Extending a StarCraft AI Behaviour Library

Alex Aiton

Bachelor of Science in Computer Science with Honours
The University of Bath
April 2014

Extending a StarCraft AI Behaviour Library

Submitted by: Alex Aiton

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

Your abstract should appear here. An abstract is a short paragraph describing the aims of the project, what was achieved and what contributions it has made.

Contents

List of Figures

List of Tables

Acknowledgements

Add any acknowledgements here.

Chapter 1

Introduction

This is the introductory chapter.

Chapter 2

Literature Survey

In 2012, Simon Davies created an AI agent for the real-time strategy game StarCraft: Brood War. He followed the Behaviour Oriented Design development methodology and focused his efforts on an agent for the Zerg race. This project aims to generalise the work Davies performed to allow for agents of other races and possibly implement improvements.

StarCraft has recently seen increased development in the field of AI, due to the release of a convenient API for the purpose of AI development. Before beginning focused development effort, a survey of the work done in this area was conducted. What follows is the results of this survey, including an explanation of what StarCraft is; the field of AI and agents in general; the Behaviour Oriented Design methodology; an overview of what Davies accomplished; and a review of some techniques that could be used to improve Davies agent.

2.1 StarCraft

StarCraft is a real-time strategy game where the player commands an army from one of three distinct races [1]. It was released in 1998 by Blizzard Entertainment and its expansion StarCraft: Brood War was released later the same year (*Blizzard Entertainment – StarCraft*, n.d.). During a game, the player controls their army from a top-down perspective, or "God's eye view" (see figure 2.1). The army must harvest two limited resources, minerals and vespene gas, and use them to construct buildings, recruit units and research upgrades with the ultimate goal of destroying the opponent. The game can be said to be split into three stages: The opening, where choosing the most suitable order of construction is vital due to limited resources; the mid-game, where players build up for larger attacks and must expand to gain more resources; and the end-game, where one player has gained the advantage and should be looking to make a very large push to secure it (Yi, 2011).

The three races differ from each other substantially in both play style and unit ability. The insectile Zerg focus on small, fast and cheap units and swarm tactics; the psychic and

[1] Races can be thought about as teams or factions. Race is the term used in the StarCraft community

advanced Protoss focus on expensive but extremely powerful units; and the human Terran are the balance between the two, not as weak or as numerous as the Zerg, but not as powerful or as expensive as the Protoss. They also focus on attacking from range. Due to the game's age and popularity, there are many online communities that have collated a large amount of information about the game and effective strategies, such as the StarCraft wiki (*StarCraft Wiki*, n.d.) and team liquid (*Team Liquid*, n.d.).

StarCraft has regularly found itself used as a subject in academia, probably due to its prominence and well-balanced game play. Uses can vary from essays on what StarCraft's design communicates (Galloway, 2007), analysis of the game's network traffic (Dainotti, Pescape and Ventre, 2005) , to algorithmically generating playable maps (Togelius, Preuss, Beume, Wessing, Hagelback and Yannakakis, 2010). The creation of the Brood War Application Programming Interface (BWAPI) in 2010 (*BWAPI*, n.d.) has led to increased usage as a platform to explore and construct AI and a target for AI competitions (*BWAPI - Competitions*, n.d.).

Figure 2.1: A screenshot of StarCraft during a game. It shows the base of a Protoss player

2.2 Artificial Intelligence

AI is a wide ranging field concerned with the research and development of machines/programs capable of intelligence. It includes a wide variety of sub-fields like "computer vision, natural language, decision theory, genetic algorithms and robotics" (McCorduck, 2004).

There is the concept of strong AIs, which are the kind of AI visualised by the general public and science fiction writers, true "machines that think", conscious and capable of human emotion (Kurzweil, 2005). AIs that don't attempt to simulate the entirety of human intelligence can be called weak AI, and it's probably safe to say all AI's created to date are weak AI, including those found in games.

The concept of an agent is a popular one in AI. An agent is a system capable flexible, autonomous action in some environment (Wooldridge and Jennings, 1995). That's a rather vague definition, so to expand it; an agent is a computer system which has a goal to accomplish, resides in an environment which it can effect some change upon and can react to changes in that environment. An environment does not have to be physical, but there are other classifications to consider (Russell, Norvig, Canny, Malik and Edwards, 1995):

- Accessible vs. inaccessible How much information is available to the agent about the environment? Accessible environments provide complete, accurate and up-to-date information about the environment's state.
- Deterministic vs. non-deterministic What happens when the agent acts? In deterministic environments an action has a single guaranteed effect, with no uncertainty.
- Static vs. dynamic Is anything else changing the environment? Static environments only change due to actions by the agent.
- Episodic vs. non-episodic How is the agent rated? An agent rated in an episodic environment takes part in several discrete episodes which remain independent, so past and future performances aren't important. For example, an AI taking part in a tournament could consider each game to be a single episode.
- Discrete vs. continuous How many actions are available to the agent? Discrete environments have a very limited and fixed number actions and a small amount of required knowledge in it.

StarCraft can be argued to be an inaccessible, deterministic, dynamic, continuous environment with the possibility to be episodic and accessible depending on game settings (Davies, 2012). The majority of StarCraft games have the fog of war setting enabled, so only objects within unit vision are visible and have up-to-date information. It is possible to disable the setting, turning it into an accessible environment. Almost all agent actions are deterministic, though there is some small randomness in ranged attacks. StarCraft is played against another agent, so must be dynamic. The number of units and the number of positions they can be in make it more like a continuous environment than a discrete one, though I think the number of actions available is just extremely high rather than infinite.

2.3 Behaviour Oriented Design (BOD)

BOD is an AI development methodology conceived by Bryson (Bryson, 2001). It details both an agent architecture and a design methodology. The BOD architecture has two parts, a modular library of behaviours (actions, senses and state) and parallel rooted, ordered, slip-stack, and hierarchical (POSH) reactive plans for action selection. The methodology specifies how to decompose an agent's behaviours and emphasises an iterative development approach for implementation (Bryson, 2003). BOD focuses on the design of the AI system, aiming to produce agents that are easy to extend and maintain. It's modularity allows many different AI techniques to be used in tandem, encouraging the developer to use whatever approach fits an individual problem best (Gaudl, Davies and Bryson, n.d.).

2.3.1 Architecture

As stated, the BOD architecture consists of two main parts:

The Behaviour Library Behaviours are like object oriented design (OOD) objects and encapsulate how to do something, including perception and actions. Behaviours should be modular, can have state, can implement learning techniques, or even be wrappers for external libraries. This separation from the action selection also facilitates code re-use and allows modification of behaviours without affecting plans (Gaudl et al., n.d.).

POSH plans and action selection POSH plans are a hierarchy of behaviours and their triggers (Gaudl et al., n.d.). The primitives of the hierarchy are actions and senses. These primitives are formed into three groupings, action patterns, competences and drive collections, in order of increasing complexity. Action patterns are simple sequences of primitives; competences are basic reactive plans, a prioritised list of actions and triggers; drive collections are the root of the hierarchy and determine where the agent's attention should be focused (Bryson, 2003).

2.3.2 Decomposition

BOD provides a set of initial steps to begin the decomposition of behaviours and the construction of plans. The steps are as follows (Bryson, 2003):

1. Specify at a high level what the agent is intended to do.
2. Describe likely activities in terms of sequences of actions. These sequences are the basis of the initial reactive plans.
3. Identify an initial list of sensory and action primitives from the previous list of actions.

4. Identify the state necessary to enable the described primitives and drives. Cluster related state elements and their primitives into specifications for behaviours. This is the basis of the behaviour library.

5. Identify and prioritize goals or drives that the agent may need to attend to. This describes the initial roots for the POSH action selection hierarchy.

6. Select a first behaviour to implement.

As BOD is meant to be performed iteratively, getting these steps right initially isn't paramount, but will save some effort later on.

2.3.3 Methodology

BOD champions an iterative development cycle and rapid prototyping. It specifies to only implement a section at a time, to fully test and debug each section and to re-evaluate the specification after each section is complete. During this evaluation, the focus is to simplify, if feasible, anything that has gotten too complex. Plan elements are also analysed, to determine if they are still doing what they were intended to do. Complex primitives should become action patterns; Long action patterns should either be reduced into primitives or extended into a competence; Simple, deterministic competences should become action patterns and complex competences should be broken down into simpler competences. The line where a element becomes too complex can be a bit blurry, though BOD does offer guide lines. It is the designer's role to use trial and error and their own experience to determine the best way to keep the AI simple (Bryson, 2005).

2.4 Simon Davies' AI

In 2012, Simon Davies used BOD to create a Zerg AI bot for StarCraft (Davies, 2012). The strategy it follows is quite simple; it builds up a simple army until it has a certain number of units and then it attacks the enemy until it has lost a certain number of units. It also builds defences and expands its resource harvesting capacity.

This strategy leaves the bot weak in the early game, and stronger in the late game. It doesn't use information about its opponents to modify this strategy, so often lost when opponents tried an early game push. The bot also has problems with the implementation being a bit rough; there's only two hard coded base expansions; melee (close combat) units will try to attack air units; and forces wait on a single slow scout to find the enemy, when they could do it themselves.

The architecture of the bot's implementation has three layers, detailed below and visualised in 2.2

The POSH layer This layer contains the POSH action selection and plans, and a thin jython behaviour library which calls into the behaviour layer.

The Behaviour layer This layer contains the behaviour implementations in Java and it uses a Java native interface to BWAPI (*JNIBWAPI*, n.d.) to talk with the StarCraft layer.

The StarCraft layer This layer contains the actual game and the BWAPI library which is implemented in C++ and communicates to the game through DLL injection.

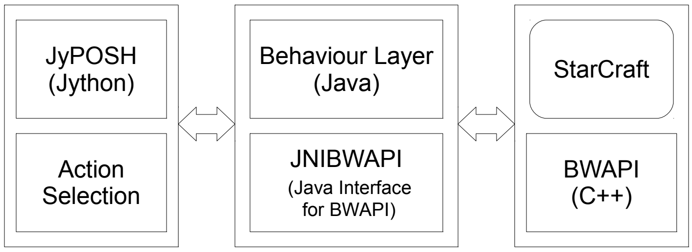

Figure 2.2: The architecture model for the StarCraft BOD AI. (Davies, 2012)

The architecture is slightly more complicated than it needs to be, but Davies had trouble linking the python of POSH to the C++ used by BWAPI, so Java is used as an intermediary. Simplifying the architecture of the project now would require a major re-write and possibly a re-implementation of POSH in C++. This is outside the goal and scope of this project and in no way necessary, as long as JNIBWAPI remains up-to-date with BWAPI.

The Java behaviour library of the bot is implemented as a collection of managers, which split the behaviours into correlated sub-sections. The decomposition Davies chose appears effective and his reasoning is solid. Most of the behaviours and the action patterns are specific to the Zerg race, and as mentioned above, the other races do play quite differently. However, the base mechanics of the game (resource gathering, construction, research) are nearly identical between races and it is these areas I will look to generalise.

2.5 Possible ideas for expansions

This section covers various AI strategies and techniques that have been applied StarCraft and offers some opinion on their applicability for improving the bot's performance.

2.5.1 Unit micro-management strategies

Unit micro-management (micro) refers to the low-level control of units; how the units are positioned, what the units attack, and how the units attack (*Team Liquid, Micro*, n.d.). The micro of Davies' bot is relatively simple compared to some other examples. There is a single army that moves towards the enemy's base. Each unit prioritises visible enemies based on their type and how far away they are. Each unit will then attack the unit it calculates to have the highest priority. While simple, this can be effective in some situations. It does have problems, units don't make use of any abilities they possess and units don't consider their position, so a weak ranged unit can end up in melee range. There are other works that show possible paths for improvement.

Potential fields

Adapted from an idea in physics, this is where individual entities in the game are given an attractive or repulsive force. So units are attracted to weaker units while being repulsed from stronger or more dangerous units. Units can also be affected by changes in their own circumstances, perhaps retreating while under fire. The advantage of potential fields is that a single function can control lots of units, however the strength of the forces involved must be figured out manually and carefully considered to avoid units becoming stuck in local maxima. Genetic algorithms have been used to reduce the workload needed to optimise the values (Rathe and Svendsen, 2012). The Berkeley Overmind (*Berkeley Overmind*, n.d.) implemented potential fields to such success that it was able to beat an ex-professional player (Huang, 2011). There are also an open source implementation available (*A StarCraft AI bot*, n.d.).

Monte-Carlo planning

Planning is usually associated with large scale strategy, due to its time cost and the large search space of micro-management. Wang, Nguyen, Thawonmas and Rinaldo (2012) have applied it to the micro-management of units using a Monte-Carlo planning method. Monte-Carlo methods rely on entering random samples into a simulation and analysing the results (*Monte-Carlo method*, n.d.). Wang et al created a simulation of the game state and then evaluated several predefined stochastic plans. They tested their AI in game and an imbalanced scenario (their AI was at a disadvantage) and it was able to beat beginner players and the original AI with some consistency, though was still beaten by an expert. This does show some potential for planning at a micro level. However, plan evaluation must be weighted manually and the simulator is limited to a few types of units and a small environment. It is unclear if this method could be scaled to a full game, as they have concerns about simulator speed .

Bayesian Modelling

There is a project that focuses on using Bayesian models for all aspects of a StarCraft AI (Synnaeve and Bessiere, 2012) which includes unit control (Synnaeve and Bessiere, 2011). This uses the distribution of game elements to decide what to do with a unit next. Units receive tactical goals as sensory inputs and maintain a finite state machine (FSM) for different modes (attacking, defending, etc.). It uses re-enforcement learning to build its probability tables and succeeded quite well in trails against both the original AI and against winners of the 2010 AIIDE competition. Their implementation is available open source.

2.5.2 Macromanagement strategies

Macromanagemt (macro), as used in computer gaming, is a term for the economic or large-scale strategy of the player. For StarCraft, this covers resource collection and spending; the decision on what buildings or units to build. Davies' bot is quite effective on the resource collection side, but is very restricted on the 'what to build' side. Its strategy is very rigid and doesn't adapt to the enemies actions, which is a key part of successful StarCraft play. Improving and generalising his work will require creating several strategies, and should look into making them adaptable. There is work done in StarCraft in this area.

Micro Focus

Safadi and Ernst (Safadi and Ernst, 2010) take the interesting position that a agents macro strategy is fairly unimportant. Their view is that planning and pattern recognition is something humans excel so well at that a bot couldn't hope to match it, the bot would require a database of possible strategies which would require continuous updates as the meta-game evolves. Instead the bot should play to it's advantages in the multi-tasking area; 300 unique action per minute is considered the average for StarCraft professionals, but it would be trivial for a computer to match and beat that. They did have some success and this is comparable to how the Berkeley Overmind succeeds. However, StarCraft units generally have a "Rock Paper Scissors" dynamic; they are strong against some units and are countered by others. Effective unit-level play needs effective unit choice otherwise it can be easily exploited.

Planning

There are many replays of StarCraft games publicly available (*StarCraft Replays*, n.d.), some of which are of high level play. Using BWAPI, it's possible to retrieve lots of data from these replays and data mine the results. There are several papers that do this and use the data to create Bayesian models of what the opponents are likely to be doing (Synnaeve and Bessiere, 2011; Hostetler, Dereszynski, Dietterich and Fern, 2012; Weber and Mateas, 2009). This has the advantage that the bot gains expert-level knowledge without designers needing to know expert strategies. Planning and prediction are popular areas in the StarCraft AI field, probably due to the availability of vast amounts of completed game data. The main strategy in StarCraft revolves around predicting and countering the opponents high-level decisions, so a successful bot does need to react in someway to what it thinks its opponent is doing.

Case-based reasoning

Certicky and Certicky (2013) implemented cased-based reasoning (CBR) for army composition. CBR is a cyclical process where there is a databases of problem cases with solutions. For each problem, a case is chosen that best fits. The solution is adapted to fit the current problem and if it works it is added back to the database as a new case. Certicky constructed cases based on the ratios of units that the opponent has and the solutions are the recommend ratios for the bot's army. This process has the potential to be useful, but it needs quite a bit of expert knowledge about common army ratios and their counters. Additionally, this relies on good scouting and accurate information, if the bot picks the wrong case it could be disastrous. This method is similar to the Bayesian planning mentioned above, but there is more learning done on the fly, which might impact performance.

2.6 Summary

This literature survey has covered a wide range of topics. It has

Chapter 3

Requirements

This chapter covers the basic requirements specification that was followed for the project.

3.1 Functional Requirements

- A POSH plan must be created that can control a Protoss bot:
  - The bot must be able to gather resources as the Protoss
  - The bot must be able to create Protoss units and buildings
  - The bot must be able to attack the enemy as the Protoss
- A POSH plan must be created that can control a Terran bot:
  - The bot must be able to gather resources as the Terran
  - The bot must be able to create Terran units and buildings
  - The bot must be able to attack the enemy as the Terran

3.2 Non-Functional Requirements

- The behaviour library refactoring should not negatively affect the existing Zerg agent's performance.
- The behaviour library should avoid race specific behaviour where possible

3.3 Inherited Non-Functional Requirements

Davies (2012) specified several non-functional requirements that are still relevant to the project.

- The application must use the BWAPI interface for communicating with StarCraft
- The agent must be able to work in the latest version of StarCraft: Brood War
- The AI should be designed in a modular and extensible fashion, to allow for future developments and improvements
- The application must be able to run at a minimum of speed of a frame every 42ms, so not to slow down the game, as specified by the current AIIDE rules.
- All relevant AI behaviours must be accessible to the POSH action selection mechanism

Chapter 4

Design

This chapter covers the design work done for the project, including an overview of the initial class structure of the behaviour library, the changes needed to generalise the behaviour library and the process

4.1 Initial Behaviour Library Structure

The behaviour library is the main place that needs to be refactored and generalised. As mentioned in 2.4, the behaviour library is decomposed into eight classes which manager the different areas of the game. Davies' class diagram can been seen in figure 4.1 and he outlined the class thusly:

Resource Manager Senses for actual and predicted resource counts, as well as reserving resources for future building construction.

Unit Manager Allows the selection of specific unit types. Mostly used by other managers.

Worker Manager Keeps track of workers and holds the behaviour for collecting minerals and gas. Used by the building manager for accessing workers for construction purposes.

Intelligence Manager Holds a knowledge base of enemy locations, and sends units to gather intelligence on enemy positions.

Building Manager Allows for construction of new buildings. Calculates viable building locations, and also holds senses for which buildings are currently possessed by the player.

Production Manager Behaviours for the creation of new units.

Upgrade Manager Behaviours for researching upgrades for units.

Military Manager Behaviours for the movement of military forces. This includes attacking the opponents base, retreating, and contains a list of priority targets.

The immediate design work was to analyse each class's code and identify methods that could be generalised or that were already general. Any methods that are too specific would be moved into a race specific sub-class of the manager.

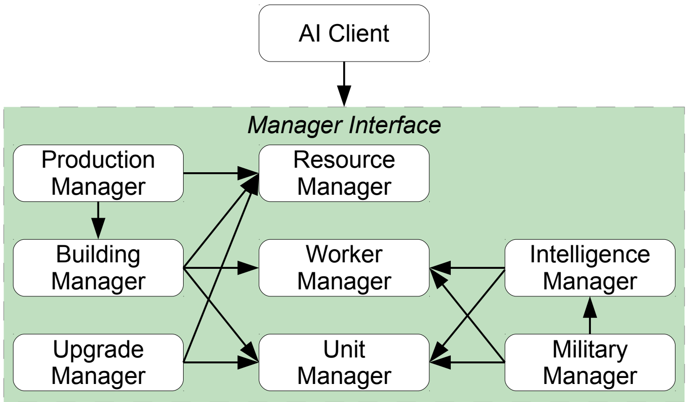

Figure 4.1: Davies' class diagram that shows the Managers used for the categorising the different behaviours. The connections between the managers shows which classes call behaviours from other managers.(Davies, 2012)

4.2 Behaviour Library Structure Changes

4.2.1 Initial Design

After statically analysing the code, it was found that half the classes could be fully generalised or were already general enough to not need any modification at all.

These classes were:

Resource Manager Every race competes for the same resources and they each handles resources in the same way.

Unit Manager The methods in this class are already parameterised by unit type.

Worker Manager Each race uses workers and gathers resources in nearly the same way. Generalisation is mostly changing method and variable names.

Intelligence Manager Generalisation just needs to adjust the unit used for scouting.

The remaining classes were sub-classed in a general way, which can be seen in figure 4.2. During the initialisation of the AI client, the appropriate sub-class will be created, based on what BWAPI reports about the player's selected race.

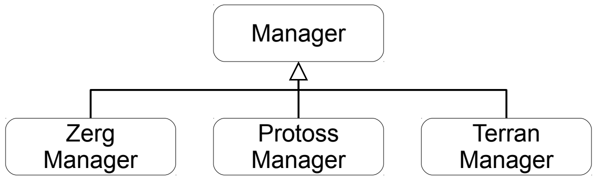

Figure 4.2: The general class structure used when methods couldn't be generalised.

4.2.2 Iterative design process

After this initial design work, I proceeded with implementation. The iterative development methodology used in BOD means the design is repeatedly modified as work progresses. The process that was followed for implementation was a very slightly altered version of the BOD development methodology which went like this:

- Choose a behaviour of the Zerg agent.
- Implement and test the behaviour for the other races.
- Evaluate race specific behaviours, looking for common patterns that could be made general.
- Implement and test the generalisation.
- Repeat the process.

This differs from BOD in that there is less focus on revising the specification, more on getting the other races capabilities up to par with the Zerg. The Zerg agent was used as the specification for the other races agents, so while revisions weren't avoided, they were not the focus of the process.

4.2.3 Final Class Structure

The final class structure of the behaviour library can be seen in figure 4.3. The race-specific sub-classes have been hidden as they increase the complexity of the diagram without really adding any information. The only change to the structure that came up during implementation was to the intelligence mananger. It was found to be better to split the scouting control(moving the unit around) into its own class and have the intelligence manager start the scouting. This allowed for race specific scouting behaviour without code duplication.

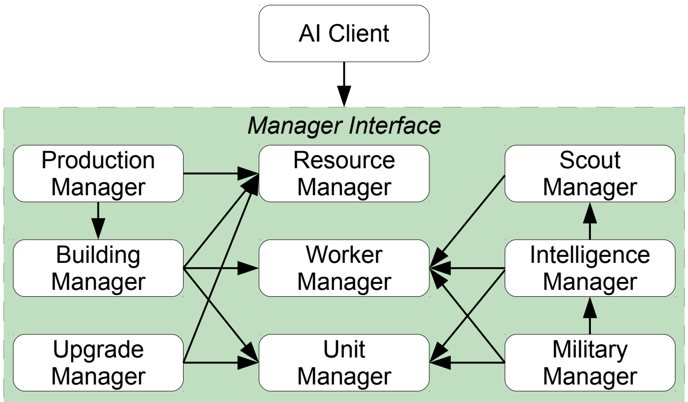

Figure 4.3: The final class structure of the behaviour library. Classes with a "[*]" in their name have race specific behaviour in sub-classes in the style of figure 4.2.

Chapter 5

Implementation

5.1 Behaviour Library Generalisation

This section covers some of the work done generalising the Java behaviour library.

5.1.1 API issues

This subsection explains the issues experienced by updating the external libraries that the AI uses and details how they were overcome.

BWAPI and JNIBWAPI had both released newer version since Davies's (2012) work. It was decided to update the project to use these newer versions as they offered bug fixes and the JNIBWAPI update exposed more BWAPI functionality. The update went fine, and seemed to be without issues. It was then noticed that workers would not act on any command they were given, unless it was sent during a game tick call.[1] Actions from POSH are called from outside the game tick and were seemingly ignored by the engine.

To solve this required the implementation of a queue in the worker manager class where a POSH action can add the order data to when it wants to tell the game to do something. Order data generally includes the type of order; the unit id of what is being ordered; an x and y value of the location of the order, or another unit id to perform the order on. This data is stored in a short class. When the game tick gets called, all entries in the queue are processed and the orders sent to the game.

The bug was not limited to just workers but also affected producing units and researching upgrades, so the same fix has been applied to the production and upgrade manager classes.

[1] A common feature of real-time games. This is a function called regularly that will perform all the computation for the game engine. For StarCraft/BWAPI the function is called gameUpdate() and gets called at 20Hz.

5.1.2 Parameterisation of building, recruitment and research

This subsection explains how the building, production and upgrade managers were moved from specific behaviour methods to general, parameterised methods.

In the initial stages of the project, the main content of the race-specific managers for building, production and upgrades was methods to build specific buildings, produce specific units and research specific upgrades. An example can be seen in listing 5.1. As more of these methods were added for each race, a pattern was noticed.

The pattern is:

- The progenitor unit type for the requested unit type is determined, hard-coded initially.
- A free unit of progenitor type is found.
- It is checked if the necessary resources and research are available.
- The action is performed.

Listing 5.1: The original method used to produce a Zergling[a]

```
public boolean spawnZerglings(){
    for (Unit unit : bwapi.getMyUnits()) {
        if (unit.getTypeID() == UnitTypes.Zerg_Larva.ordinal()) {
            if (resourceManager.getMineralCount() >= 50 &&
                resourceManager.getSupplyAvailable() >= 1 &&
                buildingManager.hasSpawningPool(true))
            {
                bwapi.morph(unit.getID(),
                    UnitTypes.Zerg_Zergling.ordinal());
                return true;
            }
        }
    }
    return false;
}
```

[a]Zerglings are the basic unit of a Zerg player's army. They are small, fast, and have a melee attack

Using methods provided by JNIBWAPI, it is possible to find out all the needed information just from the unit type. The code to do this can be seen in listing 5.2. The important methods that allow this simplification is canMake() method provided by JNIBWAPI and the getLeastBusyUnitofType() method added to the unit manager.

canMake() has two forms in BWAPI, and it is the first that is the most valuable. The first method checks that it is valid to build that unit type; that means the necessary resources are available, the necessary research has been performed, and that there exists a unit on your roster that is the progenitor type for the passed type. The second canMake() method checks if a specific unit can make the passed unit type; so it's the correct type and it is not busy with another task.

getLeastBusyUnitofType() is needed as it is common to build multiple production units as certain races to parallelise the tasks they perform.[2] The method checks each unit of the passed unit type and returns the unit that has the smallest production queue, and that isn't researching anything.

Listing 5.2: The current method to produce any unit in the game. IntTriple is a custom type that simply contains three public integers.

```
public boolean produceUnit(UnitType.UnitTypes unitType){
    int unitTypeID = unitType.ordinal();
    if( bwapi.canMake(unitTypeID) ){
        int builderTypeID = bwapi.getUnitType(unitTypeID).getWhatBuildID();

        Unit buildUnit = unitManager.getLeastBusyUnitofType(builderTypeID);

        if (buildUnit != null){
            int buildUnitID = buildUnit.getID();

            if( bwapi.canMake(buildUnitID, unitTypeID) ){
                if(bwapi.getUnitType(unitTypeID).isAddon()){
                    buildQueue.add(new IntTriple(IntTriple.ADDON,
                        buildUnitID, unitTypeID));
                }
                else if(bwapi.getUnitType(builderTypeID).isBuilding()){
                    buildQueue.add(new IntTriple(IntTriple.TRAIN,
                        buildUnitID, unitTypeID));
                }
                else{
                    buildQueue.add(new IntTriple(IntTriple.MORPH,
                        buildUnitID, unitTypeID));
                }

                return true;
            }
        }
    }
    return false;
}
```

5.1.3 Generalising Scouting

This subsection explains how the scouting behaviours of the AI were generalised and the reasoning behind the decisions.

The original scouting code had a single behaviour to start scouting with a worker unit

[2]Production units can only perform a single task at a time, and most tasks take a long time, so it is pretty much a required tactic as the Terran and Protoss.

and it would continually scout with any Overlord units[3] that were produced. Scouts are moved along a predetermined path by sending move commands and waiting until the unit is idle to send another. This was done separately for the worker scout and the Overlord scouts. The aim when generalising was to refactor the two processes together into a single parameterised scout method.

The chosen solution to this is to have a list of the currently scouting units, which can be checked for idleness each tick. A ScoutUnit class was created which along with the unit, kept track of the assigned scouting path, how far along the path the unit is, and a completion handler to be called when the path has been completed. The scouting path is stored as a queue of Point[4] objects. As the the scout reaches each location, the next location is popped from the queue. The completion handler is implemented as an interface with a single method scoutRouteCompleted(), which takes the scouts ID as a parameter (otherwise, the called object won't know which scout finished).

The implementation of this generalisation grew quite large, large enough that scouting took up a large proportion of the intelligence manager. This spurred the decision to move scouting into its own class. The intelligence manager remains the initiator and completion handler for scouts; the scout manager is envisioned as more of an internal utility class. If the scout manager handled scouting completely, it would require race-specific sub-classes, which is to be avoided as much as possible as specified in the requirements.

5.1.4 Generalising the Military

This subsection explains how the military behaviours of the AI were generalised and the reasoning behind the decisions.

Military units were organised into hard-coded lists of units of certain types. So there was a "zergling" list and a "hydralisk[5]" list and so on, but these lists were hard-coded for a small number of types.

The first target of generalisation was focused on these lists. The individual lists have been combined into a map where keys are the unit type id and the value is the list of units. All units that are created are added to these groups (excluding buildings). Units that change into other units are moved between lists. Destroyed units are removed from the lists. This provides a solid base to generalise the class further.

The next target of generalisation looked at how units are controlled. Originally, this was done by iterating through each unit list and sending the chosen command to each unit. The solution first tried was to just copy this functionality in the race-specific sub-classes. Apart from being bad software development practise, this was problematic as it meant controlled units had to be hard-coded in, just like the original code. On the other hand, controlling all units in the same way is undesirable as different unit types often have abilities they can

[3]Overlords are a flying Zerg unit that increase the unit cap. A Zerg player will build lots of these units.
[4]java.awt.Point: Has only two public int variables x and y
[5]Hydralisks are a Zerg ground unit with a ranged attack

use or want to move and attack in a different way.

The implemented solution to this is to use the basic movement and attack behaviour for all units in the super class, while having any specialised behaviour in the race-specific sub-class. A list of "special unit types" is kept which indicates to the default method that it should skip the current unit type and let the sub-methods handle it. This makes it easy to add new units to POSH plans while allowing specialised behaviour where necessary.

5.1.5 Small improvements

This subsection covers small improvements made to the behaviour library.

Build locations

The building manager maintains a list of build locations, which it chooses from when it is time to build a building. The locations were originally generated based on the locations of resources nodes near initial base building. The method needed to modified, but it was too complex. The method has been re-written to be simpler and easier to follow.

Another improvement to build locations is that when one is added, the position is checked to ensure it doesn't conflict with any existing build locations. This is needed for both the Protoss and the Terran as both would duplicate locations as they built buildings which would then create cluttered bases.

Resource reservation

The original resource reservation system simply a integer variable that could be set when a behaviour needed to reserve some resources. Multiple resources reserving at the same time just wasn't feasible because while behaviours could simply increase the reservation instead of setting it, the behaviours are no longer running by the time the task is performed and the resources consumed.

The solution implemented to solve this is to pass an integer ID along with the reservation amount. Then when the resources have been consumed, the reservation with the chosen ID can be cleared. This makes simultaneous reservations easier to deal with, but it doesn't solve the problem of not knowing when to clear the reservation. Reserving resources is only done when building as there is a delay between the command and the resources being spent. Any other time resources are spent, it happen instantly. By using the worker unit ID as the reservation ID, it can be identified when the resources have been spent and which reservation to clear.

Impatience

The original AI had an issue where after it destroyed the main enemy base, it wouldn't search for any extra bases that may have been built. This can result in a loss, as the enemy can rebuild its army. To counter-act this, an "impaitience" timer has been implemented. This is a timer that waits for a certain number of game ticks, then changes the attack destination to a different base location[6]. The timer is reset upon sighting an enemy. The effect of this it that if the army successfully destroys the enemies main base, it will cycle around the map until it finds the remnant force.

Expansion Locations

A problem in the AI identified by Davies (2012) was that possible expansion locations were calculated on straight line distance, instead of walking distance. So bases that are close by but take a long time to walk to would be preferred over more accessible options. This couldn't be fixed at the time as JNIBWAPI did not provide the necessary functions. The updated JNIBWAPI does provide the necessary functions, so the method has been updated to make better choices.

5.2 POSH Plans

This is the chapter in which you review the implementation and testing decisions and issues, and critique these processes.

Code can be output inline using `\lstinline|some code|`. For example, this code is inline: `public static int example = 0;` (I have used the character `|` as a delimiter, but any non-reserved character not in the code text can be used.)

Code snippets can be output using the `\begin{lstlisting} ... \end{lstlisting}` environment with the code given in the environment. For example, consider listing 5.3, below.

Listing 5.3: Example code

```
public static void main() {

  System.out.println("Hello World");

}
```

Code listings are produced using the package "Listings". This has many useful options, so have a look at the package documentation for further ideas.

[6] Most, if not all, StarCraft maps have a limited number of viable base locations around clusters of resource nodes.

Chapter 6

Results

This is the chapter in which you review the outcomes, and critique the outcomes process. You may include user evaluation here too.

Chapter 7

Conclusions

This is the chapter in which you review the major achievements in the light of your original objectives, critique the process, critique your own learning and identify possible future work.

Bibliography

A StarCraft AI bot (n.d.), https://code.google.com/p/bthai/. Retrieved November 22, 2013.

Berkeley Overmind (n.d.), http://overmind.cs.berkeley.edu/#desc. Retrieved November 22, 2013.

Blizzard Entertainment – StarCraft (n.d.), http://us.blizzard.com/en-us/games/sc/. Retrieved November 21, 2013.

Bryson, J. J. (2001), 'Intelligence by design: principles of modularity and coordination for engineering complex adaptive agents'.

Bryson, J. J. (2003), 'The behavior-oriented design of modular agent intelligence', *Agent technologies, infrastructures, tools, and applications for e-services* pp. 61–76.

Bryson, J. J. (2005), 'The behavior-oriented design of modular agent intelligence', http://www.cs.bath.ac.uk/~jjb/web/BOD/AgeS02/node13.html. Retrieved March 30, 2014.

BWAPI (n.d.), https://code.google.com/p/bwapi/. Retrieved November 21, 2013.

BWAPI - Competitions (n.d.), https://code.google.com/p/bwapi/wiki/Competitions. Retrieved November 21, 2013.

Certicky, M. and Certicky, M. (2013), Case-based reasoning for army compositions in real-time strategy games, *in* 'Scientific Conference of Young Researchers 2013', FEI, Technical University of Košice, pp. 70–73.

Dainotti, A., Pescape, A. and Ventre, G. (2005), A packet-level traffic model of starcraft, *in* 'Hot Topics in Peer-to-Peer Systems, 2005. HOT-P2P 2005. Second International Workshop on', IEEE, pp. 33–42.

Davies, S. (2012), Development of an ai for the real-time-strategy game starcraft using behaviour oriented design, Master's thesis, University of Bath.

Galloway, A. R. (2007), 'Starcraft, or, balance', *Grey Room* **28**, 86–107.

Gaudl, S., Davies, S. and Bryson, J. J. (n.d.), 'Behaviour oriented design for real-time-strategy games'.

Hostetler, J., Dereszynski, E. W., Dietterich, T. G. and Fern, A. (2012), 'Inferring strategies from limited reconnaissance in real-time strategy games', *arXiv preprint arXiv:1210.4880* .

Huang, H. (2011), 'Skynet meets the swarm', `http://arstechnica.com/gaming/2011/01/skynet-meets-the-swarm-how-the-berkeley-overmind-won-the-2010-starcraft-ai-compet` Retrieved November 22, 2013.

JNIBWAPI (n.d.), `https://code.google.com/p/jnibwapi/`. Retrieved November 13, 2013.

Kurzweil, R. (2005), 'The singularity is near: When humans transcend biology'.

McCorduck, P. (2004), 'Machines who think: A personal inquiry into the history and prospects of artificial intelligence, ak peters', *Natick, Mass* .

Monte-Carlo method (n.d.), `http://www.encyclopediaofmath.org/index.php/Monte-Carlo_method`. Retrieved November 22, 2013.

Rathe, E. A. and Svendsen, J. B. (2012), 'Micromanagement in starcraft using potential fields tuned with a multi-objective genetic algorithm'.

Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M. and Edwards, D. D. (1995), 'Artificial intelligence: a modern approach'.

Safadi, F. and Ernst, D. (2010), 'Organization in ai design for real-time strategy games'.

StarCraft Replays (n.d.), `http://www.teamliquid.net/replay/`. Retrieved November 22, 2013.

StarCraft Wiki (n.d.), `http://starcraft.wikia.com/wiki/StarCraft_Wiki`. Retrieved November 21, 2013.

Synnaeve, G. and Bessiere, P. (2011), A bayesian model for rts units control applied to starcraft, *in* 'Computational Intelligence and Games (CIG), 2011 IEEE Conference on', IEEE, pp. 190–196.

Synnaeve, G. and Bessiere, P. (2012), A bayesian tactician, *in* 'Proceedings of the Computer Games Workshop at the European Conference of Artificial Intelligence 2012'.

Team Liquid (n.d.), `http://www.teamliquid.net/`. Retrieved November 21, 2013.

Team Liquid, Micro (n.d.), `http://wiki.teamliquid.net/starcraft2/Micro_(StarCraft)`. Retrieved March 27, 2014.

Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelback, J. and Yannakakis, G. N. (2010), Multiobjective exploration of the starcraft map space, *in* 'Computational Intelligence and Games (CIG), 2010 IEEE Symposium on', IEEE, pp. 265–272.

Wang, Z., Nguyen, K. Q., Thawonmas, R. and Rinaldo, F. (2012), Monte-carlo planning for unit control in starcraft, *in* 'Consumer Electronics (GCCE), 2012 IEEE 1st Global Conference on', IEEE, pp. 263–264.

Weber, B. G. and Mateas, M. (2009), A data mining approach to strategy prediction, *in* 'Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on', IEEE, pp. 140–147.

Wooldridge, M. and Jennings, N. R. (1995), 'Intelligent agents: Theory and practice', *The knowledge engineering review* **10**(02), 115–152.

Yi, S. (2011), 'Adaptive strategy decision mechanism for starcraft ai', *EKC 2010* pp. 47–57.

Appendix A

Design Diagrams

Appendix B

User Documentation

Appendix C

Raw results output

Appendix D

Code

D.1 File: yourCodeFile.java

```java
// This is an example java code file, just for
    illustration purposes

public static void main() {

  System.out.print ("Hello World");

}
```