, Distance2),
   Distance2 < Distance1)) then
      attack(EnemyX1, EnemyY1).
```

However, this pseudo-code does not function in GOAL. A **`not`** statement can only nest a single **`percept`** or **`bel`** statement, not multiple. Instead of that, the code was changed to store the distance and priority in a separate belief which is kept updated, as we need this belief to act as a 'variable' to ensure that we are targeting the closest enemy. However, inserting and deleting beliefs is an operation that is preferably avoided in order to maximise performance. I believe a future version of GOAL should aim to have the functionality required for this pseudo-code to function, as it should allow for an increase in performance. Furthermore, it is also the type of code that one would expect to work, but currently it does not.

9.4.2. Unused Features

Over the course of ForceBot's development, the majority of GOAL's language features saw some degree of use. In this section, the lesser used or unused features of GOAL's language are discussed.

Goals

GOAL separates the mental state of an agent into knowledge, beliefs and goals. The use of knowledge is clear, for example storing a rule to calculate the distance between two points. However, the difference between beliefs and goals is less clear, particularly in an environment such as StarCraft. For example, if the combat manager gives an order to a Zergling to destroy an enemy base, ForceBot will adopt a `task('attack', (<X>, <Y>, <RegionId>))` goal. The Zergling will inform the combat manager with a `cleared(<X>, <Y>, <RegionId>)` message, indicating the enemy base that was destroyed. However, there is no meaningful distinction between this task being adopted as a goal, or inserted as a belief. You can either say that the Zerglings goal is to attack, or that it believes its order to be attack. The distinction is fairly arbitrary.

One of the main uses of goals is the use of the additional **`goal-a`** and **`a-goal`** statements, which function as a combination of **`goal`** and **`bel`** statements, shown in Equation 9.1 and 9.2.

$$\mathbf{a}-\mathbf{goal}(qry) = \mathbf{goal}(qry), \mathbf{not}(\mathbf{bel}(qry)) \tag{9.1}$$

$$\mathbf{goal-a}(qry) = \mathbf{goal}(qry), \mathbf{bel}(qry) \tag{9.2}$$

However, in the StarCraft environment, these statements are rarely useful. For example, you cannot make adopt a goal for there to be "no enemies in <RegionId>". Another example is when you give a worker a task to construct a building, you cannot adopt a goal such as `friendly(<ID>, <Type>)`, as the <ID> will only be known after the building is placed. Finally, in order to improve performance, many percepts are not stored in the belief base by agents. As such, while the **`goal`** statement was used, primarily to indicate tasks, the **`goal-a`** and **`a-goal`** statements were never used.

Focus

The focus option can be used to set the 'focus of attention' of a module on a particular goal. This feature is designed to be used when there are conflicts in the goals of an agent. The reason why this functionality was never used is simply because goals themselves were not frequently used. Goals were primarily used to store the current task, however upon being given a new task, the old task would be removed. As such, no conflicts between goals can occur in ForceBot.

Exit

In modules it is possible to add an exit condition to the module, namely: *always*, *never*, *nogoals*, *noactions*. By default, all modules use the *always* option, while the main module will use the *never* option. In the case of ForceBot, this feature was never used, as there was simply no need to change from these values.

Order

The order setting of a module allows for developers to specify the order in which the rules of the module are traversed. The following options are available: *linear*, *linearall*, *linearrandom*, *linearallrandom*, *random* and *randomall*. In the case of ForceBot, only the *linear* and *linearall* options were used. These were primarily used in order to optimise the code in cases where only only the first applicable rule of the module needs to be performed. One of the examples of this is the code which decides whether to build a Hatchery as an expansion or in the base. This module contains 13 different rules to determine whether it should expand or not, of which only the first applicable rule needs to be performed. These rules are listed according to priority, and as such using a random order is undesirable. In the case of ForceBot, I found that there was always a priority to things, and as such a random order was never desirable.

Macros

Macros can be used to define a combination of queries, an example of this can be seen in the pseudo-code below. The defined macro determines whether the unit has a goal to morph into something, and whether it has the resources required to do so. In the module, this macro is used to trigger a `morph` action.

```
define shouldMorph(Type) as goal(morph(Type)),
   bel(costs(Type, MineralCost, GasCost, SupplyCost),
   resources(Minerals, Gas, CurrentSupply, MaximumSupply),
   Minerals >= MineralCost, Gas >= GasCost,
   MaximumSupply - CurrentSupply >= SupplyCost)

module Larvae {
   if shouldMorph(Type) then morph(Type).
}
```

However, in order for a macro to be useful, it must be used multiple times within a single module. Furthermore, they can often be replaced by knowledge rules. For example, the below pseudo-code would work can be used by moving the **`bel`** portion of the macro into the knowledge file of an agent and naming the knowledge rule `hasResourcesFor`. The benefit to this approach is that this knowledge rule is then re-usable across all modules which use that module file.

```
if goal(morph(Type), bel(hasResourcesFor(Type)) then morph(Type).
```

Consequently, macros are primarily useful when there are multiple rules within a single module that perform the same queries on bases other than the belief base. This situation did not occur for ForceBot.

Listall

`listall` is one of the three rules that can be used in GOAL, in addition to **`if`** and **`forall`**. This rule can be used in order to create a list of all results of a query, in a way that is similar to **`forall`**. However, while **`forall`** handles each result separately, the **`forall`** query allows for all results to be compared by placing them in a list. There are a number of times where this can be useful, such as when finding a result in a list with the highest value. However, finding the highest value in such a list can also be done using a **`forall`** statement that compares the result. An example of this is the targeting priorities for combat units. The approach that ForceBot uses currently has been explained in Section 9.4.1.

Another solution to finding the closest enemy can be made by using the **`listall`** rule. By placing all results in a list and iterating over the list to return the closest enemy. Using this approach, no inserts or deletes are required in order to determine the closest enemy. However, upon performing profiling while using this approach, the computational performance was found to be worse. As a result, the **`listall`** rule has not been used in ForceBot.

Although there are a number of features that were not frequently used in the development of ForceBot, I do not believe that there is a problem. A number of these features do address missing functionality, for example the **`listall`** rule can be used to retrieve the sum of a list of values efficiently. Using **`forall`** this would be inefficient, as unlike the targeting priorities this would require an equal number of inserts/deletes as there are results. Furthermore, the built-in Prolog predicate `findall` can be used, but only when finding single beliefs, not percepts, goals or queries involving multiple beliefs. Based on this, I believe that the majority of the unused features simply require conditions that are not frequently encountered in the StarCraft environment.

9.5. Conclusion

In this section the various tools offered by GOAL were analysed. Among these tools, debug mode was shown to not be useful as a result of providing information in a way that is difficult to parse when dealing with large numbers of agents, in addition to slowing down as a result of the number of agents. In order to address this, a suggestion was made to have debug mode only display the currently selected unit within StarCraft, in order to indicate to GOAL which agent the developer is interested in.

Both the logging and profiling tool have been useful assets during development, and no major difficulties were encountered in using them for developing a GOAL-based StarCraft bot. However, suggestions have been made to improve the ease of use by adjusting the output format of these files to one that can be parsed more easily and does not contain data entries that are not relevant for debugging.

Finally, the language features of GOAL were analysed. The only problem that was noted was that the syntax of the language does not allow for multiple **`bel`**, **`goal`** and **`percept`** statements to be included within a single **`not`**. However, this does not limit the functionality of the language, as the problem can be solved using different approaches, although these approaches are less than ideal.

10
Lessons Learned

In this chapter the results of the previous sections will be analysed in order to provide answers to the research questions posed in Section 1.3. The research questions that were posed correspond to four aspects of the development of a GOAL-based StarCraft bot: **language**, **connector**, **tools** and **computational performance**. These questions will be answered in Section 10.1, 10.2, 10.3 and 10.4 respectively.

10.1. Language

During the development of ForceBot, no major issues arose with the language aspects of GOAL. One issue was found in Section 9.4.1, although it is not major as there is a workaround for the issue, albeit less than ideal.

However, there are problems with the computational performance of GOAL. One particular case that was noted is the computational performance cost of detecting events such as a unit being created, dying, morphing, and more. As there are no events in GOAL, these must instead be detected by tracking changes to percepts. This problem was addressed for the `friendly` percept type in Section 8.1.2 through the addition of the shutdown module. However, I believe that this problem demonstrates a missing feature in GOAL's design: there are no events. Although the shutdown module addressed the problem for the `friendly` percept type, the problem persists for the `enemy` percept type which ForceBot's combat manager tracks. For bots utilising BWAPI itself, there are event handlers available for events such as a unit dying, however these events are not available in GOAL.

Considering the benefits gained from utilising an event-driven paradigm for the `friendly` percept type in Section 8.1.2, I believe that GOAL should aim to include explicit support for events. One approach to solve this problem would be to add percept types for most, if not all, of the events available in BWAPI. This approach would only solve the problem for StarCraft, however. Another approach is for GOAL to provide a way to 'hook' a percept, so that changes to it will trigger a notification of some kind. This could be used for any percept type that includes a 'key' parameter, such as the <UnitId> parameter of the `friendly` or `enemy` percept types. An example of this can be seen in the pseudo-code below.

```
if true then hook(enemy, [insert, modify, delete]).
if percept(hook(enemy(UnitId)) then ...
```

In the first line, the `enemy` percept type is hooked for event triggering, with the desired event types supplied with the list. The three options for event types are:

- **insert** – A `enemy` percept with a new <UnitId> is added to the percepts.
- **modify** – A `enemy` percept with the same <UnitId> changed in the new cycle.
- **delete** – A `enemy` percept is removed from the percepts.

Whenever one of these events trigger, a `hook` percept is received containing the changed percept and its key, in this case the <UnitId>. An optional third parameter can be added which allows for the event to only

trigger for a specific key value. For example, the blow psuedo-code will only trigger if an `enemy` percept with the given <UnitId> value is removed from the percepts.

```
if bel(enemy(UnitId, ...)  then hook(enemy, [delete], UnitId).
```

Using this approach, notifications can be sent out for events such as a new key appearing, parameters changing, or an existing key disappearing. Although this solution is more complex, it can be used in any environment.

At the start of this thesis, the following research question was raised: *What are the advantages and disadvantages of using GOAL for an environment such as StarCraft?* In order to answer this question, we must compare GOAL to languages traditionally used to program StarCraft bots: C++ and Java. C++ is used primarily for its speed, as well as BWAPI itself being programmed in C++. Although Java is slower than C++, this is not a factor that greatly limits the performance of the bot, evidenced by PurpleWave ranking 2nd in AIIDE. While C++ and Java are object-orientated languages, GOAL is a language designed for use in cognitive multi-agent systems, which StarCraft can be represented as by mapping agents to units and buildings within StarCraft.

In my opinion, GOAL possesses advantages over C++ and Java in its information density and abstraction. GOAL handles a large number of tasks for developers, such as starting and terminating agents and processing BWAPI data into compact percepts. Furthermore, the increased information density of GOAL code allows for small amounts of code to perform a great deal of work. As a result of these factors, a competent GOAL bot can be created in a short amount of time. However, GOAL currently suffers from problems with computational performance and synchronisation with StarCraft, which will be discussed in Section 10.4. This results in ForceBot possessing poor micro skills relative to other bots. At present, these problems are the primary bottleneck in improving ForceBot's game performance. Until these problems are resolved, GOAL bots are unlikely to be able to match the game performance of traditional bots.

10.2. Connector

The StarCraft-GOAL Connector is one of the main components of creating a GOAL bot for StarCraft. It is responsible for establishing a link between GOAL and BWAPI by converting the in-game information retrieved by BWAPI into percepts for the agents to process in GOAL, and converting the actions from agents in GOAL to in-game commands through BWAPI. This section aims to analyse the state of the connector, asks if and how it limits the potential of a GOAL bot, and discusses the lessons that can be learned from its development.

10.2.1. MAS Project 2018

The MAS Project was held again in 2018, from April to June, with a total of 49 teams of Bachelor students participating. Since the 2017 MAS Project the connector has seen a great number of changes, making the 2018 MAS Project a good opportunity to determine the effectiveness of these changes and the maturity of the connector. For this reason the changes that the connector went through over the course of this project will be outlined.

First is a small number of general changes that were made. Both StarCraft and ChaosLauncher, a tool used to utilise BWAPI, are now automatically started and closed along with the connector so that this task does not need to be done by the user. Furthermore, in order to assist with testing, the connector was changed to provide a number of additional game cheats that can be toggled easily using the connector, namely:

- Instant construction of buildings
- Removal of technology restrictions
- Removal of supply limit

Additionally, in order to simplify the startup sequence of GOAL bots, manager entities are now started one second before any other agent is started. As the connector pauses StarCraft until four actions have been performed by agents, this only results in StarCraft being paused for one second longer before the match is started. This is also a change made to simplify working with GOAL, as in the case of ForceBot a number of fail-safes are in place to address cases where a manager is started after an agent. By placing a delay between the launch of managers and agents, this problem is removed.

Lastly, a small number of percepts were changed as well, which will be outlined over the following sections.

10.2.2. Global Dynamic Percepts

enemy/11

The enemy percept type has been altered to also provide information regarding neutral units. Neutral units most frequently come in the form of buildings which obstruct part of the map. In some cases, neutral buildings are positioned in such a way that destroying them will open a new path. Prior to this change, GOAL agents could not perceive neutral units and therefore not clear these blockades.

nuke(<X>, <Y>, <RegionId>)

The <RegionId> parameter was added to this percept type. This change was made for consistency, as providing the <RegionId> whenever an `(X,Y)` location is given is standard.

researched(<CompletedList>)

This is a new percept type that has been added to simplify researching, as well as provide a guaranteed way of determining the completion of researches. <CompletedList> is a list of all researches that have been completed over the course of the match. Before the addition of this percept type, detecting the completion of a research was done by detecting when a researching agent stopped receiving the `researching/1` percept. However, in the event that the researching agent attempts to cancel its currently active research while it is near completion, you cannot be certain whether the research was completed or cancelled. This new percept solves this problem and provides information regarding the completion of researches directly.

underConstruction(<Id>, <BuilderId>, <Vitality> , <X>, <Y>, <RegionId>)

The <BuilderId> parameter was added, which gives the worker responsible for constructing the building. This is primarily important for Terran, as it tells which SCV is constructing a building.

10.2.3. Generic Unit Percepts

order(<Primary>, <TargetUnit>, <TargetX>, <TargetY>, <TargetRegionId>, <Secondary>)

The <TargetRegionId> parameter was added to this percept type. This change was made to address an inconsistency, as all other percept types which provide an `(X,Y)` location also provide the <RegionId>.

10.2.4. Unit-Specific Percepts

researching(<Type>)

The <Type> parameter of this percept type has been changed. This parameter now also provides the level of the research being performed, if applicable.

10.2.5. Connector Overview

The changes made to the connector over the course of the 2018 MAS Project are primarily aimed at ease of use and consistency. Although ForceBot has been updated to operate with these changes, no changes were made to ForceBot in terms of behaviour. Out of all changes, the only change that addresses missing functionality concerns the change to the `enemy` percept. However, on all maps used within SSCAIT and AIIDE, any region which has a neutral-blocked path leading into it also has an unblocked path leading to that same region. The neutral-blocked path is typically shorter, but requires the destruction of the neutral building before it can be used. In many cases, it is preferable to have a worker unit walk the long way, instead of opening a path by destroying a neutral blockade using combat units. Therefore, although the changes to the `enemy` percept did serve to address a missing functionality, no important functionality was lacking.

Major problems which were encountered with the connector over the course of ForceBot's development, have all been resolved over the course of the thesis. The main lessons learned from this were the changes to the connector with regards to manager handling and percept delivery. At the start of this thesis, the connector did not support manager entities receiving percepts. Over the course of development this proved to be a major obstacle, as the solution of having agents forward percepts to the manager was prone to errors if the agent responsible for that task is killed. Therefore, managers became better supported, with dedicated manager entities being added to address the problem. The important lesson to take from this is that there should always be a 'stable' entity present, as there exist many tasks which require a single, persistent agent in

order to properly handle them. Agents are not capable of performing such a task in an environment such as StarCraft, as units may die at any time.

Secondly is the change to percept delivery. Prior to the start of the thesis, the percept types delivered to each agent were pre-determined based on their unit type. However, as the percept types that an agent is interested in changes depending on the implementation, there was no choice but to have agents receive any percept type which could be of use to them. As a result, agents received many percept types which they did not use. This problem was solved by allowing developers to specify the percept types that each agent type is interested in. The lesson to learn from this is that in a game such as StarCraft, the relevance of percept types to specific units is implementation-specific and the decision to receive them or not is one that should be left to the developer, rather than the connector.

By addressing these major issues in the functionality, the connector became easier to work with, as the connector now performed tasks such as sending percepts to managers. The connector became more flexible and efficient by allowing developers to specify which percepts should be received by which unit types.

The biggest changes that have occurred to the connector over the course of this thesis are all related to the flow of information: changes to manager handling and percept subscription. As the usage of information differs based on the implementation of agents, the best choice is to allow the developers to control the flow of information.

At the start of this thesis, the following research question was raised: *What does the abstraction of the StarCraft-GOAL connector mean for GOAL bots?* In order to answer this question, we have looked at the changes that have occurred to the connector over the course of the 2018 MAS Project. The connector was used by a total of 49 groups of Bachelor students during this project. Over the course of the 2018 MAS Project, no major changes were found to be needed. Additionally, all major obstacles that have appeared during the development of ForceBot that were related to the connector have been addressed over the course of this thesis. Based on this, I believe that connector has significantly matured and can be used to support any strategy in StarCraft. However, the level of abstraction may need to be further adjusted in order to improve the computational performance problems that GOAL-based StarCraft bots suffer. This is discussed in detail in Section 10.4.5.

10.3. Tools

GOAL provides several tools to be used when programming with GOAL. These come in the form of an IDE using a plug-in for Eclipse. This plug-in provides many of the basic features of modern IDE's such as syntax highlighting, code completion, automatic building and error reporting. The IDE also provides additional options for debugging, logging and profiling. In this section, the usage of these tools over the course of ForceBot's development is noted and areas for improvement are explored.

IDE

The IDE itself is well-functional and I did not experience many difficulties while using it. The only point for improvement noted in Section 9.1 is that the IDE does not fully support the refactoring of files and modules.

Debug Mode

The debug mode offerred by the IDE could not be effectively used. The information provided by debug mode is primarily real-time, in-depth information about all active agents. In StarCraft, the number of agents is very high, and it isn't clear which agent corresponds to which unit. Additionally, the amount of agents also results in the amount of information being provided by debug mode is overwhelming. As a result, the user is unable to make effective use of the information. The quantity of information, combined with the number of agents, also leads to the IDE slowing down, which includes the bot. Consequently, it is a hindering factor in the 'debug matches'. Lastly, the StarCraft environment continues running even when an agent is paused, making it difficult to make use of breakpoints.

For these reasons debug mode does not serve as an effective tool when it comes to debugging a StarCraft GOAL bot. In order to be more effective, debug mode will need to display information in a way that can scale to potentially more than a hundred agents. A suggestion to assist in this was proposed in Section 9.2.1. It was suggested to have the debug mode display information only about the units that are currently selected within StarCraft. By doing so, the user is able to easily indicate which agents they

are interested in to GOAL. Nevertheless, debug mode is known to be an effective tool in GOAL environments in which the number of agents can fit on the screen and the environment is more controlled. Additionally, other tools already provide effective means of debugging. As such, I do not believe that improving the debug mode for use with StarCraft is a high priority issue.

Logging

Logging is the most frequently used debugging tool throughout the development of ForceBot. Whereas debug mode is a real-time debugging tool, logging instead is a post-execution debugging tool. After execution has completed, the log files can be analysed to determine the cause of the bug. As it is possible to specify the type of information that is logged, the log can be made to be as verbose as is required. It is a tool which does not place a significant strain on the bot during runtime, meaning that the logging option can remain enabled at all times. If any strange behaviour is noted during a test, the logs can be analysed to determine the cause.

The logging tool has been an effective tool at assisting in debugging, for any bug whose cause is not immediately obvious, logging is the standard tool to determine the cause. There are two notable issues with logging, however. First, while the log settings can be modified to be as verbose as needed, the one thing that it does not log is when an agent crashes. This information can still be obtained, but this requires the default console log of Eclipse to be written to file and analysed afterwards instead, as detailed in Section 9.2.2.

Secondly, currently the log is difficult to navigate as a result of each log entry spanning a total of 12 lines. The XML format currently being used provides information which is not important, leading to larger log files than is needed. To resolve this problem, changes to the logging format have been suggested in Section 9.2.2. If these changes are made, the current 12 lines per log entry can be reduced to a single line.

Profiling

Profiling is a tool used to measure the amount of processing time spent on segments of the code. Writing faster code will allow for more cycles to be made, making the bot faster to respond to its environments. As is the case with logging, it is a post-execution tool and has the benefit of not significantly burdening the agent during runtime.

This tool has frequently been used over the course of ForceBot's development in order to determine current bottlenecks and analyse the performance improvements of changes. It is typically a good idea to profile the bot every few weeks to see whether any improvements can be made. A simple approach is to only profile individual rules, and sort the rules based on total execution time. A complicating aspect of profiling is that the results can greatly differ based on how the match plays out. As a result of this, when testing with ForceBot, I would typically always have ForceBot play against the same race and with the same strategy. Ideally however, using the same seed in each game would be better, to further improve consistency. However, this functionality requires BWAPI 4.1.2 or higher, and therefore cannot currently be used with GOAL, which uses BWAPI 3.7.5.

I believe that the existing profiling tool provides the necessary features to make use of the data, however the data is not presented in an easy to parse way. Suggestions for adjustments have been made in Section 9.3, in which the output is suggested to be changed to a format which can immediately be opened by commonly used spreadsheet software. Additionally, unimportant information should be removed from the output format. Furthermore, by adding an option to combine the collected data for the same agent types into a single file, the results of many different agents can be analysed at once.

At the start of this thesis, the following research question was raised: *How well do the tools provided for GOAL provide support for developing a StarCraft bot?* In order to answer this question the use of these tools over the course of this thesis were analysed. The logging and profiling tools provided for GOAL performed well, and saw extensive use over the course of ForceBot's development. However, the output of this data is not presented in a convenient manner. This is a problem which the debug mode, logging and profiling all possess. In order for these tools to be used to their full capability, the data must be presented in a cleaner and more compact form.

The debug mode performed poorly, and did not see use during development. This is a result of the large number of agents present in StarCraft, combined with the high level of detail that debug mode provides. In its current form, the amount of information made available to GOAL developers in StarCraft is too high, causing the debug mode to become difficult to make use of.

10.4. Computational Performance

At the start of this thesis, the following research question was raised: *How good is computational performance of GOAL in the StarCraft environment?* Based on the results of this thesis, my conclusion is that the computational performance of GOAL bots forms a bottleneck for the development of GOAL-based StarCraft bots. For example, the level of abstraction on which the combat simulator used by ForceBot operates, described in Section 6.3.1, is greater than that of comparable combat simulators written in languages traditionally used for StarCraft bots, such as C++ and Java. This is the result of performance limitations.

The computational performance limitations have served to shape a number of other aspects of ForceBot. Unlike most Zerg bots, ForceBot does not make frequent use of Mutalisks, as they require a high number of Actions Per Minute (APM) to make effective use of. Many of ForceBot's strategies also revolve around ending the match earlier rather than later, as ForceBot is poor at competing with traditional bots during later stages of a match as a result of the increasing number of units. Consequently, much of ForceBot's development has been aimed towards improving its strategic play, rather than its unit control. This is quite different from many existing bots, which utilise superhuman APM to gain advantages over their opponents.

In this section a number of underlying issues of GOAL and the connector that play a factor in causing the computational performance bottlenecks are discussed. Before these issues can be discussed, the system specifications of the computers running the StarCraft games should first be discussed. However, the only tournaments that has disclosed their computer specifications is SAIL, making the information rather limited:

- **Internal test tournaments** – Intel(R) Core(TM) i7-3820 CPU @ 3.6GHz with 32GB of RAM. However, this PC was also running the opposing bot, meaning only half of that can be expected to be used.
- **SAIL** – 1 CPU core of an Intel(R) Core(TM) i7 CPU @ 2.60GHz and up to 256MB of RAM.
- **AIIDE** – Unknown, but presumed similar or better than SAIL.
- **SSCAIT** – Unknown, but worse than SAIL.

The RAM requirements for SAIL are not a problem, as ForceBot will typically use around 180-220MB of RAM. Problems with SSCAIT were encountered some months into development, however, when ForceBot began slowing down StarCraft during play. In order to address this problem, a single-thread mode was added. By enabling this mode, all agents would run in a cycle thread in sequence, instead of multi-threaded, which solved the problem. Although this mode does reduce ForceBot's computational performance, due to the low computer specifications that SSCAIT is known to have, the reduction did not seem significant to ForceBot's performance prior to the single-thread mode on SSCAIT. The single-thread mode was also used on SAIL.

I believe that it is unlikely that the computer specifications of the internal test tournament significantly affect the outcome of these in comparison to SAIL. ForceBot has been designed to use less CPU by ensuring that the agents sleep frequently. As a result, in internal test tournaments ForceBot rarely exceeded 30% CPU, and usually staying below 20%. When these sleeps are removed, I have encountered ForceBot using over 60% CPU, however. Due to the multi-threaded nature of agents, GOAL-based StarCraft bots scale well with CPU, making a better CPU another solution to the computational performance problems. Nevertheless, current StarCraft AI tournaments do not provide such powerful CPU's, and as such ForceBot should ideally operate on the CPU's provided.

10.4.1. Prolog

Prolog is the language that is used by an agent to reason about its knowledge, beliefs and goals. There are two reasons that cause Prolog to serve as a computational performance bottleneck. The first reason is GOAL's bases, meaning the **`bel`**, **`percept`** and **`goal`** statements. These statements are internally abstracted in a way that does not allow them to be queried simultaneously. As a result, the below code will result in Prolog performing a separate **`percept`** query for each enemy in its belief base.

```
forall bel(enemy(Id)), not(percept(enemy(Id))) do ...
```

This is particularly problematic when there are a large number of percepts of that type being received. But any statement which queries multiple bases would become faster if this internal design were improved. In ForceBot there are over 250 statements which query multiple bases.

The second reason is the sharing of information between agents in GOAL. This aspect is problematic as a result of no information being inherently shared between agents. Each agent is its own process, with its

own beliefs, knowledge and goals. This means that if information must be held by all agents, the steps are repeated for every active agent.

One example of this is that during the startup procedure a large number of knowledge and beliefs must be inserted. Most of the information being inserted have already been inserted in other agents, however, but this information cannot be accessed by the new agent. Another example for this is the orders sent out towards all combat units by the combat manager. Whenever an order for one of the regions on the map is changed, this change must be sent to all agents, the old order deleted and the new order inserted. In a StarCraft match, the number of combat units will frequently be between 50 and 100. All of these repeated steps add to the overhead of Prolog.

Based on these reasons I believe that the existing abstraction of GOAL's bases within Prolog should be improved in the future, as the existing implementation produces too much overhead for a game with the complexity and number of agents that StarCraft possesses.

10.4.2. Agent Scheduling

One of the problems of having independent agents running asynchronous to the environment is that the difficulty of scheduling agents in a way that performs well. A good example of this can be seen in the start-up sequence of GOAL agents. When a new unit such as a Zergling is created, the agent for this unit is started by GOAL. This process takes around 5-10 milliseconds. Zerglings perform around 30 to 45 belief insertions in their init module, which can be performed at a rate of about 10-12 per milisecond. Therefore, this process is relatively fast.

However, the time between the start of an agent and its first movement can still be considerable. The reason for this is the scheduling in GOAL, particularly when the GOAL is running in the single-thread mode which was used for SSCAIT and SAIL. After starting the new agent, it is added to the list of agents, and will not run its first cycle until it receives its turn. In a 20 minute run of ForceBot, an average delay of 177 milliseconds between the start and the first cycle was found over a total of 670 agents. Combat units and Drones then send a task request to the managers, to which they receive a response in an average of 1434 milliseconds. Once they have a task, the agents will begin acting. In SSCAIT, the response times have often been found to be significantly longer, with agents not acting anywhere between 1 to 8 seconds depending on the state of the game.

This problem is not as significant once a task has been received – the agent will simply be slower to switch its current task. However, in the case of combat units this can result in the army being slow to retreat and suffering losses that could have been avoided. Using the multi-thread mode significantly alleviates the scheduling issue, however the problem that the agents are reliant on the cycle time of the managers persists.

These delays are a natural consequence of using independent agents, consequently there is not a clear solution to this problem. However, this is a problem that is common to distributed systems, and a solution may be found by looking at how distributed systems approach this problem. For example, distributed systems feature remote method invocation, which will handle messages the moment that they are received. This solution could also be used in GOAL through the addition of 'response' modules, responsible for responding to certain messages. This approach presents a problem if the agent agent receives a message requesting information that it is currently processing, however, resulting in concurrency issues.

10.4.3. Synchronisation

Traditional StarCraft bots operate in languages such as C++ or Java. These bots run synchronous to StarCraft: on each frame, BWAPI calls the `OnFrame` function of the bot, and StarCraft remains paused until code execution returns from this function. This means that BWAPI will not run faster than the bot is able to handle. GOAL instead runs asynchronous to the StarCraft environment, meaning that agent cycles do not match game frames. This does not mean that traditional bots no longer operates in real-time, however, as there are limits to how much this feature can be used. Tournaments such as AIIDE impose limits to how much time can be spent on individual frames. As GOAL operates asynchronously, these limits are not a concern, as GOAL will not pause StarCraft game frames.

However, the frames allows traditional bots to plan their time for each frame. One example of this is PurpleWave [13], ranked #2 in AIIDE 2017. PurpleWave sorts its various tasks such as micro management, economy planning and construction planning according to their individual priority and the time since the task were last performed. Using the previous execution lengths of those tasks it executes as many tasks as it has time for, and skips those that it does not have time for. Using this approach PurpleWave is able to accurately utilise its allotted time while prioritising tasks which have greater priority.

Currently, GOAL only offers agents to sleep for a chosen number of milliseconds, which is independent of the speed at which the StarCraft environment is running. It is therefore not possible to plan agent execution according to StarCraft game frames using GOAL. One way to improve this for StarCraft would be to allow agents to sleep for a number of game frames. An alternative approach, that would work in any environment, is to allow agents to sleep until a condition is met. This would allow agents to sleep until the desired `gameframe(<Frame>)` percept is received. This solution can also be used in other areas, such as having Drones which are gathering minerals sleep until there are either no more mineral fields, or it receives a new task.

10.4.4. Agent to Unit Mapping

GOAL utilises a 1-on-1 mapping of agents to units and buildings within StarCraft. This has been done in order to allow each unit and building to perform its own autonomous decision making. However, I believe that a 1-on-1 mapping is not ideal for StarCraft.

The reason for this is that a number of units and buildings do not require autonomous decision making, but rather centralised decision making. This is particularly the case for production and research buildings. This is related to a change that occurred to ForceBot V4 in Section 6.2, in which the decision making aspect of research buildings was removed. This was done because the general manager was capable of making a better decision on when research should be performed, as it is better aware of the overall state of the game. This meant that these agents no longer made decisions, instead performing the commands they received without thinking. In the same manner, the training and morphing of units was also controlled by the general manager. As these agents do not perform any decision making, there is currently no argument for having agents for buildings and Larvae. This holds for the majority of buildings which are only able to train units and perform researches. It is greatly preferred that any decision which results in resources being spent is controlled by a manager.

One of the solutions for this is to allow other agents to perform commands for other units that do not have an agent attached. Currently, the `cancel` command allows for an optional <ID> argument in order to cancel buildings that are under construction, as buildings do not possess agents until their construction is complete. If actions such as `train` and `research` also featured an optional <ID> argument, these agents can be safely removed. In most cases, allowing buildings and Larvae to be controlled by other agents only requires an additional <ID> argument to specify which building or Larvae should perform the command. Various buildings are capable of additional actions, however, and a number of distinctions must be made between buildings for this change in design to occur:

General
The information contained within the `status` percept must be made available through another means for units or buildings which are not connected to an agent. This information could be included in the `friendly` percept type, or a new percept type could be added. Often times the additional information is not needed, as such I believe that a new percept type is the best option for this.

Static defences
Static defences are able to perform decision making by deciding which target to attack. Static defences can already operate without agents if the default targeting priorities are not considered a problem. For this reason, I believe that developers should continue to be able to assign agents to static defences. The following buildings are considered static defences: Missile Turret, Photon Cannon, Sunken Colony and Spore Colony.

Bunkers
The Terran Bunker is also a static defence, but cannot attack itself. Instead, Marines, Firebats, Medics and Ghosts can enter the Bunker and safely attack from within it. Bunkers can be controlled by other agents by changing the `unitLoaded(<ID>)` percept type from unit-specific percepts to global dynamic percepts and including an additional <ID> parameter. Note that the existing `unitLoaded` percept type is still required, as it is used by transport units. In addition, the `unload(<ID>)` and `unloadAll` actions must be expanded with an optional <ID> argument.

Terran buildings
A number of Terran buildings are capable of using 'lift off' to fly. At this point, these units can fly around freely, and therefore perform decision making. Considering this, developers should continue to be able

to assign agents to the following Terran buildings: Command Center, Barracks, Engineering Bay, Factory, Starport and Science Facility. However, if these buildings are changed to allow for other agents to control them, the following actions will require an optional <ID> argument: `ability(<ID>)`, `ability(<X>, <Y>)`, `buildAddon`, `land(<X>, <Y>)`, `lift` and `move(<X>, <Y>)`.

Nydus Canals
Nydus Canals allow for instantaneous transportation between two fixed locations. A Nydus Canal must place its exit using the `build(<Type>, <X>, <Y>)` action. This process requires decision making, but it is likely that a developer will want this decision to be made by its combat manager, so that it can coordinate the army to utilise the Nydus Canal correctly. Bunkers can all be controlled by other agents by including an optional <ID> argument in the `build` action.

Remainder
All other buildings, including Larvae, do not perform any special behaviour. These can all be controlled by other agents by changing the `researching(<Type>)` and `queueSize(<Size>)` percept types from unit-specific percepts to global dynamic percepts and including an additional <ID> parameter that specifies which building is performing the research. Note that the existing `queueSize` percept type is still required, as it is used by Carriers and Reavers. Additionally, the `morph(<Type>)`, `research(<Type>)` and `train(<Type>)` actions must be expanded with an optional <ID> argument.

All of these distinctions make it difficult to generalise a change having all buildings and Larvae be controlled by other agents. For this reason, I believe that a number of them, namely Terran buildings capable of flight as well as Nydus Canals, should retain agents. However, for all other buildings and Larvae, this change is simple to make.

10.4.5. Connector Abstraction

One of the ways that has not yet been explored in order to improve the computational performance of GOAL is by utilising the StarCraft-GOAL Connector. At present, the connector provides actions for all of the commands available in StarCraft: move, attack, stop, patrol, etc. However, the connector could be updated to include 'action sequences', abstracted as standard actions. For example, an action could be included to perform kiting on a unit. An action such as `kite(<ID>, <X>, <Y>)` would command the unit to attack the target with the given ID, and upon the attack completing, the connector can immediately follow up with a move command to the given `(X, Y)` location.

These action sequences can be quite complex, and sometimes unique depending on the unit. Another example of this can be seen in Figure 10.1. In this image we can see that in order for a Vulture to kite enemies, 3 commands must be performed in sequence. Furthermore, the position of the patrol command is important, as a mis-positioned patrol command can lead to the Vulture losing speed while kiting, or failing to attack.

Figure 10.1: Vulture kiting commands – (1) Move (2) Patrol (3) Move

In addition to the micro techniques already mentioned, there are a number of other micro techniques which are relatively simple sequences of commands, such as:

- **Ability kiting** – Similar to basic attack kiting, but using abilities instead.

- **Air kiting/chasing** – Flying units will slow down upon approaching close to their target. To avoid this, a movement command can be used and cancelled into an attack command when already in attack range.
- **Mutalisk micro** – Essentially mixes the air kiting technique with the Vulture kiting to quickly approach and avoid slowing down after the attack.
- **Carrier hit and run** – Retreating the moment a Carrier begins ejecting Interceptors.
- **Unload-Load** – Units capable of transporting other units can quickly unload and load a unit to allow the transported unit to attack while making it difficult for the enemy to attack the transported unit.

Furthermore, there are a number of micro techniques which would be more difficult to precisely implement using actions. Examples include things such as burrowing a Lurker the moment an attackable enemy is in attack range, surrounding enemies with melee units such as Zerglings, and retreating with low health units. All of these micro techniques are currently difficult to perform as a result of the computational performance limitations of GOAL.

A large number of additional actions are needed in order to include all the micro techniques that are available to players in StarCraft as action sequences. However, by utilising the connector for this task these micro techniques can be successfully performed. The connector operates faster than the GOAL agents, and while GOAL runs asynchronous to StarCraft, the connector runs synchronous. Therefore, the connector is able to perform these micro actions with greater precision. I believe that the addition of action sequences into the StarCraft-GOAL Connector can greatly alleviate the computational performance problems of GOAL, as these action sequences would allow for precise micro without requiring agents to be able to run a cycle every frame of StarCraft.

10.5. Conclusion

In this section, the language, connector, tools and computational performance of GOAL-based StarCraft bots were all analysed. Out of these topics, I believe that the language and the tools do not pose major problems, although suggestions have been made for further improvements in these areas.

The main problem that GOAL-based StarCraft bots suffer is with regards to the computational performance. This restricts the micro of ForceBot, making it difficult to make the most with its individual units. Instead of this, ForceBot has to rely on strategy more than other bots. Considering the game performance of ForceBot, which I believe are for a large part the result of its strategic playstyle, I believe that it has therefore been proven that GOAL is capable of handling the complex strategies that arise in StarCraft and that the level of abstraction of the connector does not impose limits on strategies.

However, the problem of the computational performance of GOAL-based StarCraft bots is an issue which must be resolved in order for ForceBot to significantly improve its current game performance. In order to address this, a number of issues were discussed that contribute to this problem: Prolog, agent scheduling, synchronisation, agent to unit mapping and connector abstraction. Out of these issues, I believe that addressing the level of abstraction in the connector is the issue which should be addressed first, as the other issues are more difficult to address. Furthermore, I believe that it is the approach that is most likely to show success. While resolving the other issues would allow agents to make faster cycles, adjusting the level of abstraction by adding ‘action sequences’ instead reduces the need for agents to make fast cycles. Therefore, it becomes easier to further solve the computational performance problem by first addressing the level of abstraction of the connector.

11 Conclusions

Over the course of this thesis, ForceBot has become a capable bot. During the first milestone, ForceBot achieved the highest win-rate among the bots present in the 2017 MAS Project. A combat simulator was added in the second milestone, allowing ForceBot to make intelligent combat predictions. Unfortunately, the second milestone saw poor performance in AIIDE as a result of a critical bug. In the third milestone an internal testing environment was established, allowing more data regarding ForceBot's game performance to be collected. Using the results of these tests ForceBot was improved, in particular with regards to its strategic elements. This allowed it to achieve an above average score in SSCAIT, ranking 32nd among the 78 competing bots, and 7th among the 25 student bots. The internal testing was continued throughout the fourth milestone, notably adding various strategic responses to enemy behaviours. In SAIL, ForceBot ranked 42nd among the 88 competing bots. Finally, ForceBot improved its win-rate in the internal test environment from 41.1% during AIIDE to 84.4% with the final version on July 2018. Based on these achievements, ForceBot has shown the capability of performing at above average levels relative to existing StarCraft bots.

In this thesis, I set out to analyse the language, connector, tools and computational performance of GOAL-based StarCraft bots. Starting at the language, I did not encounter any major issues in this area over the course of ForceBot's development. The problems that were encountered were minor, and suggestions to address these problems were made.

The StarCraft-GOAL Environment Connector saw a great deal of changes over the course of this thesis. These changes have allowed it to mature, which is made clear by the lack of significant changes that occurred to it during the 2018 MAS Project which took place at the end of this thesis. The biggest change to occur to the connector during this thesis has been with regard to the flow of information. By allowing manager entities to receive percepts, and allowing agents and entities to subscribe to the percept types they were interested in, the flow of information can now be controlled by developers.

The tools provided by GOAL for development saw mixed usage. The debug mode performed poorly, and was not used throughout development. However, the logging and profiling tools provided sufficient information in order to successfully develop, debug and test a GOAL-based StarCraft bot. The output format of the logging and profiling tools was less than ideal. Suggestions were made to improve their output format in order to make their results easier to parse and navigate.

Finally, the computational performance of GOAL was established as the primary bottleneck with regards to the game performance of ForceBot. This problem results in ForceBot possessing poor micro, which affects its game performance. A number of underlying issues that lead to this problem were identified: Prolog, agent scheduling, synchronisation, agent to unit mapping and connector abstraction. Out of these, I believe that addressing the level of abstraction of the connector is likely to yield the best results. This is because resolving the other issues will allow agents to perform faster cycles, but by adjusting the level of abstraction reduces the need for agents to make fast cycles is reduced instead. As such, moving forward I believe that addressing the level of abstraction of the connector is the first issue that should be addressed.

My overall conclusion from these analyses is that the tools do not pose themselves as a problem in the development or game performance of GOAL-based StarCraft bots. However, the language and the connector are two areas in which adjustments can be made in order to improve the main issue of computational performance.

Furthermore, a significant portion of ForceBot's development has been aimed towards improving its strategic strength, particularly in order to make up for its poor micro skills as a result of the computational performance problems. In doing so, I did not encounter any problems in the implementation of strategies within ForceBot. Moreover, the higher information density of the GOAL language allows for such strategic decision making to be easily implemented. My conclusion is therefore that no aspect of GOAL poses a problem for the strategic capabilities of GOAL-based StarCraft bots. However, the poor micro does result in a number of strategies being less effective. This is because strategies that rely on units such as Mutalisks require good micro. These strategies can still be used, and ForceBot successfully uses them depending on the situation, but improving the micro capabilities of GOAL-based StarCraft bots would result in these strategies becoming significantly stronger. At present, I believe that until the computational performance problems are addressed, a GOAL-based StarCraft bot will be unable to match the game performance of StarCraft bots written in traditional programming languages.

The final research question to be posed in this thesis was the following: *how does a cognitive multi-agent system compare to traditional StarCraft bots?* Throughout the course of this thesis I did not encounter any difficulties in implementing the complex strategies and decision making that is present in StarCraft, and I believe that, on a strategic level, cognitive multi-agent systems can perform on the same level as traditional C++ or Java bots. The primary bottleneck encountered within this thesis is caused by computational performance problems originating from GOAL. This problem may not be present in cognitive multi-agent systems besides GOAL, and can be improved over time for GOAL itself. The problem of computational performance is one that other cognitive multi-agent systems should be aware of however, as the difficulty of running over a hundred independent cognitive agents concurrently is evidenced by this thesis.

11.1. Recommendation for Future Research

There are a number of areas in which ForceBot can be improved upon in the future, as well as further research that can be performed for GOAL-based StarCraft bots. The following is a list of recommendations for future research:

1. The issues underlying the computational performance problems can be resolved in the future using the work performed in this thesis. If ForceBot becomes able to perform micro with skill close to or equal to that of existing C++ and Java bots, a significant game performance increase is to be expected, and the effectiveness of GOAL-based StarCraft bots should then be re-evaluated.

2. All of the top performing bots in StarCraft feature learning behaviour, allowing them to learn strategies that have been effective or ineffective against certain opponents in the past. All of the functionalities required to add learning behaviour into ForceBot are present in GOAL, but the addition of learning behaviour has been left for future research. In order to implement this, the effectiveness of strategies and build orders against specific opponents or races must be stored using Prolog's write functions. In subsequent games, these files can be read and parsed in order to determine the strategy that ForceBot should. Learning can also be used in other areas, such as determining effective army compositions, defence quantities and economic behaviour.

3. This thesis has focused purely on the game performance of StarCraft bots and the aspects that accompany that. However, the challenge of developing a StarCraft bot can also be approached with the aim of designing a human-like bot. Professional human players average around 300 Actions Per Minute (APM). Most existing bots frequently perform over 10,000 APM, making them distinctly inhuman and mechanical. ForceBot's APM varies from 200 to 1500 depending on the state of the game, with an average of around 500. Although this exceeds that of human players, there are differences in APM between humans and bots. This is because bots control units on a per-unit level, while humans will often command multiple units at a time using groups. As a result, ForceBot's APM is fairly human-like despite being higher than that of humans. As there is a demand for human-like bots, a future research could look into the human-like properties of GOAL-based StarCraft bots.

Bibliography

[1] SSCAIT 2017/18 crosstable. URL https://purplepie.bitbucket.io/sscait18.html.

[2] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. Debugging distributed systems. *Queue*, 14(2):50, 2016.

[3] Michael Buro and David Churchill. Real-time strategy game competitions. *AI Magazine*, 33(3):106, 2012.

[4] Murray Campbell, A Joseph Hoane, and Feng-hsiung Hsu. Deep Blue. *Artificial intelligence*, 134(1-2): 57–83, 2002.

[5] Michal Certicky. SSCAIT 2017/18 results. URL http://challonge.com/pckilndo.

[6] Jim X Chen. The evolution of computing: AlphaGo. *Computing in Science & Engineering*, 18(4):4–7, 2016.

[7] Dave Churchill. AIIDE 2017 results, . URL https://www.cs.mun.ca/~dchurchill/starcraftaicomp/2017/.

[8] Dave Churchill and Vincent Koeman. StarCraft AI Tournament Manager. URL https://github.com/Venorcis/StarcraftAITournamentManager.

[9] David Churchill. SparCraft GitHub, . URL https://github.com/davechurchill/ualbertabot/tree/master/SparCraft.

[10] David Churchill. The Current State of StarCraft AI Competitions and Bots. 2017.

[11] David Churchill, Mike Preuss, Florian Richoux, Gabriel Synnaeve, Alberto Uriarte, Santiago Ontanón, and Michal Certickỳ. Starcraft bots and competitions. 2016.

[12] Mihaly Csikszentmihalyi. *Flow and the psychology of discovery and invention*. New York: Harper Collins, 1996.

[13] Dan Gant. PurpleWave GitHub repository. URL https://github.com/dgant/PurpleWave.

[14] Igor Dimitrijevic. Iron Bot. URL http://bwem.sourceforge.net/Iron.html.

[15] Victor do Nascimento Silva and Luiz Chaimowicz. On the development of intelligent agents for MOBA games. In *Computer Games and Digital Entertainment (SBGames), 2015 14th Brazilian Symposium on*, pages 142–151. IEEE, 2015.

[16] Arpad E Elo. *The rating of chessplayers, past and present*. Arco Pub., 1978.

[17] Johan Hagelbäck. OpprimoBot. URL https://github.com/jhagelback/OpprimoBot.

[18] Koen V. Hindriks and Wouter Pasman. Environment Interface Standard. URL https://github.com/eishub/eis.

[19] Koen V Hindriks, Birna Van Riemsdijk, Tristan Behrens, Rien Korstanje, Nick Kraayenbrink, Wouter Pasman, and Lennard De Rijk. Unreal goal bots. In *Agents for games and simulations II*, pages 1–18. Springer, 2011.

[20] Hindriks, Koen V. and Koeman, Vincent J. and Pasman, Wouter. GOAL Confluence. URL https://goalapl.atlassian.net/wiki/spaces/GOAL/overview.

[21] Haomiao Huang. Skynet meets the Swarm: how the Berkeley Overmind won the 2010 StarCraft AI competition. URL https://arstechnica.com/gaming/2011/01/skynet-meets-the-swarm.

[22] Matthew Johnson, Catholijn M Jonker, M Birna van Riemsdijk, Paul J Feltovich, and Jeffrey M Bradshaw. Joint Activity Testbed: Blocks World for Teams (BW4T). In *ESAW*, volume 9, pages 254–256. Springer, 2009.

[23] Dan Klein. The Berkely Overmind Project, 2010.

[24] Vincent J. Koeman, Harm J. Griffioen, and Danny C. Plenge. StarCraft-GOAL Environment Connector. URL `https://github.com/eishub/Starcraft`.

[25] Vincent J Koeman, Koen V Hindriks, and Catholijn M Jonker. Designing a source-level debugger for cognitive agent programs. *Autonomous Agents and Multi-Agent Systems*, 31(5):941–970, 2017.

[26] Vincent J. Koeman, Harm J. Griffioen, Danny C. Plenge, and Koen V. Hindriks. Designing a Cognitive Agent Connector for Complex Environments: A Case Study with StarCraf. 2018.

[27] Erik Kvanli and Eirik M Hammerstad. A Coalition based Agent Design for Heroes of Newerth. Master's thesis, Institutt for datateknikk og informasjonsvitenskap, 2014.

[28] Haewoon Kwak, Jeremy Blackburn, and Seungyeop Han. Exploring cyberbullying and other toxic behavior in team competition online games. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3739–3748. ACM, 2015.

[29] Raúl Lara-Cabrera, Carlos Cotta, and Antonio J Fernández-Leiva. A review of computational intelligence in RTS games. In *Foundations of Computational Intelligence (FOCI), 2013 IEEE Symposium on*, pages 114–121. IEEE, 2013.

[30] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[31] Cyrus F Nournai. Multiagent Chess Games. In *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence*, pages 45–52, 1997.

[32] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game AI research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):293–311, 2013.

[33] OpenAI. OpenAI Five. URL `https://blog.openai.com/openai-five/`.

[34] Jay Scott. SAIL bots and maps, . URL `http://satirist.org/ai/starcraft/blog/archives/551-SAIL-bots-and-maps.html`.

[35] Jay Scott. SAIL race balance tables, . URL `http://satirist.org/ai/starcraft/blog/archives/550-SAIL-race-balance-tables.html`.

[36] Jiefu Shi and Michael L Littman. Abstraction methods for game theoretic poker. In *International Conference on Computers and Games*, pages 333–345. Springer, 2000.

[37] Victor do Nascimento Silva and Luiz Chaimowicz. MOBA: a New Arena for Game AI. *arXiv preprint arXiv:1705.10443*, 2017.

[38] Victor do Nascimento Silva and Luiz Chaimowicz. A tutor agent for MOBA games. *arXiv preprint arXiv:1706.02832*, 2017.

[39] Leon Sterling and Ehud Y Shapiro. *The art of Prolog: advanced programming techniques*. MIT press, 1994.

[40] Alex Turner. Soar-SC: A Platform for AI Research in StarCraft: Brood War. 2013.

[41] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[42] Michal Čertický. Student StarCraft Artificial Intelligence Tournament Ladder. URL `https://sscaitournament.com/index.php?action=scores`.