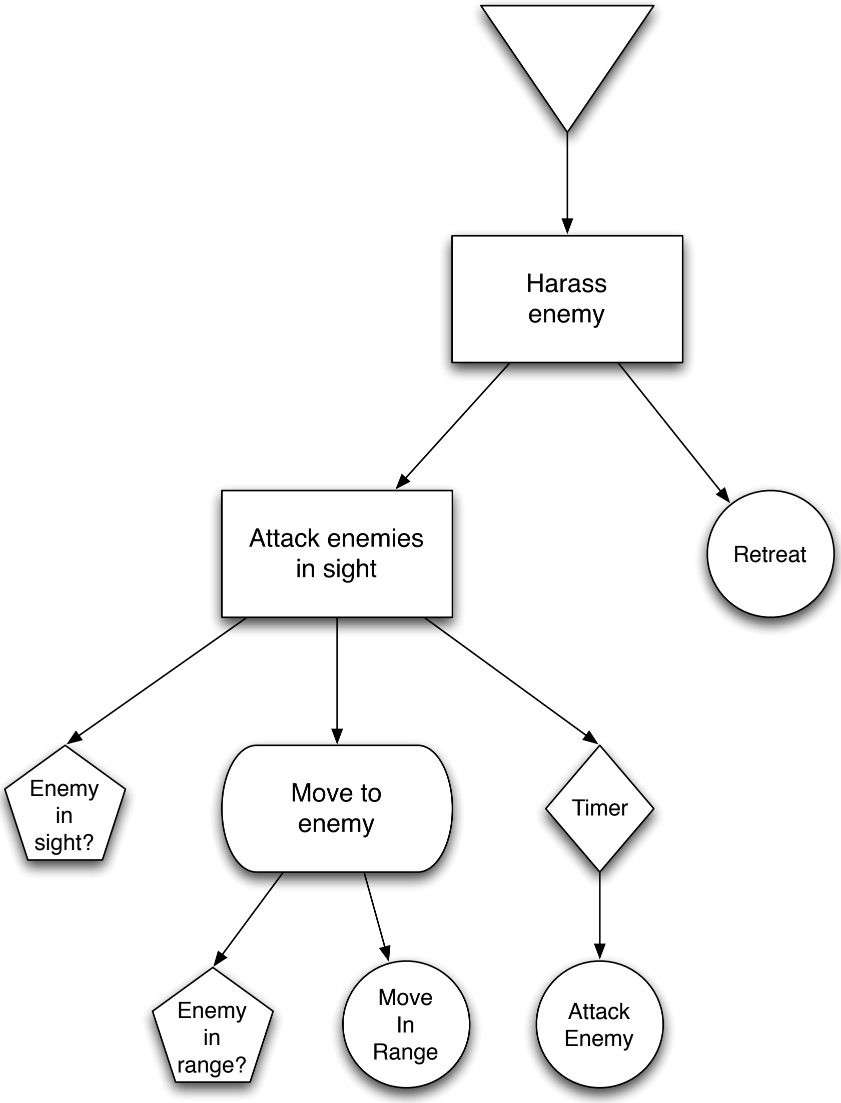
Applying Behavior Trees to StarCraft AI

DAT5, FALL SEMESTER 2010

GROUP D518A

DEPARTMENT OF COMPUTER SCIENCE

AALBORG UNIVERSITY
22ND OF DECEMBER 2010



Title: Applying Behavior Trees to StarCraft AI

Theme: Artificial Intelligence in RTS games

Project period: Dat5, fall semester 2010

Project group: d518a

Group members:

____
Søren Larsen

____
Jonas Groth

____
Kenneth Sejrsgaard-Jacobsen

____
Torkil Olsen

____
Long Huy Phan

Supervisor: Yifeng Zeng

Abstract:

We investigate the area of game AI with focus on the video game genre of Real-Time Strategy using StarCraft as test platform. We proceed to discuss different AI methods that have an application in RTS games. We choose to focus on behavior trees and present our implementation of a behavior tree framework, for implementing behavior trees, and an editor for designing them. We test the framework by implementing a set of behaviors described in the report and finally conclude on the work done by proposing new possibilities for further development.

Copies: 7

Number of pages: 70

Appendices: 2

Completion date: 22nd of December, 2010

Additional material available on the attached CD.

Preface

In this report we discuss the aspects of constructing an artificial intelligence (AI) for a computer game that can provide the human opponent with an entertaining challenge, rather than showing off super-human intelligence and decision making. Several aspects are important for this area, such as the techniques used to create the AI, the performance of the AI, the computer game genre the AI is intended for and so forth.

We will describe some of the game mechanics of the selected game, StarCraft, but the reader is required to have some basic knowledge of StarCraft such as the units and structures etc.

In Chapter 1 we discuss the different types of video game genres and select a genre that we want to further investigate. We proceed to discuss key aspects of this genre and select a game for which we want to develop an AI for. Chapter 2 gives a more in depth description of StarCraft and the game mechanics. We also define the scenario we will be using in the rest of the report.

In Chapter 3 we introduce different methods for creating game AI. We also briefly comment on other AI methods, such as learning and planning. We end the chapter by summarizing the discussed methods and select one method for further use in the report.

Chapter 4 describes the framework and editor, for designing and implementing behavior trees, developed as part of the report. In Chapter 5 we propose a set of behavior trees that make up a simple StarCraft AI. The behavior trees are implemented in the designed framework and tested in Chapter 6. Finally we conclude our work and propose possibilities for further work in Chapter 7 and Chapter 8.

The framework and the editor are available for testing on the CD enclosed with this report or by request from the authors.

We would like to thank deathknight13579 and lowerlogic for providing the BWAPI framework used for implementing AI's in StarCraft and dpershouse for providing bwapi-mono-bridge and responding to issues with his implementation.

Contents

1 Introduction

Video games offer a great platform for AI development. Depending on the genre of video game it may vary which method to use when designing the AI.

Before going into details in a specific game and AI methods we will give an introduction to the video game domain by describing and discussing the genres video games are generally categorized by. Then we will narrow it down to a more specific branch in the strategy game genre and briefly introduce the game chosen in this project.

1.1 Video Games

There exist a tremendous amount of video games. Each of these games can be put into one category or a hybrid of multiple categories. The categories have different game aspects in focus, for instance adventure games rely a lot on the story while action games rely more on fast paced action. In this section we will present the most commonly known genres in video games and discuss their key features.

Action

Action games are typically characterized by being fast paced and require the player to have quick reflexes and good accuracy to win the game. The action game category has several sub-categories such as beat-em ups and shooters. Both sub-categories are action games but provide the player with very different experiences.

In beat-em up's the object is to fight your opponent(s), typically in an arena using melee weapons or hand-to-hand combat. The first player that loses all his or her hit points loses the game. Amongst widely known titles in this sub-genre are *Tekken* [17], *Mortal Combat* [15] and *Dead or Alive* [24].

Shooters are different in the sense that you take the role of a character that needs to eliminate a lot of enemies while finding some path to a goal location or complete a mission. These games often offer a huge arsenal of weapons ranging from melee weapons to ranged weapons like pistols, machine guns and rocket launchers. The most popular ways of interacting with shooter games are by first or third person gaming. In first-person shooters your view is defined by where the avatar is looking. In third-person shooters your view is defined as a point somewhere behind the avatar effectively looking over his or her shoulder.

Role-Playing

Role-playing games are often based on one or more avatars who through questing, combat, puzzle solving, monster slaying etc. can gain experience and increase in level. At each level the avatars can improve their skills usually selected by the player. The primary source of entertainment for the player is seeing and affecting the way the avatar(s) progress through the game as well as being immersed in the storyline.

Many role-playing games incorporate some kind of adventure game aspects, which will be explained below, and a story or primary mission for the player to complete in order to complete the game. Role-playing games often implement some notion of good vs. bad for example in the form of karma. Karma determines how *non-playable characters* (NPCs) in the game perceive the avatar(s). This forces the player to give decisions additional consideration as every decision has an impact on the rest of the game.

Adventure

Adventure games primarily present the player with riddles and puzzles that the player has to solve in order to progress in the game. There is always a main goal or storyline in the game that is reached through successfully solving the riddles and puzzles.

An old but still popular example of adventure game, are the games of the Monkey Island series [11]. In this series the player is presented with one major goal at the beginning of the game and then through smaller chapters, sub goals are completed, in order to reach the final goal. A significant part of this specific series is the oddness of the riddles. The player will often be presented with very illogical problems requiring illogical solutions.

Strategy

The strategy genre revolves around the player making strategic and tactical decisions in order to win a game. The core of most strategy games is warfare and the strategic and tactical decisions revolve around the player training the correct units to counter the opponent units while seizing control of the map.

Some strategy games revolve around a more global and realistic scenario. Here the player can be presented with both military, political, economic and research problems that need to be solved in order to complete the goals of the game.

Simulation

The simulation genre is based on making a game as realistic as possible. Probably the commonly best known type of simulation games are the flight simulators. These games are designed to teach the player how to pilot a plane by having realistic cockpits and flight conditions, and letting the player take off, maneuver and land the plane on the ground again.

Some simulation games revolve around making some business or company run and grow. This would require the player to make decisions about expanding operations and making use of supply and demand to increase revenue of the players company.

Summary

We have decided to focus on a game from the strategy genre. The primary reason for this is the obvious application area for an AI. Strategy games rely on strategic and tactical decision which often require some clever reasoning. We have also chosen the strategy genre because the remaining genres do not offer the same possibilities in AI development as the strategy genre. For example in adventure games there is no real need for an AI to act as the player as the game is most likely predetermined.

1.2 Strategy games

For years, the genre of strategy video games has been popular among gamers all over the world. The reason for this may very well be the fact that strategy games encourage the use of strategic and tactical thinking. However there is a major difference between some games in the genre that has lead to the definition of two subsets of strategy games, namely *Real-Time Strategy* (RTS) and *Turn-Based Strategy* (TBS).

1.2.1 Turn-Based Strategy

The most defining characteristic in Turn-Based Strategy (TBS) games is the fact that each round each player has one turn to perform his or her actions. Actions can consist of moving or attacking with military units, selecting new research, setting up a building queue for a town etc. When all players have finished their turn the round ends and a new round begins. As one would imagine TBS games can be long running. The turn-based game play also has an impact on the creation of an artificial intelligence to provide an entertaining challenge for a human player. This, due to the fact, that when the AI would get its turn and that the length of the turn being, theoretically unlimited, the AI could perform more complex calculations and use more complex models for the game to determine the course of action.

Many people know the series of TBS games, *Civilization* [14] created by Sid Meier. This series features turn-based game play with a strong AI and a lot of features that need to be used strategically to win the game. These features include diplomacy, keeping your civilization happy and researching new technology etc.

1.2.2 Real-Time Strategy

Opposed to TBS games are Real-Time Strategy (RTS) games. These games are more action-oriented and tactical and strategic decisions need to be made on the fly while the opponent(s) may be attacking you. Most RTS games have relatively short game time compared to TBS games, but pride themselves on being more realistic due to the fast paced action packed nature of the games. This real-time fast paced game play puts an obvious limit on the AI that can be used in the game. The AI cannot be too complex as the calculations and models need to be computationally light, or else the AI will not be able to make decisions in due time as actions become obsolete if not executed when required. The challenge with RTS AI is creating an intelligent solution that does provide an entertaining challenge to the human player and uses relatively few resources.

1.3 Video Game Environment for This Project

A classic in the RTS genre, and the focus of this project, is StarCraft: Brood war [2]. This game features a fast paced action game of resource management and strategic decision making a long with tactical management of military units.

Having been on the market over a decade, with a huge community sharing their thoughts and opinions of the game with the Blizzard Entertainment development team, has resulted in a stream of updates and patches to tailor the game to be very balanced. This means that out of the three playable races, there is no one more powerful than the others.

StarCraft: Brood war is a complex game to play. It involves actions being performed at multiple scales as players have to make economic, strategic and tactical decisions alongside reacting to the opponent. All this is done on the fly due to the game being an RTS game.

With the game having fog of war and thus being partially observable, it often comes down to mind games. This means that it is possible to keep your opponent in the dark while you execute some strategy and surprise attack him, but if not done with care or not having eyes open on the opponent, he can surprise you or spot your strategy and make it obsolete.

The *Brood War Application Programming Interface* (BWAPI) enables the development of custom AIs for StarCraft: Brood war with relative ease. The BWAPI will be further explained in Section 2.5.

All this combined along with our own interest in the game makes StarCraft: Brood war the obvious choice of game and a very interesting environment for developing an AI and studying it. We will go into more details on StarCraft: Brood war in Chapter 2.

1.4 Project Purpose

The main goals of this project are as follows:

- Give an in depth description of StarCraft: Brood War to pinpoint the challenges in developing an AI for the game and create a game scenario to be used for case studies.
- Discuss viable AI methods for creating RTS game AI and present a survey of applying the different AI methods to StarCraft: Brood War. With the purpose of finding the pros and cons of the different methods.
- Present possibilities for future work and improvements.

2 StarCraft: Brood War

StarCraft: Brood War is a real-time strategy game developed by Blizzard Entertainment. The game was released in 1998 and later that year Blizzard Entertainment released StarCraft: Brood War, an expansion to the game which brought several game play improvements including new units, maps, campaigns and balancing changes. From this point forward will we refer to StarCraft: Brood War as StarCraft.

The Setting

StarCraft is set in a science fiction universe created by Blizzard Entertainment. The location is the distant Koprulu Sector of the galaxy where three powerful races fight for supremacy and survival.

StarCraft offers three playable races. The Terran, human outlaws exiled from earth. Protoss, the proud and highly intelligent race of their home planet Aiur. Finally the Zerg, monstrous creatures who seek only to destroy.

Every unit in StarCraft is unique to its respective race. The technologically advanced Protoss are known for their expensive but strong units. They rely on the quality of each unit over their numbers. The Zerg on the other hand have inexpensive but rather weak units. Their power lie in their numbers. The Terran units are more balanced regarding price and strength.

2.1 Game Aspects

The competencies for succeeding in the game are often categorized into macro management and micro management. Macro management includes the big decisions, like which initial strategy to use, which units to get, when to attack, moving armies across the map etc. Micro management is when individual units are given specific orders. The major aspects of StarCraft will be described in the following sections.

Resource Management

Resources are needed in order to construct buildings, train units and research upgrades. In StarCraft there are two kind of resources, minerals and vespene gas. Minerals can be mined right of the mineral patches by workers but in order to get vespene gas a gas extractor has to be built on the vespene geysers which workers then can extract gas from. Throughout the game players need to sustain a consistent income to ensure unit training, research and base construction when needed. The workers gathering resources are the lifeline of each players progress in the game. Therefore, it is critical to protect the resource line from enemy attacks, as the loss of even a few workers will cripple the economy and result in a major set back in training military units.

Base Construction

Base construction includes which building to construct and where to place them. Which buildings chosen affects which units are possible to train and the placement of building is also important as it is possible to block some entry point to your base. The three races all have diverse and unique buildings and way of construction.

The Zerg base appears as one living organism and the buildings are able to heal themselves slowly over time. Some buildings spread a carpet of biomass called creep and most new buildings may only be built on the creep. The Zerg workers sacrifice themselves in the process of building structures as they do not construct the structures but rather evolve into buildings themselves.

Protoss buildings are not constructed but warped in from the Protoss home planet. The workers only need to start the warp in and are then free to perform other commands. The majority of Protoss buildings require psi power in order to be warped in. The building known as a pylon emits psi power around itself. In this radius new buildings can be warped in and the finished buildings need the psi power to continue operation.

Unlike the other two races, the Terran have no restriction to where they can place buildings. Some of the Terran buildings can lift off and fly to a new location. When constructing a building one worker attends the construction site until it is finished and is vulnerable to attacks. A weakness of the Terran construction is when a building takes a certain amount of damage it will catch fire and if not repaired by a worker it will eventually burn to the ground.

Scouting

The partial observability of the game makes scouting and map awareness an essential part of the game. The players only see what their own units and buildings have vision of, so in order to see what the opponent is planning they have to send some units to do reconnaissance. It is important to scout to see what your opponent is doing so that you can take counter measures. For instance if your opponent is training flying units or units capable of cloaking, you need some anti-air units or detector units. Scouting is also the only source of information gain in the game. By scouting your opponent early in the game you can expose his strategies and if he for example is doing a rush strategy, where he attacks very early, appropriate countermeasures can be taken.

Combat

Combat in StarCraft includes several aspects. They range from selecting appropriate units to counter your opponent's units, timing of attacks and controlling individual units in combat. The controlling of individual units is often referred to as micro management. One specific technique called *dancing* is when the player orders a unit to attack, while the weapon is reloading the unit moves back to avoid damage and then in again to shoot.

2.2 Game Mechanics

All aspects of the game are cut in stone and there are close to no random factors. This means that the outcome of a game depends solely on the players playing it and their skill. For instance the outcome of a battle between two equal forces of military units depends only on how the players control their units and not on any varying damage or probability factors. Other RTS games have chance elements like ranging damage, critical hits (probability of increased damage), evasion and dodge (chance of avoiding attacks) and also a unit armor decrease incoming damage by some percentage. None of these elements exist in StarCraft. For example the marine in StarCraft has a starting damage of six. This can be upgraded three times giving +one damage each time. Said unit will always deal this damage, no more no less. The unit taking the damage may have some armor value. Say it has a armor value of two, then all incoming damage will be decreased by two leaving a non-upgraded marine deal four damage each shot. There is although one thing that has an effect on damage in StarCraft and this is the size of a unit. Units in StarCraft are either small, medium or large. Building are considered large. The size of a unit is taken into account when calculating the effects of different classes of damage. The damage classes and their effectiveness in relation to the three unit sizes are as follows:

- Normal weapons are equally effective against all types of units.
- Concussion/Plasma weapons do 50% damage against Medium units and 25% against Large units.
- Explosive weapons do 50% damage against Small units and 75% against Medium units.

The Scenario used in this project will only include one type of unit, which is of small size, and hence always does full damage. These damage calculations will therefore not be discussed further.

Although not existing as unit abilities, StarCraft has the terrain determined property of a unit being on high ground. Units attacking enemy units who are on higher ground have a chance to miss on attack. As our scenario will not allow units being on high ground this element is considered not relevant for the time being.

This section is based on the information from the StarCraft Compendium [4].

2.3 Full Game Scenario

A full game scenario of StarCraft provides all players with a base and a few workers at the start of the game. Players build and expand their base, train units and ultimately eliminate all opposing players. The players keep in mind all the aforementioned elements as the game progresses. They plan their strategies and switch to new strategies if they see the opponent do something unanticipated which affects the current strategy.

Challenges

To summarize, the challenges in playing a game of StarCraft are many. First of all an initial strategy must be chosen. This strategy will be executed until successfully defeating the opponent or be replaced by a new strategy if your opponent's strategy is resistant to your current strategy. Furthermore you have to scout your opponent to see what he is doing and ideally keep some pressure on him by attacking his resource gatherers. All this, alongside defending yourself from your opponent doing exactly the same to you, poses several interesting challenges.

2.4 Scenario

The scenario that we will use as a test bed for our approach is a simplified subset of a full game of StarCraft. The goal is to capture the important aspect of a full game and still keep it as simple as possible.

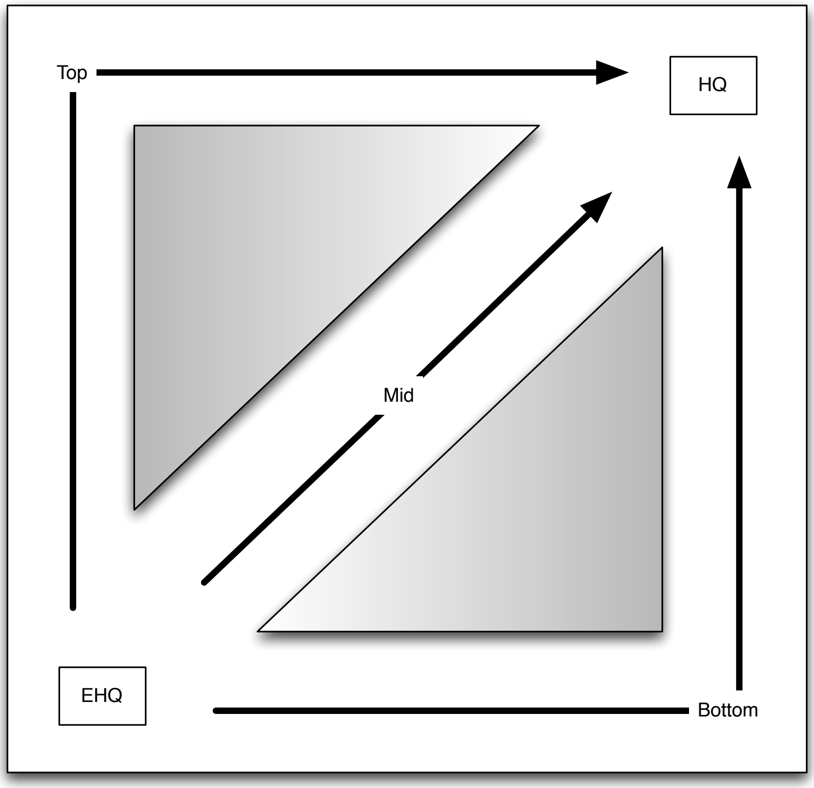


Figure 2.1: A map illustrating our scenario. There are three attack routes, some obstacles and two HQs.

Our scenario, which is depicted in Figure 2.1, consists of a small squared map with two players. Each player will have a HQ in opposite corners of the map, that is at the bottom left and top right of the map. Every ten seconds a marine will spawn close to the HQ. The map is designed in such a way that there are three attack paths, top, mid and bottom. The goal of the players is to destroy their opponent's HQ and by doing this they win the game. The purpose of having units spawn at time intervals rather than having a building training units at the cost of resources is for simplification of the scenario. By doing this we can disregard the base building and resource management aspect of the game and focus solely strategic and tactical decisions. The map design with only three paths also adds to the simplicity of the scenario.

The challenges to face in this scenario are the following:

- Should one launch an attack on one path with full force or split up the army and attacks several fronts

- Scout the opponent for information and decide how many units to risk for this information
- Decide when to retreat in order to defend the HQ or just continue the an ongoing attack
- Predict what the opponent will do based on the gathered information
- Should one send some units to harass the enemy and ideally take out some units

2.5 Brood War Application Programming Interface

For the implementation we will be using Brood War Application Programming Interface (BWAPI) [6]. BWAPI is a free open source C++ framework for developing custom AIs for StarCraft: Brood War. BWAPI uses DLL injection The required software for using the BWAPI is StarCraft Brood War 1.16.1, an Integrated Development Environment (IDE) appropriate for the chosen programming language, Microsoft Windows and due to BWAPI using DLL injections a loader is needed to enable the BWAPI injections to be loaded into the StarCraft process. Although BWAPI being an interface written in C++ there exist a wide range of wrappers and proxy bot implementation. With proxy bot the AI bot is running in a remote process and communicates with the AI module on top of the BWAPI through sockets. Wrappers and proxy bot allow the programmers to write AI behavior in different programming languages than C++.

With BWAPI it is possible to query the game state for relevant information on every game frame. Methods for retrieving game information include methods being queried on individual units to get their type, position, health status, upgrades, etc. and methods for querying players in the game for information like amount of resources, get a list of all their units, supplies used, etc. Furthermore there are methods for retrieving information on the specific map being played. For instance mineral fields, vespene geysers, etc. Supplementing the built in terrain methods in the BWAPI there exists an add-on for BWAPI called Brood War Terrain Analyzer (BWTA) which analyzes the map being played and returns all possible base locations, choke points and regions.

In addition to methods for retrieving game information there are methods in the BWAPI for issuing orders to specific units. These orders include move, attack, build and train commands as well as using unit abilities etc. There are also methods for querying specific units for their current status for instance one can check whether a unit is a worker with the isWorker method. Furthermore the methods isIdle, isMoving, isGatheringMinerals, isCarryingMinerals, isAttacking, isBeingConstructed, which are self explanatory, grant the programmer great control over the units and their status.

Every unit and entity in the game has a position on the map specified as an x and y coordinate. A map of size 64x64 in StarCraft consists of 64 position tiles time 64 position tiles. Each position tile consists of 32x32 positions and these positions define the x and y coordinates of units. Tile positions and positions are used depending on the orders to issue, for instance when constructing buildings tile positions are used and positions can be used to move units. This fine resolution of the map gives a fine grained control of units.

BWAPI works by triggering methods on certain events in the game. Also there is a method that is fired on every frame of the game. Examples of game events are onStart, onEnd, onSendText, onUnitShow, onUnitHide, onUnitCreate and onUnitDestroy. OnSendText can be used by the programmer to trigger some AI behavior or set some game options, like game speed or cheat flags, while the game is running. In BWAPI there are two cheat flags accessible, enable full map information and enable user input. These are great for debugging purposes, but are of course not allowed in official games. If full map information is enabled then all units on the map are fully accessible to the BWAPI meaning that all information on the unit are accessible. With the flag disabled then units owned by the BWAPI player remain fully accessible, but the enemy units fall in the categories normal, partial and no accessibility. Normal accessibility is when a unit is detected and visible. The information available includes all of the attributes in full accessibility except some specific attributes like what a building is producing, how much time left on the production or the quantity of a units items etc. Partial accessibility is for cloaked units that are visible, but not detected. The onUnitHide and onUnitShow events are triggered when units transition between partial and no accessibility. Units hidden by the fog of war or dead units fall in the category no accessibility.

We will be using the bwapi-mono-bridge for implementing the AI [7]. It is a wrapper implementation that exposes the BWAPI functionality to the .NET framework through Mono. This makes it possible to do the implementation in a .NET language of our choice which is C#. Bwapi-mono-bridge implements two solutions, one that embeds the AI bot in the same process as StarCraft and one that runs in a separate process and communicates as client/server. We will be using the client/server solution in order separate the AI and game execution. When run in the same process space they can interfere with one another for instance if there is a computationally expensive task in the AI, it can slow down the game if run in same process.

3 AI Methods

AI is quickly becoming a larger part of computer game development today. With graphics approaching photorealism, the next big improvement is the AI experience. However, there are several issues one must deal with when developing game AI. First off, games today are often very complex with respect to the size of the game world and the realism demanded of NPCs. If one were to consider the entire state space of a modern game, including the different actions and positions of the NPCs, it would be impossible to apply any of the traditional academic AI methods.

To address this problem, developers will often have to simplify the state space of the game. However, this often results in a simplified AI which could reduce the game experience, as players today demand even more realistic behavior from the AI. The representation of the game must thus be simple enough for the AI to process it in real-time, and still complex enough to produce a believable behavior.

Another problem one must address is the scope of game AI. When working with academic AI one will often try to reach a level where the AI can be considered super-human. That is, it is superior to a human with respect to the particular problem for which it was developed. In game AI we never want to reach a super-human level of AI. An AI who can defeat the human player every time will drastically reduce the game experience to the point where the player forfeits. As a game developer it is in ones interest to create an AI which poses a certain challenge for the player, but is still capable of mistakes resulting in defeat.

There are several different approaches to game AI capable of dealing with the above mentioned problems to some extent. Two of these approaches are scripting and *Finite State Machines*(FSM). Each of these approaches have their own pros and cons and depending on the game scenario it has been up to the programmers and designers of the game AI to find a compromise.

In the next sections we will describe the general principles of these two approaches to game AI, namely scripting and FSMs, and a third approach *Behavior Trees*(BT). The methods will be described, discussed and evaluated in relation to the scenario proposed in Section 2.4. A discussion of different AI techniques such as learning and planning will also be included, to make grounds for further improvements of the methods evaluated.

3.1 Scripting

When developing AI for computer games the speed of execution is of considerable importance. Depending on the type of game an AI should typically make decisions within a time window of a few game frames. It is therefore of no use developing sophisticated and complex AIs if they are not capable of executing in real-time. Another factor is the amount of work it takes for a development team to develop an AI. Typically the AI is a very complicated part of the game and may take months of developing time. Furthermore, due to the complexity of AIs, it often demands the attention of skilled programmers; programmers who are already working on many other aspects of the game.

For this reason, it is in every game company's interest to reduce this development time and ease the strain on their programmers. The best way to achieve this would be to put some of the work on other parts of the development team, mainly game designers. To do, this the process of developing AIs would have to be simplified.

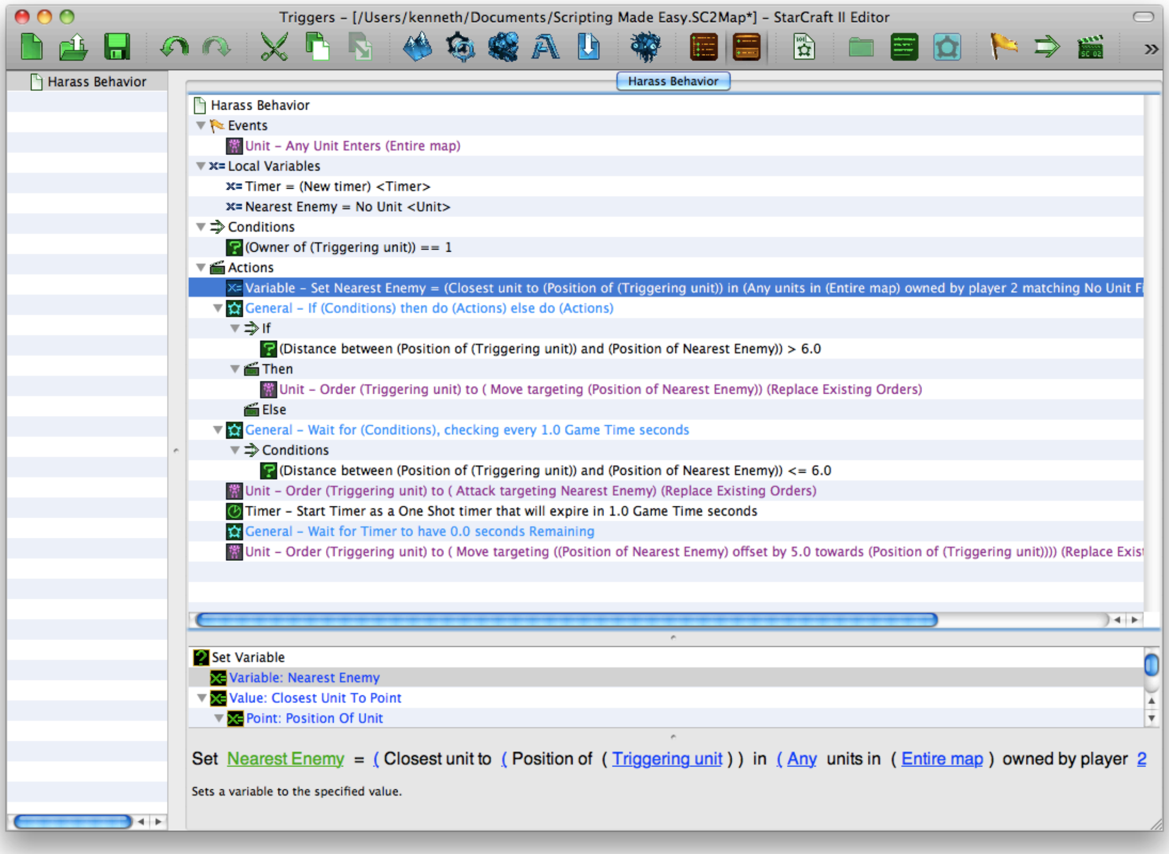
It is due to these demands that scripting has become the most popular method for developing computer game AI. While being very efficient compared to advanced AI methods, it is also quite easy to learn. Usually scripting is done in a scripting language, which is always very high level and thus accessible. The high level constructs also reduces the time it takes to write scripts, as more instructions take up fewer lines compared to lower level languages.

It has also become quite popular for game companies to develop their own scripting languages for their games. By doing this they can ensure that it only contains the constructs and functions relevant for a particular game. Often, such languages are accompanied by tools providing GUI-based editors for the language. One such tool is Blizzard's StarCraft 2 Editor [3], the world editor for StarCraft 2. This editor implements Galaxy, a scripting language developed by Blizzard Entertainment, and makes scripting easily accessible for everyone, even the end users.

Example

This section will give an example of scripting applied to a small part of our StarCraft scenario. Although we could have used the editor of the original StarCraft game, we feel that this would result in an unfair assessment of scripting, since that editor is more than a decade old. Instead, we will show how scripting can done today, using the StarCraft 2 Editor to demonstrate the script of a similar scenario.

Figure 3.1 illustrates a harassing behavior of a single unit, created in the StarCraft 2 Editor's trigger editor, and show how scripting can be done without the need to manually write the code.



Triggers – [/Users/kenneth/Documents/Scripting Made Easy.SC2Map*] – StarCraft II Editor

```
Harass Behavior
▼ Events
    Unit – Any Unit Enters (Entire map)
▼ Local Variables
    Timer = (New timer) <Timer>
    Nearest Enemy = No Unit <Unit>
▼ Conditions
    (Owner of (Triggering unit)) == 1
▼ Actions
    Variable – Set Nearest Enemy = (Closest unit to (Position of (Triggering unit)) in (Any units in (Entire map) owned by player 2 matching No Unit F
  ▼ General – If (Conditions) then do (Actions) else do (Actions)
    ▼ If
        (Distance between (Position of (Triggering unit)) and (Position of Nearest Enemy)) > 6.0
    ▼ Then
        Unit – Order (Triggering unit) to ( Move targeting (Position of Nearest Enemy)) (Replace Existing Orders)
      Else
  ▼ General – Wait for (Conditions), checking every 1.0 Game Time seconds
    ▼ Conditions
        (Distance between (Position of (Triggering unit)) and (Position of Nearest Enemy)) <= 6.0
    Unit – Order (Triggering unit) to ( Attack targeting Nearest Enemy) (Replace Existing Orders)
    Timer – Start Timer as a One Shot timer that will expire in 1.0 Game Time seconds
    General – Wait for Timer to have 0.0 seconds Remaining
    Unit – Order (Triggering unit) to ( Move targeting ((Position of Nearest Enemy) offset by 5.0 towards (Position of (Triggering unit)))) (Replace Exis
```

```
Set Variable
  Variable: Nearest Enemy
▼ Value: Closest Unit To Point
  ▼ Point: Position Of Unit
```

Set Nearest Enemy = (Closest unit to (Position of (Triggering unit)) in (Any units in (Entire map) owned by player 2

Sets a variable to the specified value.

Figure 3.1: The StarCraft 2 Editor's trigger editor

A behavior similar to the one in Figure 3.1 is shown in Listing 3.1 just to illustrate how the code of a written script could look like. Parts of the code have been omitted and replaced with comments to reduce the number of lines of code.

```
def HarassEnemy()

  Unit nearestEnemy = getNearestEnemy()

  while nearestEnemy.getDistance() > 6 do
    # move to nearest enemy
  end

  if nearestEnemy.getDistance() <= 6 then
    # attack enemy for one second
  end

  # move away from enemy
end
```

Listing 3.1: Guard behavior on unit sighted event.

Assume that both of these scripts are set to execute when the unit is created on the map. First it will find the nearest enemy unit and, if outside its range, start moving towards it. At every given interval the script will check whether the enemy unit is within range of its weapon. When this condition is satisfied it will attack the enemy unit for one second and then start moving away from it. Both of these scripts do roughly the same, but what is important to note, is that every variable, condition and action used in the script in Figure 3.1, have been created just by clicking through the various options offered by the GUI.

With tools such as this, scripting has been simplified to the point where programmers need only to create the tool. Afterwards the playable parts of the game can be created in the editor without help from programmers. Furthermore, scripts such as these are interpreted by the game engine and do not need to be compiled with the game. This allow for developers to change the scripts and view the changes without re-compiling the entire game. It also allows end users to write their own scripts for the game once released and create their own AI.

3.1.1 Summary

Scripts have been and still are the more popular approach to AI development, and with good reason. Much effort is put into developing scripting languages and tools to assist developers and designers creating even more realistic AI in shorter time. Especially tools such as the StarCraft 2 Editor show just how powerful and efficient scripting can be, and that the use of the method is far from declining. Other than the ease of use, scripts are also an efficient methods when it comes to computation time compared to advanced AI techniques. However, scripting has a limited functionality, as the complexity of scripts quickly increase with the complexity of the behavior. This can result in scripts becoming incomprehensible to game designers who may not have the necessary programming experience.

3.2 Finite State Machines

FMSs are regarded as one of the simplest approaches to NPC behavior in games. FSMs have been widely applied especially to first-person shooters [1], but also other game genres such as RPGs and RTS games [16]. The reason for FSMs being so popular is their ease of use as humans find the concept of ’being in a state’ comprehensible. With FSMs game designers can design the behavior of NPCs with close to no knowledge about programming.

In this section we will discuss the structure of FSMs and *Hierarchical Finite State Machines* (HFSMs). We will then show an example of how HFSMs can be applied to a small part of the StarCraft scenario. Concluding the section, the pros and cons of FSMs compared to other methods will be discussed.

3.2.1 Structure

The definition of an FSM can be seen in Definition 3.1.

Definition 3.1 – Finite State Machine
An FSM can be defined as a quintuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite, non-empty set of states
- Σ is a finite set of inputs
- $\delta : Q \times \Sigma \to Q$ is the state-transition function
- q_0 is an initial state s.t. $q_0 \in Q$
- $F \subset Q$ is the set of accept states

Visually the FSM can be represented as a directed graph with states and transitions, both illustrated in Figure 3.2.

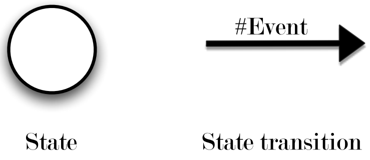


Figure 3.2: Illustration of a state and a transition arc.

When applied to game AI each of the states represent some behavior or an action of an object. Σ will be the set of events or conditions in the game which can trigger a transition from one state to another. An example of this is given in Figure 3.3.

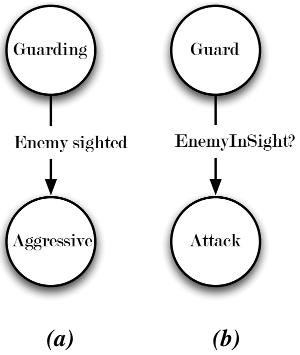


Figure 3.3: Two simple finite state machines based on either behaviors (a) or actions (b).

We observe two FSMs which both model a unit standing guard. The unit will stand guard as long as it does not spot an enemy. Depending on how the unit is modeled the sighting of an enemy will either trigger an event or change a condition according to Figure 3.3. With each iteration of a game, events will be observed and conditions will be updated. Considering Figure 3.3a, if at some point the event **Enemy sighted** is fired, it triggers the transition function $\delta(Guarding, Enemy\ sighted) = Aggressive$. As a result of this transition function the behavior of the unit will change to **Aggressive**. In a simple scenario such as this, we can assume that Figure 3.3b would represent roughly the same behavior. When the condition **EnemyInSight?** becomes true, the unit will execute the **Attack** action.

The difference between the two FSMs in figure 3.3 is that Figure 3.3a implements the **attack** action as part of its aggressive behavior, whereas the FSM in Figure 3.3b executes the **attack** action as a direct result of the condition. If states are used to represent behavior, the states implement all the actions relevant to the behavior and are responsible for executing the entire behavior. When using FSMs to design computer game AI, both approaches can be applied and both have their pros and cons. Using behavior will often provide a better overview with fewer states and transitions, but requires more lines of code in each state. On the other hand, if every state represents a single action, the number of states and transitions can become quite numerous. It is to the programmers and designers to decide upon which approach to use.

3.2.2 Hierarchical Finite State Machines

Even though FSMs are often adequate to describe the behavior of a game character the number of states and transitions often present a problem. When the complexity of a game increases the number of states and transitions between these will rapidly increase. It quickly becomes very difficult for the game designers to keep track of the different states and their transitions. Thus working with regular FSMs in larger games is tedious work to say the least.

An extension of FSMs, HFSMs [5], addresses some of these problems. This model introduces the concept of modularity by allowing groups of states to share transitions. The purpose of this is to avoid redundant transitions and get a better overview of the model. Using HFSMs it also becomes easier to group action states to form behaviors. An example of this is illustrated in Figure 3.4.

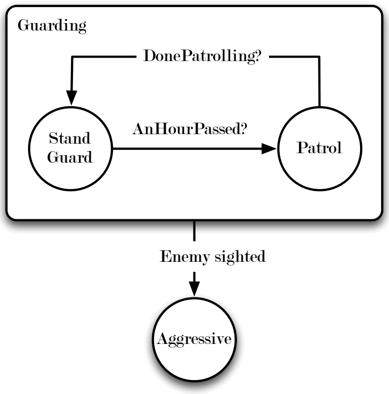


Figure 3.4: Illustration of a HFSM modeling a guard unit.

This illustration is a modification of the FSMs from Figure 3.3. The **Stand Guard** and **Patrol** actions have been grouped together to form the more general **Guarding** behavior super state. This grouping should be understood s.t. whenever an enemy unit is sighted from any state in the **Guarding** super state, it triggers the **Enemy sighted** event and changes behavior to **Aggressive**. In short, there is a transition from both **Stand Guard** and **Patrol** to **Aggressive**.

We could have modeled the **Aggressive** behavior state in the same way, including the actions and transitions necessary to model an aggressive unit. By doing this, events that causes a unit to abandon the aggressive behavior, will need only one transition from the **Aggressive** super state, instead of one from each of the inner states.

3.2.3 Example

To show how HFSMs can be applied to our scenario in StarCraft, we have illustrated a small harass behavior in Figure 3.5. The scenario assumes that we have a unit standing idle awaiting orders.

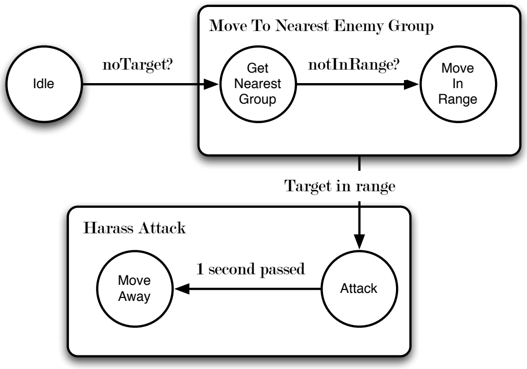


Figure 3.5: Illustration of a HFSM modeling a harass behavior in StarCraft.

If the unit does not have a target it will transition to the **Move To Nearest Enemy Group** super state. In this state there are two actions **Get Nearest Group** and **Move In Range**. Both can transition to the **Attack** state if the **Target In Range** event is triggered. The **Attack** state is contained in the more general **Harass Attack** super state, which specifies that a harass attack behavior is characterized by attacking an enemy unit for one second and then moving away from it. This very simple HFSM illustrates how unit behavior in StarCraft can be expressed, and is only meant to give insight as how a potential behavior could be modeled. It should be clear to the reader, that more complex behavior would result in a quite big HFSM with many more states and transitions.

3.2.4 Summary

The HFSM model is a great improvement over the FSM model, as it greatly reduces the number of transition arcs and allow for a better overview of the behavior. The comprehensibility of the method is also of great help to the game designers, who might not have expert knowledge on programming. Still, when dealing with complex AI, the number of states and transitions can become quite extensive, resulting in an opaque model structure. This also poses a problem if parts of a behavior need to be changed, as it can be difficult to see through the consequences it might have in other parts of the behavior.

Furthermore, to even consider HFSMs, the behavior must be easily decomposable in order to divide it into different states. This can be rather difficult with more complex behavior. On the other hand, behavior may also become too simple with the risk of states becoming visible to the player. This will end up shattering the illusion of intelligence and may decrease the players game experience. All in all, HFSMs provides a very intuitive framework for dealing with transitions between behaviors, but at the cost of being less flexible.

3.3 Behavior Trees

The concept of a BTs is a relatively new approach to behavior design. BTs combine elements from both scripting and HFSMs to provide a flexible framework, usable by both game designers and programmers, with as little complexity as possible. The structure of BTs are also meant to provide a more scalable approach than HFSMs, by reducing structural complexity.

BTs have already been used in some recent games, including Halo 2 [10], Halo 3 [8] and Spore [9]. This indicates that the method is not only applicable in theory, but also in practice. Our investigation suggests that the typical use for BTs in commercial games is mainly for modelling NPC behavior. So far, we have not seen any examples of BTs in RTS games, which means that further investigation is needed to confirm whether it is a viable solution.

In this section we will define the syntax and semantics of BTs. This will be followed by an example based on a small part of the StarCraft scenario specified in Section 2.4. Concluding the section will be a comparizon of the previously mentioned methods.

3.3.1 Syntax

The syntax, we have chosen for the BTs described in this project is based on [23] with some modifications. A BT is a tree structure with a single root node specifying the beginning of the tree. The root node can only have one child, which can either be a non-leaf node or a leaf node. All other node constructs can only have one parent.

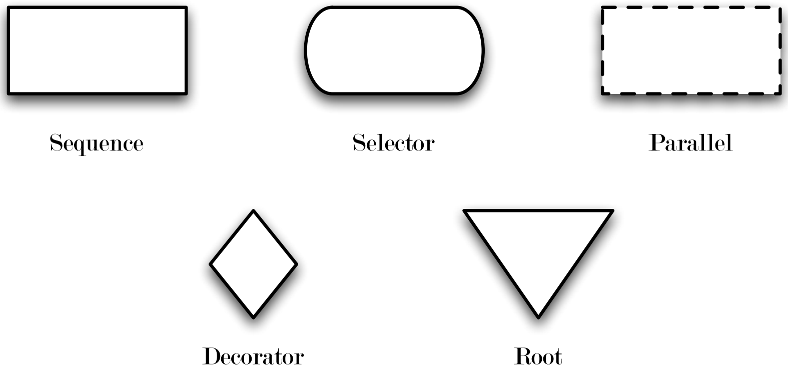


Figure 3.6: Illustration of non-leaf nodes in a BT.

The non-leaf nodes, which are illustrated in Figure 3.6, must each have any finite number of children, with the exception of the root node and decorator node which must have exactly one child. These non-leaf nodes can be viewed as tasks to be performed. Each task is defined by a subtree consisting of the different elements that make up the task. There are two basic non-leaf nodes that act as complements of each other. These are the *selectors* and *sequences*. Furthermore, there are *decorators*, which can be used to provide more functionality to the BT and *parallels* which adds concurrency.

The leaf nodes, illustrated in Figure 3.7, in a BT specify an observation or interactions with the game environment. These leaf nodes are called **conditions**, **actions** and **links**. Leaf nodes are often viewed as elementary actions and should be as concise as possible.

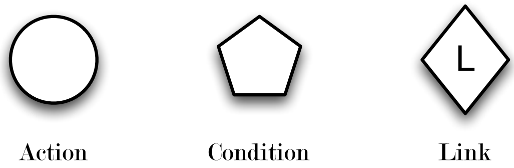


Figure 3.7: Illustration of leaf nodes in a BT.

3.3.2 Semantics

The execution of a BT is done depth-first, starting with the root node. Usually, all non-leaf nodes will execute their children from left to right, but there are exceptions. When a node has finished executing, it returns a status which can either be success, failure or exception. The circumstances under which success or failure is returned, depend on the node type. Exception is returned in the case that something went wrong and the node was unable to produce either a success or failure.

Selector

The selector node will sequentially try to execute its child nodes from left to right until it receives a successful response. When a successful response is received it responds to its parent with a success. If a child node responds with a failure, the selector node executes the next child node in line. If all child nodes respond with failure the selector node itself responds with a failure.

Probability Selector

A probability selector is a selector node with a probability distribution over its children. The probability indicates how likely a child is to be chosen during execution. If the child node chosen responds with a failure it will normalize the probability distribution over the remaining children and choose a new child to be executed. It responds to its parent in the same way as a normal selector node.

Sequence

The sequence node will execute each of its child nodes in sequence from left to right until a failure is received. If every child node responds with a success the sequence node itself will respond with a success to its parent. If somewhere during the sequence of execution, a child node responds with a failure, the sequence node will respond with a failure.

Decorator

The decorator node can be added to the BT to provide even more flexibility. A decorator is essentially a construct that allows for additional behavior to be added without modifying existing code. Decorators are most commonly used as filters with conditions such as, execute once, execute with some probability, etc. Besides filtering, a decorator can be used for pretty much any behavior modification the developer can think of. In contrast to selector nodes and sequence nodes, decorators only have one child node.

A decorator returns success if the specified conditions are met and its subtree has been executed successfully, otherwise it returns with a failure. A decorator which produces a failure will not execute its subtree.

Parallel node

A parallel node is a node, which concurrently executes all of its children. The circumstances under which a parallel node should return success or failure is up to the programmer. It could be that the parallel node is only successful if all its children are successful, like the sequence node, or it could behave like a selector node, only waiting for one successful child.

Condition

Conditions are nodes that will observe the state of the game environment and respond with either a success or a failure based on the observation. This could be a value comparison, a condition check, etc.

Action

Actions are nodes that are used to interact with the game environment. Through actions we can control various aspects of a game character such as movement and interaction with objects. When actions are performed successfully, the action node will respond with a success. If the character fails to perform the action, the action node will respond with a failure.

An action should be as simple as possible and often represent single actions in the game world. Even though an action could be programmed to represent an entire behavior itself, this would be bad practice as it would neglect the entire use of the BT structure.

Link Node

A link node holds a link to the root of another BT. When a link node is executed it will execute the linked BT and wait for a response. If the linked BT is successfully executed, it will respond with success, otherwise it will respond with failure. This introduces modularity and reusability of behaviors.

3.3.3 Black Board

During the execution of a BT it is often necessary to store information obtained from action nodes or condition nodes at run time. Such information could be coordinates, names, probabilities, etc. Storing such information enables one to obtain data in some part of the tree and store that data for later use.

To store this information we introduce the black board. The black board is basically a construct that stores whatever information is written on it. We allow for the black board to be public such that any node inside or outside the tree can read to and write on it. The reason is, that we want a mechanism for sharing data between BTs.

The black board is meant to provide more freedom to the designer, but at the same time it demands more responsibility. The information stored on the black board must be meaningful both in regards to naming and context in which it is meant to be used. If used the wrong way it will quickly become difficult to maintain.

3.3.4 Example

We illustrate, in Figure 3.8, how a BT can be applied to a simple harass behavior of a unit in StarCraft.

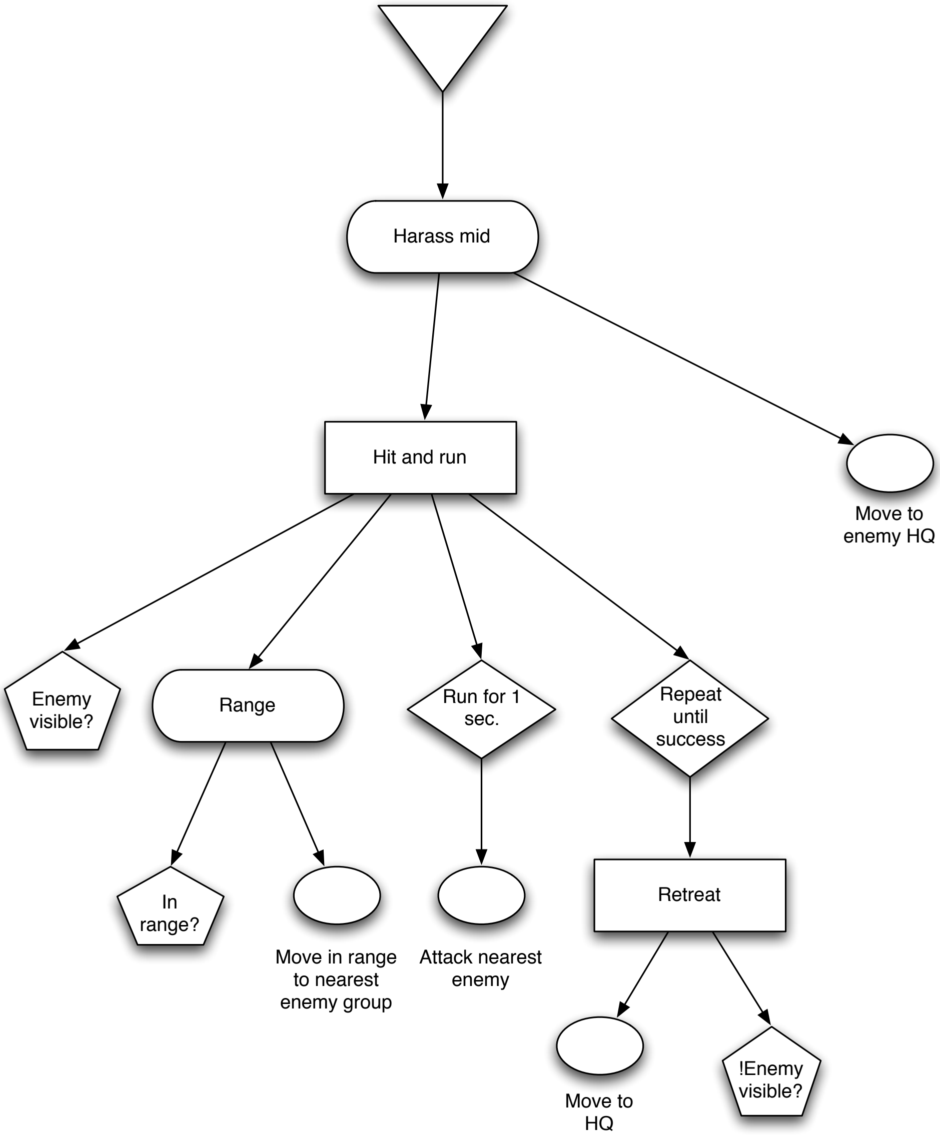


Figure 3.8: An illustration of a simple harass behavior modeled as a BT.

When this BT is executed the root node will run its child, the **Harass mid** selector node. This will execute the left-most side of the tree first, starting with the **Hit and run** sequence and then the **Enemy visible?** condition. This condition will check whether an enemy is visible somewhere in their path. If this is the case it will produce a success and the sequence will move on to the **Range** selector, which checks whether the visible enemy is inside range of the unit's weapons. If this is not the case, it will execute the action node and move the units to the nearest enemy group. When inside range, a timer decorator will execute the **Attack nearest enemy** action for at most one second. At last the **Repeat until success** decorator will execute the **Retreat** sequence, causing the units to move back as long as the enemies are visible. In case the **Hit and run** sequence fails, the **Harass mid** selector will execute the **Move to enemy HQ** action, sending the units towards the enemy's head quarter.

This all illustrates how a BT can be used to create a harass behavior in a simple way. Provided with a proper GUI, the creation and editing of this tree would be a very simple task.

3.3.5 Summary

BTs provide an intuitive and comprehensible framework that can easily be applied to a game scenario. The structure and semantics of BTs make it easy to follow the flow of the behavior, which is useful for both understanding, modifying and debugging of existing behavior. The basic constructs provided by the framework makes adding or removing parts of a behavior a simple task, and the modularity it provides scales well as the behaviors grow in size. Another great construct when dealing with the problems of scalability and reusability is the link node. By dividing smaller behaviors into BTs of their own, one can keep overview of a larger behavior by combining smaller sub-behaviors via link nodes.

As with HFSMs, one still has to maintain an overview of the structure and identify the individual parts of the behavior to decompose it into the tree structure. For this reason BTs may exhibit some of the same issues as HFSMs, where parts of the tree may become apparent to the player, thus shattering the illusion of intelligence.

3.4 Additional Methods

The preceding discussion on AI methods applicable to StarCraft only cover methods able to exhibit behavior that is completely static. Static in the sense that everything the AI is able to do has to be predefined and precoded. In order to give an evaluation on these methods and their limitations, the following covers a brief introduction on planning and learning so that a comparison between these methods and the ones already discussed is possible.

3.4.1 Planning

The notion of planning can be described as the following: Given an initial state and a goal state, we wish to obtain or reach, construct a sequence of actions that enables us to reach the goal state. This simplified notion of planning is what we also refer to as *Classical Planning* [21].

In classical planning the states are represented by atoms, or rather, conjunction of atoms. Atoms being relations optionally followed by a set of parameters, where the atom is true if the relation holds between the terms given as parameters. Actions are not directly represented as simple functions on the environment, but are instead described by action schemas. Action schemas define both actions and the effects of these actions. That is, when selecting an action for a plan, we know the corresponding effect of that action, where the effect is the resulting new state. A set of preconditions for the action is also included in the action schema. The action can only be taken if these preconditions hold true.

The problem of actually planning a sequence of actions can be viewed as a search problem. Starting in the initial state we can simply search the state space by traversing available actions from state to state until a goal state is reached. Because of the declarative representation of action schemas there are two possible ways to do this; forward search from the initial state and backward search from the goal state. During execution of the plan, re-planning can be utilized if unexpected states are reached as a result of following the plan. This introduces some overhead, because a new plan has to be computed.

Depending on the planning problem at hand, the state space can become so large that simple search becomes unfeasible. To overcome this problem a heuristic function is employed.

With the basics of planning described, two different extensions to classical planning will be discussed.

Hierarchical Task Networks

Like classical planning, *Hierarchical Task Network* (HTN) planning use the notion of conjunctive atoms to represent states and action schemas to describe actions. However, compared to classical planning, we do not plan to achieve a goal, but instead perform some set of *tasks*. A task is defined as one or more composites of sequential ordered actions or tasks. Each task holds a prescription of the distinct sequences it can be decomposed into.

HTN planning is the process of recursively decomposing these tasks until only a sequence of actions are left [12]. Actions which can be performed by an agent following the plan.

HTN planners take great advantage of the available knowledge on how tasks may be decomposed. That is, whenever we encounter a task for which a plan has already been constructed through decomposition, we may reuse that plan instead of having to search for a plan. This can greatly reduce the computational time spent searching for new plans. HTN planners are also able to expand existing knowledge with new decompositions of tasks. This knowledge base is sometimes referred to as a *plan library*.

The strength of a plan library is also one of the reasons why HTN planning has been more widely used for practical applications compared to other planning methods. Game AI developers are for instance able to predefine decomposition of high level tasks so that the AI acts as expected when encountering certain tasks. On the other hand the game AI is able to construct behavior on its own for loosely defined tasks. HTN planning has among other games been used in the first-person shooter *Killzone 2* [20].

Goal Oriented Action Planning

Goal Oriented Action Planning (GOAP) is an extension to *STRIPS* (acronym for STanford Research Institute Problem Solver) which in terms is analogous to classical planning [1]. The GOAP extension was proposed by Jeff Orkin from the M.I.T. Media Lab with the motivation being to develop the A.I. for Monolith Productions first-person shooter *F.E.A.R.* [18]. GOAP has since been used in other games and game genres, e.g. the RTS game *Empire Total War* [19].

The notion of GOAP adds several extensions to STRIPS. The most important of these extensions are, including costs to actions and adding procedural preconditions and effects to the action schemas.

The cost of actions can be used as a heuristic when searching the state space for plans.

[1] The Planning Domain Definition Language (PDDL) used to formulate the planning problems in classical planning is a minor extension to the STRIPS planning language.

STRIPS and classical planning is limited in the action precondition in the sense that states can only be recognized by a conjunction of atoms. Adding procedural preconditions extends the formal view of the world, and allows additional filtering on states.

The last extension covered here is adding procedural effects to action schemas. Recall that action schemas also describe the effects of taking an action. The declarative nature of these effects imposes instantaneous change to the state when taking an action. In real world scenarios actions are not necessarily instantaneous and instead take some amount of time to execute. Remember from Section 3.2 that FSM states are procedural compared to the declarative states of planning. In GOAP both these notions of states are used.

The time element is implemented by connecting the planning system with FSMs. When taking an action in GOAP the procedural effects set the FSM state until the action is completed and the world is transitioned to a new declarative state.

3.4.2 Reinforcement Learning

Reinforcement learning can be described as learning which actions should be taken in an environment in order to maximize or minimize some cumulative reinforcement or reward [13]. However, reinforcement learning is not defined by a specific method to achieve this learning process. It is instead defined as characterizing a set of problems which we refer to as *reinforcement learning problems*. Any method that help solve this problem are referred to as a *reinforcement learning method*.

The reinforcement learning problem in general can be visualized by Figure 3.9. Here we have an agent who interacts with an environment by some actions a. Each action leads to some change in the environment represented to the agent as states, s. The information about the environment represented in the states is dependent on the observability of the environment to the agent. It is required that this environment is at least partially observable to the agent.

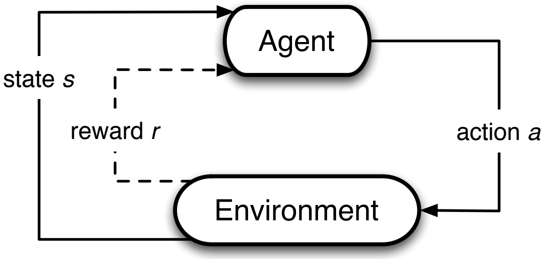


Figure 3.9: Interaction between agent and environment in reinforcement learning.

Changes in the environment may induce some reward r to the agent, both negative or positive. This reward is given by the *reinforcement function* which maps pairs of states and action to rewards. The learning part of the problem is deciding the state and action pairs that result in the maximum or minimum cumulative reward until some terminal state is reached.

The reward of an agent's actions may not be immediate, but instead depend on the reinforcement learning scenario at hand. Also, the goal of whether to maximize or minimize the cumulative reward also depends on the scenario. In the following we will however assume that the given scenario require us to maximize the reward. Consider the game Backgammon, where the configuration of the playing board would be represented as states, and the legal moves represented as available actions. Positive reward would be given only in terminal states corresponding to a win, and negative reward in terminal states corresponding to a loss. Any other states would give zero reward. This is referred to as *pure delayed reward*.

Given the reinforcement function for a reinforcement learning problem, how does the agent learn which state and action pairs are optimal? In the following, a *policy* determines which action should be taken in each state. To indicate how good a given state is, we will refer to it's *value* as the cumulative reward obtained when starting in that state and following a fixed policy to a terminal state. A *value function* describes a mapping from states to such values. Finding the optimal value function is the key problem of reinforcement learning, because with this, starting in any state we will know not only that state's value but also all reachable states' values through the available actions. Simply taking actions that result in states with maximum, equals having followed the optimal policy. In this way the cumulative reward or reinforcement will be maximized, as was the goal of reinforcement learning.

Finding the optimal value function is difficult, and usually done through approximation by dynamic programming.

3.5 Evaluation

In this chapter we have described and discussed several methods and techniques for AI design in a computer game scenario. Two of these methods, scripting and HFSMs, are widely applied in the game industry today. The last method, BTs, is a recently proposed method, which has been applied to a few games in the last few years. It is also worth noting, that we have found no material on BTs being used in RTS games. This suggest, that it has yet to be tested whether BTs are actually usable in this game genre.

Of the three methods we evaluated, scripting is the most commonly used for creating behavior in games. Scripting provides a simple solution for creating AIs, given little experience with programming. In the last couple of years there has been an increase in the number of tools, which allow for scripting via GUI-based editors, thus making delegation of behavior design an easier job for non-seasoned programmers and game designers.

FSMs have also been used for many years, mainly for NPC behavior, but have also been proven to work well in RTS game scenarios. FSMs provide a comprehensible framework for designers, but is not very flexible and does not allow for reusability of logic. This results in a design with an increasing number of duplicate states making it unnecessarily complex. The issue of increased design complexity has been addressed in HFSMs by introducing super states, but the method still suffers from lack of reusability and duplicate states.

BTs have been introduced to make up for the lack of reusability in HFSMs, by letting each node encapsulate its own logic. By doing this, a node will contain some behavior based on itself and its children. This behavior can be seen as independent from the rest of the context and can be reused where it is needed. This particular feature makes BTs very scalable compared to HFSMs and creating complex behaviors will be less tedious.

All of the three methods evaluated share some common flaws. Behavior created by scripting, HFSMs or BTs will react according to a predefined behavior. None of the methods provide any technique for learning by mistakes or planning ahead. As such the behavior will be somewhat static, as something unpredicted can never occur. Though the player may come to think of the AI as being intelligent, it is only an illusion of intelligence. If the player were to see through the model, he could easily take countermeasures resulting in a sure victory.

To gain some ideas for improvement of these methods, we chose to look at some techniques from the field of classic AI, namely planning and learning. Our intention was to identify the strengths of these techniques and use these as inspiration for future work.

Planning is shown to be very relevant when considering human-like behavior in games. Human-like behavior in general relies greatly on planning ahead, not only when playing games, but in everyday life. For this reason planning fits nicely with the concept of strategic thinking required in RTS games. Where we set up some goals and plan a way to reach them. In comparison, scripting, FSMs and BTs rely on reactive planning. That is, they only plan the next action to be taken based on the current context of the world. Planning has already been proven feasible in RTS games, as GOAP has been applied to Empire Total War. As such, it could be interesting to see how planning would fare in a fast paced RTS game like StarCraft.

Learning, as planning, is also essential when looking at the behavior of humans. When performing tasks we obtain experience, either by success or failure, and that experience is recalled next time we have to perform a similar task. We have discussed reinforcement learning as a technique, and described how it can be applied to a simple game of backgammon. This approach however, will try to find the optimal sequence of actions to reach a terminal state. To incorporate this in an RTS game, the reward received for an action would have to be defined in a way s.t. the sequence of actions does not result in super-human behavior. Instead the reward should be based on the gameplay experience, which is the variable we want to maximize. This is perhaps the greatest challenge of inducing learning in a game such as StarCraft, as gameplay experience is a difficult measurement to define.

In the end we have chosen to focus on BTs, as it is a fairly new method which, we believe, holds a lot of potential. Also, the apparent lack of use of BTs in RTS games, provides us with further challenges, as it could prove to be infeasible. If BTs prove to be applicable and efficient in an RTS game scenario, the study of planning and learning may inspire further improvements of the method, hopefully pushing towards more human-like behavior in StarCraft.

4 Framework and Editor

To support the construction and use of BTs we need an editor to create the structure of the tree and a framework that can be used to implement the structure and utilize the BWAPI.

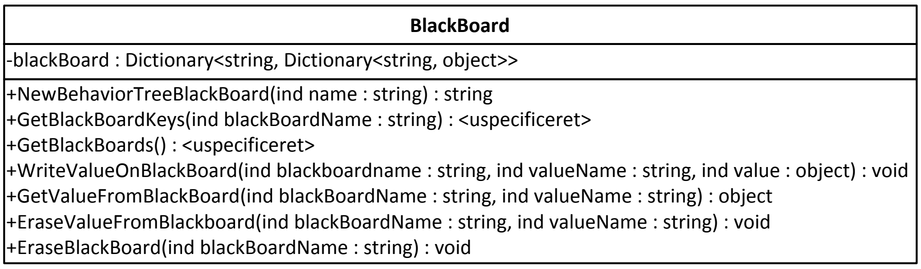
Given that BTs have only recently been adopted by the game industry for AI design, the availability of existing frameworks and editors for BT implementation is very limited. The open source C# project *TreeSharp* only includes a subset of the node constructs that BTs offer, and only offers the framework, not an editor. Another open source project, also developed in C#, called *Brainiac Designer* offers both a framework and a designer to utilize the framework. As with TreeSharp, this solution only includes a subset of the node constructs. *GameBrains* is a recently released BT middleware framework and editor, and is written in C++. It is available as open source, but currently only as a beta test agreement.

Because of this, we have developed our own graphical editor and framework supporting all the node constructs in BTs. In this way the framework and editor can be optimized for use with the BWAPI. In this chapter we will describe the approach used to create these two tools by giving an overview of the tools. We will explain the more complex parts of the tools in detail to give a better understanding. The framework has been implemented in C# using .NET 3.5 and the editor has been implemented in Windows Presentation Foundation (WPF).

4.1 Framework

In this section we describe the developed framework to use when implementing behaviors in games. As described, the framework has been implemented in C# and .NET and as such it requires the intended game to expose an API that can be used in C# and .NET. The BWAPI is, as described in Section 2.5, implemented in C++, which does not directly expose the API to C# and .NET. However, as we also described in Section 2.5 the *bwapi-mono-bridge* framework exposes the C++ API to the Mono framework. As the Mono framework is interoperable with .NET, the C++ API is available in C# through Mono.

We will provide an overview of all classes in the framework and show specific complexities in more detail where we deem it necessary. The full class diagram can be seen in Appendix A.



```
BlackBoard
-blackBoard : Dictionary<string, Dictionary<string, object>>
+NewBehaviorTreeBlackBoard(ind name : string) : string
+GetBlackBoardKeys(ind blackBoardName : string) : <uspecificeret>
+GetBlackBoards() : <uspecificeret>
+WriteValueOnBlackBoard(ind blackboardname : string, ind valueName : string, ind value : object) : void
+GetValueFromBlackBoard(ind blackBoardName : string, ind valueName : string) : object
+EraseValueFromBlackboard(ind blackBoardName : string, ind valueName : string) : void
+EraseBlackBoard(ind blackBoardName : string) : void
```

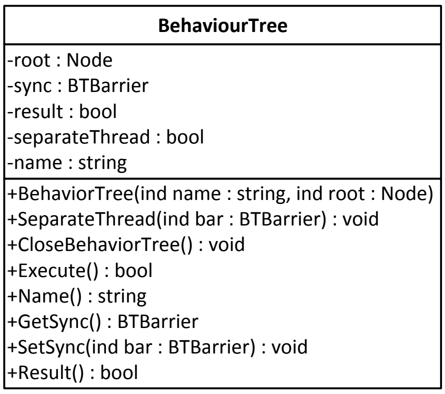
Figure 4.1: BlackBoard class

4.1.1 Auxilliary Classes

BlackBoard

The *BlackBoard* class shown Figure 4.1 provides a common place for all BTs to communicate. Each BT will on creation get a spot on the blackboard. All BTs can read from and write to all other BT's blackboards thus creating the ability to communicate information between trees.

The blackboard is implemented as a dictionary of dictionaries. Each BT gets a dictionary in the outer dictionary with its name as the key and the inner dictionary as the value. The inner dictionary consists of string keys and object values. This allows the designer to store all kinds of information on the blackboard under a given string name. This of course demands some responsibility from the designer as he or she will need to keep track of what is written where on the blackboard.



BehaviourTree

-root : Node
-sync : BTBarrier
-result : bool
-separateThread : bool
-name : string

+BehaviorTree(ind name : string, ind root : Node)
+SeparateThread(ind bar : BTBarrier) : void
+CloseBehaviorTree() : void
+Execute() : bool
+Name() : string
+GetSync() : BTBarrier
+SetSync(ind bar : BTBarrier) : void
+Result() : bool

Figure 4.2: BehaviorTree class

BehaviorTreeThreadWrapper

The *BehaviorTreeThreadWrapper* class seen in Figure 4.4 encapsulates the threaded execution of a BT in an easy to use class. The wrapper is instantiated with a tree and it creates a new barrier and a new thread for use in the execution. This barrier is then set in the tree and all of its nodes. The new thread is given the private *Run* method which calls the execute method on the tree.

The wrapper exposes three methods for execution; *Start*, *Continue* and *End*. The *Start* method starts the execution thread and waits for it to reach the first node that requires it to wait at the barrier. Once the barrier is reached the *Start* method returns. It is now up to the designer to repeatedly call the *Continue* method for as long as necessary.

The *Continue* will signal the barrier and allow the BT to continue execution. If the tree has not yet completed its execution, the *Continue* method will wait for it to complete or reach the barrier again.

If the BT has completed execution and the *Continue* method is called nothing will happen. If the tree needs to be re-executed, the *Restart* can be called which resets the tree and restarts the thread.

The *End* method attempts to join the execution thread so that the execution can be completed.

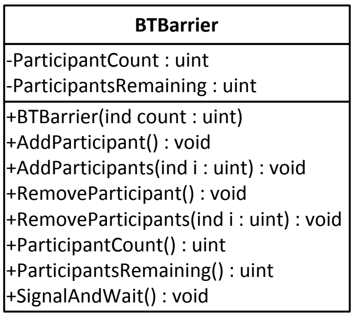

BTBarrier
-ParticipantCount : uint
-ParticipantsRemaining : uint
+BTBarrier(ind count : uint)
+AddParticipant() : void
+AddParticipants(ind i : uint) : void
+RemoveParticipant() : void
+RemoveParticipants(ind i : uint) : void
+ParticipantCount() : uint
+ParticipantsRemaining() : uint
+SignalAndWait() : void

Figure 4.3: BTBarrier class

BTBarrier

Due to the fact that the Mono does not implement .NET 4.0 completely it was necessary to implement a simple version of the *Barrier* class (Figure 4.3) from .NET 4.0. This is done by using two counters, one for counting the total number of participants and one for counting the number of participants not yet at the barrier. Once a thread reaches the barrier it signals the barrier and waits. The wait is done using the *Monitor* object from *System.Threading* in .NET. If the number of remaining participants is greater than 0 the thread waits and if there are no remaining participants all threads are pulsed and resume execution.

4.1.2 Behavior Tree-Specific Classes

BehaviorTree

The BT class depicted in Figure 4.2 is in itself quite simple. It contains a *Node* object as its root node and a name of the BT to allow for retrieval of the correct spot on the blackboard. The BT also contains a reference to the active barrier in the tree execution. This barrier will be explained later. When the barrier is set, in a *BehaviorTree* object, the method pushes the same barrier onto the root node which in turn will apply it to any children it may have.

The Behavior Tree class also contains a method to allow execution in a thread separate from the creating and executing thread. With this in hand the calling thread can continue execution while the tree runs its actions. The execution of the tree in a separate thread is made easier when using the *BehaviorTreeThreadWrapper* class described above.

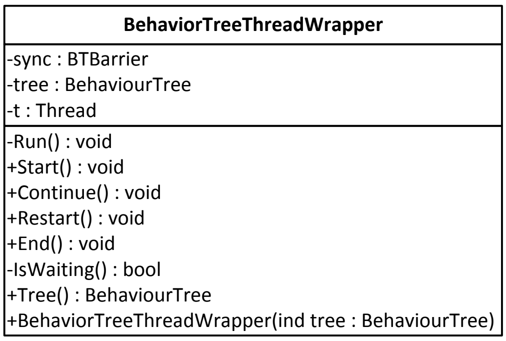


Figure 4.4: BehaviorTreeThreadWrapper class

Nodes

As described in the previous chapters BTs have several different types of nodes and they all have a specific semantic regarding their execution. Below is a description of each of the nodes available in our framework.

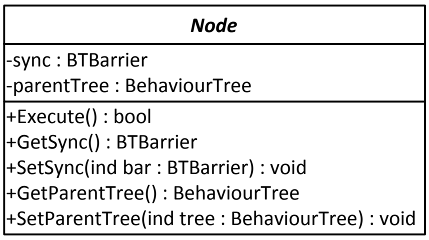

Node
-sync : BTBarrier
-parentTree : BehaviourTree
+Execute() : bool
+GetSync() : BTBarrier
+SetSync(ind bar : BTBarrier) : void
+GetParentTree() : BehaviourTree
+SetParentTree(ind tree : BehaviourTree) : void

Figure 4.5: Node class

Node

Node is the main class given in Figure 4.5. All types of nodes are descendants from this class. The class is abstract and as such each descendant must implement all abstract methods defined in this class. The class also specifies methods for how to propagate the barrier object and the parent BT object to its children if there are any. The Node class specifies that each descendant must implement the Execute method which is used to execute the BT.

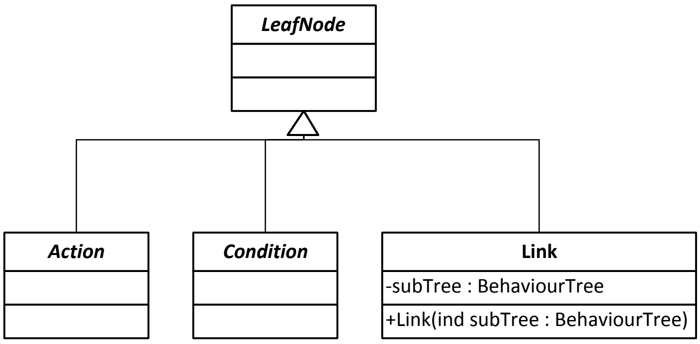


Figure 4.6: LeafNode classes

LeafNode

We have split nodes into two groups. The first group is *LeafNode* depicted in Figure 4.6. This class is abstract and serves as a way of splitting nodes that can have children from those that cannot. All descendants of *LeafNode* must not have children.

Action

The *Action* class is the class representing the action node in a BT. This class is abstract as the action nodes require details about the available actions in the game API. As such, programmers and game designers must create new descendants of *Action* that implement specific actions in the game environment.

Condition

As with action nodes condition nodes are also not possible to define before the game API is known. For this reason the *Condition* class is abstract. Again it is necessary for the programmers and game designers to implement conditions that use information from the game environment to determine if the result is true or false.

Link

The link node of a BT is implemented in the *Link* class. A link node contains a link to another BT that must be executed in the link node. The result of the link node is based on the result of the tree it calls. When the *Execute* method is called on a *Link* object the *Execute* method of the tree it references is called and the execution of that tree is started.

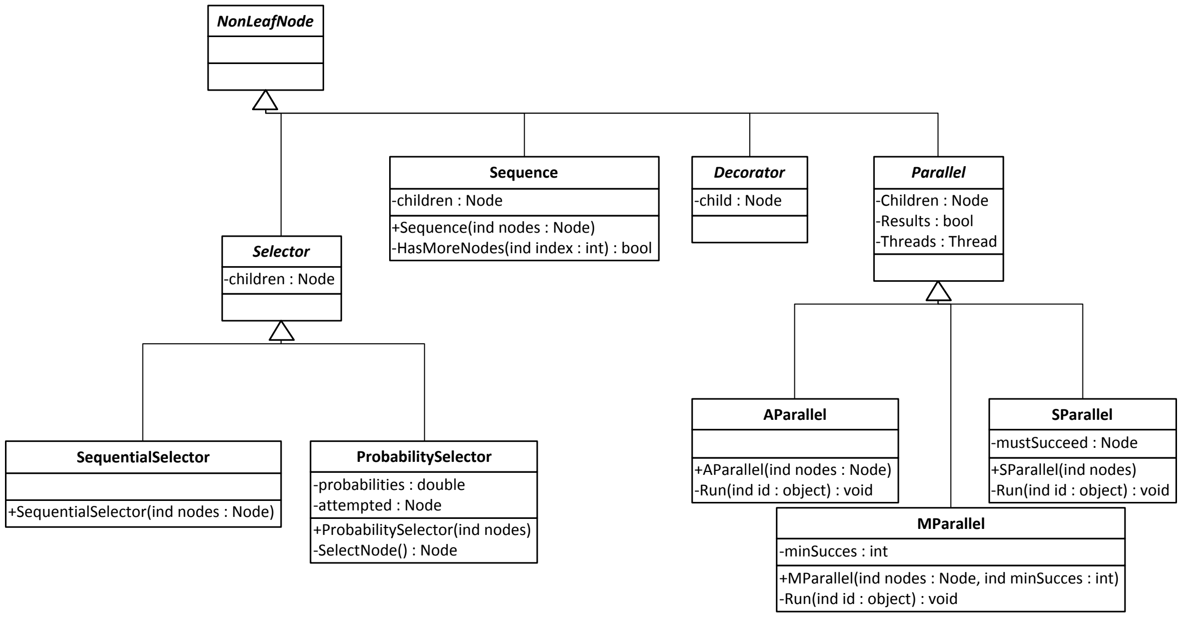


Figure 4.7: NonLeafNode classes

NonLeafNode

The other group of nodes in our framework are the non-leaf nodes illustrated in Figure 4.7. These are represented with the class *NonLeafNode*. Descendants from this class are allowed to have children.

Decorator

The decorator node is, as described previously, a node that can add functionality, not available through the other nodes, to the tree. We cannot define a generic way for a decorator to be implemented as this both depends on the game API and on the feature that it implements. The class is abstract and the programmer and game designer must define how their decorators work and implement it accordingly. It is required that decorators must be non-leaf nodes and as such are required to have one child.

Sequence

A sequence node is implemented in a generic way through the *Sequence* class. This class executes all of its children one after another until either one of the children fails or all children have succeeded. This implements the exact semantic of the node as previously defined.

Parallel

The parallel nodes are used for executing several branches of a BT simultaneously. As the success or failure of a parallel node is hard to define we have chosen to implement three different types of parallel nodes, each with their own demands for successful termination.

AParallel

The *AParallel* class implements a version of the parallel node where the requirement for successful termination of the node is given by the successful termination of all parallel branches. This means that all branches must succeed for the *AParallel* to succeed.

MParallel

The *MParallel* class relies on a certain number of branches to succeed in order for the node to succeed. When instantiating the node a minimum number of successful branches is given along with all of the branches. When all branches are done executing the node checks how many were successful and if the number of successful branches is greater than or equal to the minimum number of required successful branches the *MParallel* is deemed successful.

SParallel

The last implementation of the parallel node is the *SParallel* class. This class implements a version of the parallel, where several branches are given along with a list of which branches must succeed, in order for the *SParallel* to succeed. There may be branches that are not necessary for the successful run of the parallel and these may fail or succeed but have no influence on the result of the parallel node. Only the branches defined in the list of branches that must succeed have an influence on the result of the *SParallel*

Selector

Selector nodes are used for selecting a branch of a BT based on some defined way of selecting. In our framework we implement two types of selector nodes and these are described below.

SequentialSelector

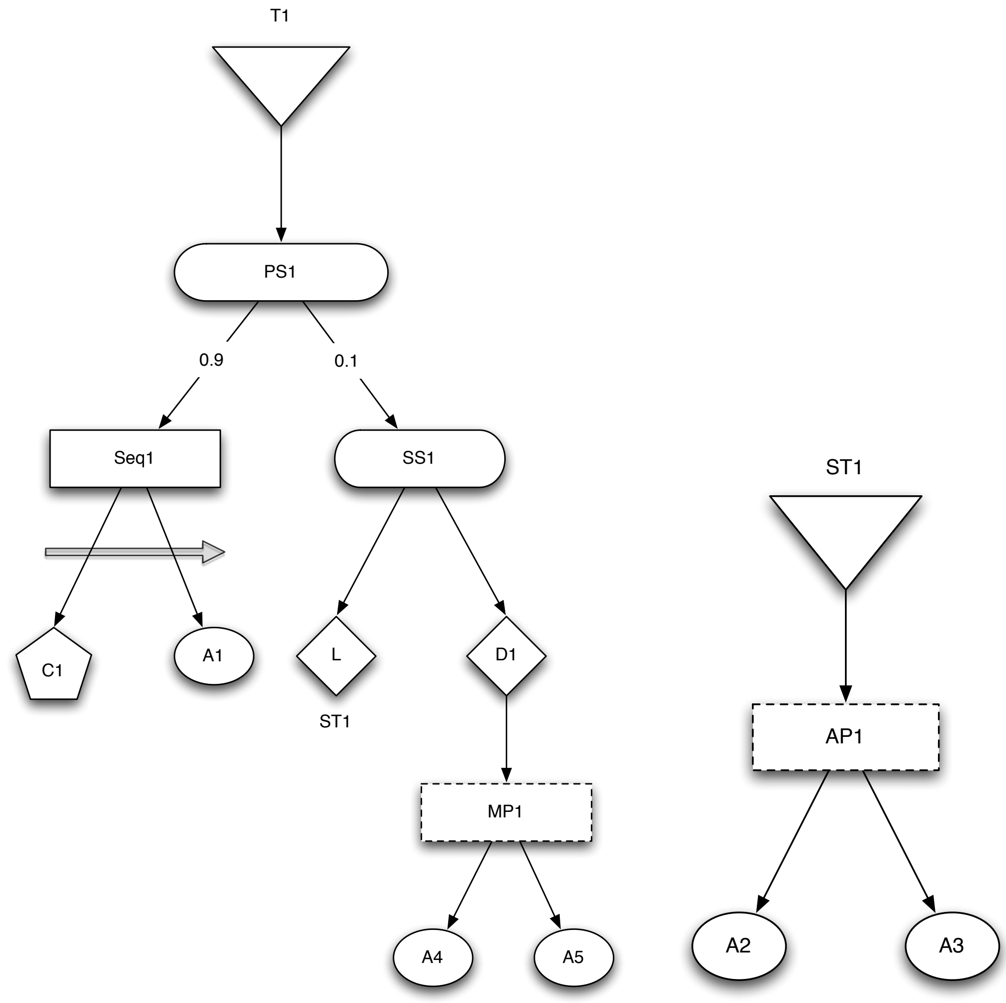
The *SequentialSelector* class is the simplest of the selector nodes. It simply tries executing its children one by one until one of them succeeds. If no children succeed the selector fails.

ProbabilitySelector

The *ProbabilitySelector* class implements a selector that selects amongst its children based on a probability for each child. As it is given by the definition of a selector node the *ProbabilitySelector* node must select a child based on the probability for each of the children until one of them succeeds.

4.1.3 Execution of a Behavior Tree

The nodes and features of the BT framework have been described but the structure of the framework is not enough to successfully construct a BT. It is also required that the programmer and game designer know how a BT is executed. In this section we will describe how the execution of a BT is done. We will use the BTs shown in Figure 4.8 as an example for the description.



(a) BT $T1$ *(b)* BT $ST1$

Figure 4.8: BTs for examplifying execution

In Figure 4.8 we see two trees. The first, $T1$, is the main tree of the example and $ST1$ is a subtree that we will use to show how link nodes work. As the trees are used to examplify execution we will not define what specific actions and conditions do but rather concentrate on the flow of execution . We also define the results of the leaf nodes, conditions and actions, in Table 4.1.

Node Name	Result	Execution Time
C1	Success	1 ms
A1	Failure	5 s
A2	Success	2 s
A3	Failure	1 s
A4	Failure	50 ms
A5	Success	5 s

Table 4.1: Execution result for leaf nodes

We recommend that the programmer uses the BT thread wrapper for executing BTs. This is due to the handling of the barriers needed, if an action takes time to execute and the exectuing thread cannot hold until the action is done. For example pausing execution, so the game frame can terminate, avoiding lag in the game experience. Specifically, in StarCraft there is on average 40 ms of execution time per game frame. Even though the actions do not take time to finish execution it is recommended to use the wrapper if the tree contains parallel nodes. This ensures that the parallel returns correctly as all threads are synchronized around the barrier from the wrapper. This is also why our example will use the thread wrapper class.

When the BT is instantiated with the nodes it contains, it is added to a *BehaviorTreeThreadWrapper*, which sets a barrier throughout the BT and subtrees. Once this is done the thread wrapper is ready to execute the BT. The execution is initiated with the *Start* method. This method starts the thread executing the BT.

The execution of a BT starts by calling the *Execute* method of the root node of the tree. All nodes are required to implement the *Execute* method to ease execution. The root node in the example in Figure 4.8 is $PS1$, which is a probability selector node. The probabilities define how likely the node is to execute each of its children. $PS1$ will execute the first branch with probability 0.9 and the second branch with probability 0.1. The *Execute* method of the probability selector will select a branch based on these probabilities and here we assume it selects the first branch, due to the high probability.

The probability selector then executes the *Execute* method of the child in the first branch. This child is the sequence node $Seq1$. $Seq1$ will start executing its children one by one until either one fails or all children succeed. $Seq1$ starts by calling *Execute* on the condition node $C1$. From the Table 4.1 we see that $C1$ is defined as succeeding with an execution time of one ms. For this reason we will not need to pause the execution as one ms execution will not cause lag in the game. This means that the first child of $Seq1$ has succeeded. $Seq1$ will then execute the next child which is the action node $A1$.

The action node $A1$ can take time to execute and in Table 4.1 we define it as taking five seconds to execute. For this reason we need to allow the tree to pause execution while the action is performed in the game environment. To better illustrate why this is important, imagine that the action is a move command to a unit. It will take some amount of time for the unit to move to the given position and the action node cannot return success or failure before it has either reached the position or it has failed to do so. Here the barrier from the thread wrapper comes into play.

As we are still running as part of the *Start* method of the wrapper, we halt the execution of the tree by signaling and waiting at the barrier. The thread wrapper is waiting for this condition to be true. Once this happens the *Start* method will return and the game can continue execution. To resume execution of the tree, the *Continue* method must be called e.g. on the event of a new game frame. This method releases all those threads waiting at the barrier and waits for them to either terminate or reach the barrier again. $A1$ may take several game frames to terminate and as such the thread may be awoken many times before it can determine if the action has terminated.

$A1$ is defined to be a failure and will return this to $Seq1$. $Seq1$ will then, because one of its children failed, return a failure to $PS1$. A selector node handles failure by executing another of its children. $PS1$ will try to execute $SS1$ which is the other child of the selector. $SS1$ is a sequential selector that will attempt to execute its children one by one until one succeeds or all children fail.

$SS1$ will first execute the link node pointing to $ST1$. A link node simply executes another BT and the success or failure of this tree defines the success or failure of the link node.

$ST1$ is a small subtree and is depicted in Figure 4.8b. The root node of the tree is the parallel node $AP1$. $AP1$ requires all of its child threads to succeed in order to succeed itself. $AP1$ starts two new threads, one for each of $A2$ and $A3$. $AP1$ then removes a participant from the barrier and adds two new for the children. The reason that a parallel node removes a participant from the barrier is because the parallel node will start busy waiting on the child threads to complete. This must be done separately from the barrier, as it would not reach the required participants to pause execution of the tree otherwise. Each child, $A2$ and $A3$, will now execute their actions. These actions take some time to execute as well and this will cause the actions to pause execution until they are done. In Table 4.1 we see that $A2$ succeeds and $A3$ fails. As $AP1$ required all children to succeed the node fails resulting in a failure of $ST1$.

$SS1$ will attempt to execute its second child as $ST1$ failed. $SS1$ then executes $D1$. $D1$ is a decorator. Decorators are as explained the wild cards of BTs. These nodes can add features not available through the use of other nodes. In our example we imagine that $D1$ runs an algorithm to select a position on the game map needed in the rest of the branch. $D1$ then proceeds to execute its child $MP1$.

$MP1$ is a *MParallel* node. *MParallel* nodes are given a specific number of children that need to succeed in order for the node to succeed. $MP1$ has two children and it requires one of them to succeed. $A4$ and $A5$ will then be executed just as with $A2$ and $A3$ under $AP1$. Table 4.1 shows that $A4$ will fail rather quickly. This has no immediate effect on the node as the rest of the nodes need to terminate before the result can be determined. Once $A5$ terminates with success $MP1$ will determine that it has succeeded.

The success of $MP1$ will in turn affect $D1$ which also succeeds. $SS1$ will now also succeed as one of its children succeeded. $SS1$ is a child of $PS1$ which also relies on one of its children to succeed in order to succeed. This condition is now also true as $SS1$ succeeded. $PS1$ is the root node of $T1$ and as $PS1$ has now succeeded, the tree $T1$ has executed with success as a result.

4.2 Editor

The purpose of the editor is to supply the framework with the structure of the BTs, such that they are ready to be executed by the framework. For this, the editor is developed as a visual designer where BT nodes can be arranged and connected into BTs visually, giving an easily comprehensible structure of the BTs compared to the actual code representation in the framework.

The editor is implemented in WPF using an article series on developing a Diagram Designer by the user sukram available at The Code Project [22].

GUI Layout

The current revision of the editor offers a simple interface with a set of basic features needed to build BTs. Figure 4.9 shows the layout of the editor.

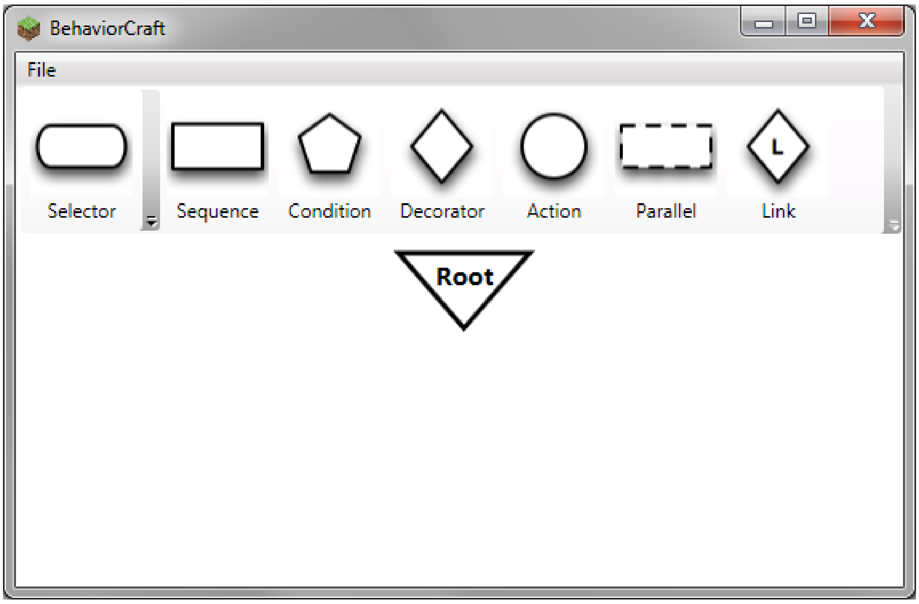


Figure 4.9: Main window layout in the editor.

The interface can be separated in two distinct areas: A menu area and a designer area. The menu area contains buttons for each available BT node that can be added to the designer area and used to contruct BTs. Implementation of the nodes and how they are presented in the designer area is discussed in Section 4.2. The *File* menu entry offers export to PNG image files of the BT in the designer area. Furthermore, the File menu entry holds an *Exit* option, which closes the editor.

Designer Canvas

The designer area handles visual representation of the BT nodes and allows creation of BTs and editing these. It is implemented with the *DesignerCanvas* class which inherits the *Canvas* class available in .NET. The canvas class defines an area where child elements can be explicitly positioned using coordinates within the canvas frame. Selecting one of the available BT nodes in the menu area, adds the given node to the child collection of the canvas, thus making it visible in the canvas frame.

The DesignerCanvas class overrides the Canvas class' *OnMouseDown*, *OnMouseMove* and *MeasureOverride* methods. Overriding the OnMouseDown and OnMouseMove methods, multiple nodes in the canvas can be selected for editing by holding down a mouse button and drag-selecting a set of nodes. This makes it easy to select a subtree of the current BT and e.g. reposition the entire branch. The MeasureOverride override handles resizing of the canvas when nodes are moved outside the bottom or right bounds of the canvas. The canvas is then properly resized according to the repositioning of the nodes.

Node User Controls

The individual BT nodes that are added to the canvas are implemented as simple *User Controls*. User controls can be viewed as reusable components that can contain other controls, resources and more. There are almost no limitations to what can be contained in a user control, making them ideal for creating the BT nodes.

The implementation of the sequence node user control in the XAML visual editor can be seen in Figure 4.10 while Listing 4.1 is the corresponding XAML code.

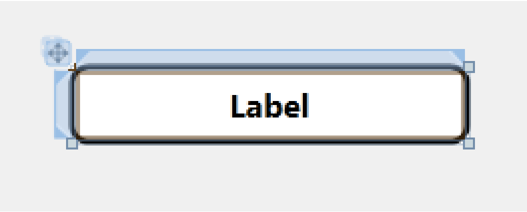


Figure 4.10: The sequence node user control in the XAML visual editor.

```
<UserControl x:Class="BehaviorCraft.Resources.Nodes.SelectorShape" mc:Ignorable="d" d:DesignHeight="40" d:DesignWidth="200">
    <Border BorderBrush="Black" BorderThickness="3,3,3,3" CornerRadius="8,8,8,8" Background="White">
        <Grid>
            <Label Content="Label" Height="28" HorizontalAlignment="Center" Margin="12,0,12,0" Name="SelectorLabel" VerticalAlignment="Center" VerticalContentAlignment="Center" FontWeight="Bold" FontSize="16" Padding="1"/>
        </Grid>
    </Border>
</UserControl>
```

Listing 4.1: The XAML code for a selector node.

All the BT node user controls share the <*Label*> control, with the *Name* property differing. This label handles the name displayed on the corresponding node, and is supplied by user input whenever a node is added to the canvas. Sizing of all the node user controls when added to the canvas is handled automatically with respect to the text length of the label.

When nodes are added to the canvas, they are not added directly by their user control representation. Instead the class *NodeItem* is used as a container class for all nodes being added to the canvas. This class inherits the *ContentControl* class from .NET, which allows us to add the node user controls as the NodeItems content using ContenControls *Content* property.

The NodeItem class also uses the *MoveThumb* and *Connector* classes as templates. The Connector class handles the connection points on nodes used to visually link parent and child nodes on the canvas. This is described in greater detail in Section 4.2. MoveThumb inherits the *Thumb* class from .NET, which has a *DragDelta* event. The DragDelta event is fired when a NodeItem is clicked and dragged. An *MoveThumb_DragDelta* eventhandler is added to DragDelta, which handles repositioning of NodeItems within the canvas as a response to the DragDelta event. This allows a user to freely reposition individual or multiple nodes within the canvas frame by simply dragging them around.

Connecting Nodes

Visually connecting parent and child nodes in the canvas is implemented using four classes, and is best described step by step. Figure 4.11 shows in four steps how a parent node is connected to a child node.

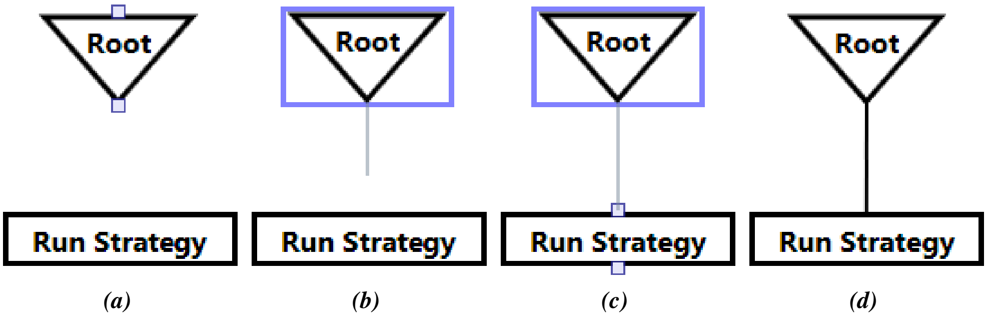


Figure 4.11: Creating a connection between a parent and a child node.

When hovering the mouse over a node in the canvas, two connector points on the node becomes visible as can be seen in Figure 4.11a. The connector points are added to the NodeItem as a template using the *Connector* class as the template. The Connector class handles the connector point relative to the node's position in the canvas. All nodes currently share the same Connector template, which is why both non-leaf and leaf nodes have two connector points.

By clicking the mouse and holding the mouse button a connection can be made to a child node by dragging a connection to it. Figure 4.11b shows how the drag connection is visually represented by a line between the parent node and the current position of the mouse. This line is handled by the *ConnectorAdorner* class using an *Adorner* from .NET. An Adorner works like an elastic rubberband in the sense that it can expand and contract according to e.g. the mouse being moved.

As the drag connection nears a connector point on the child node, the child nodes connector points becomes visible as shown in Figure 4.11c. Letting go of the mouse button with the drag connection on a connector point creates a connection between the parent and the child node. The connection is handled by the *Connection* class, which mainly handles the visual representation of a connection as seen in Figure 4.11d.

Finally, whenever a parent node, child node, or both are repositioned in the canvas frame as a result of a drag movement, the connection between these also need to be repositioned. This is handled by the *ConnectionAdorner* class, which inherits the Adorner class. An adorner is used as described for the drag connection previously.

Building a Behavior Tree

The editor currently supports all BT nodes specified in Section 3.3 and thus it is possible to build BTs utilizing all of these nodes. Figure 4.12 shows the strategy BT from Section 5.1 built in the editor, using selector, sequence, condition and link nodes.

The editor is available for testing on the CD enclosed with the report.

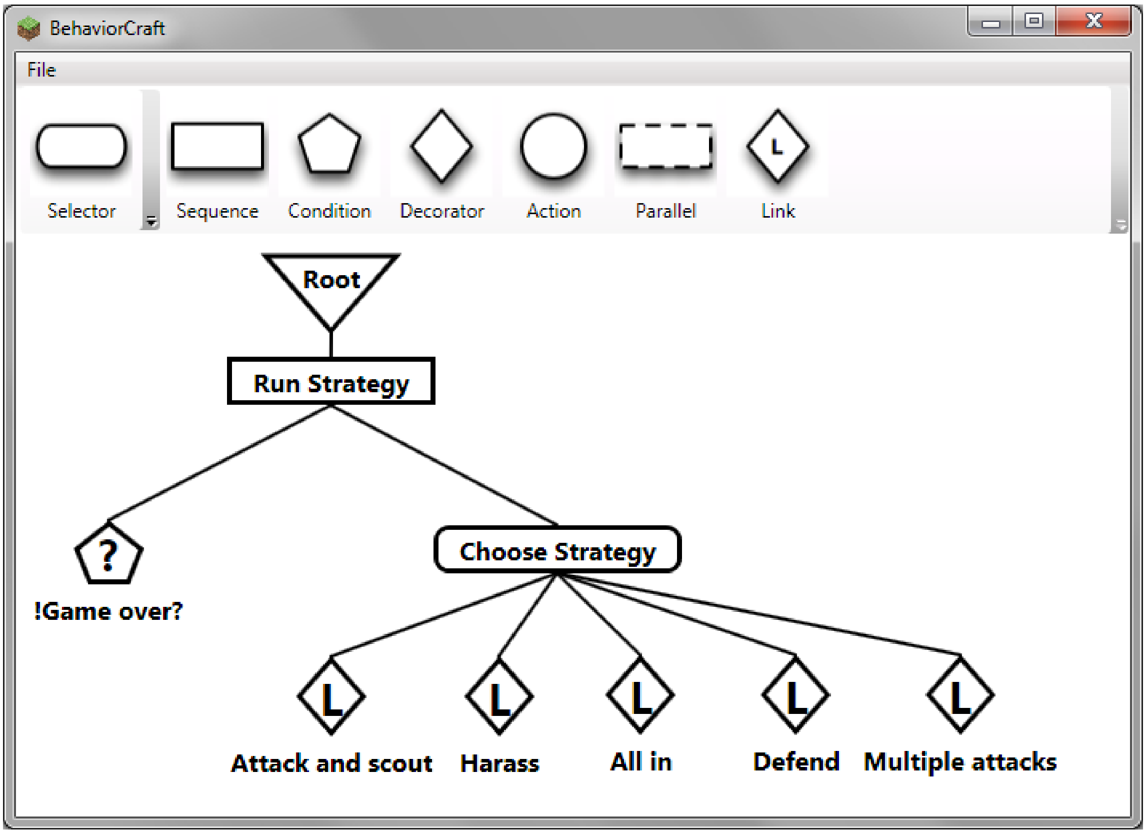


Figure 4.12: The strategy BT constructed in the editor.

5 Case Study

In the previous chapters we introduced the game Starcraft which we will use for our case study. Furthermore, we looked at the theory of BTs and introduced our proposed framework. In this chapter we will construct the BTs that we are going to use for testing. These trees are based on the simplified scenario we introduced in Section 2.4.

The way we decided to construct the BTs is by separating the what from the how. We have the main BT, *Strategy*, that consists of the strategies we can apply, which is the what. Each strategy is a link node that links to another BT. These trees define how the strategies are going to be executed, the how. Another way to construct the BTs, is to assume we have a goal we want to achieve. To achieve the goal we need some subgoals, and to achieve these subgoals we need sub-subgoals etc. As an example of this take a glance at Figure 5.1. Our main goal is to run a strategy. However, to run a strategy we need to choose one and to choose one we need some strategies we can choose from. We continue to do this until the whole tree is constructed. The following subsections describe our BTs in more detail.

5.1 Behavior Tree: Strategy

The *Strategy* BT is the top most or main tree of all the BTs. It is the tree that decides which strategy will be used in a game. The tree is illustrated in Figure 5.1.

The sequence *Run strategy* first checks the condition *!(Game over?)* to check whether the game is finished or not. If it returns success it will run the selector *Choose strategy* that will choose to run one of its children. All these children are link nodes, which will execute other BTs. It can either be *Attack and scout*, *Harass*, *All in*, *Defend* or *Multiple attacks*.

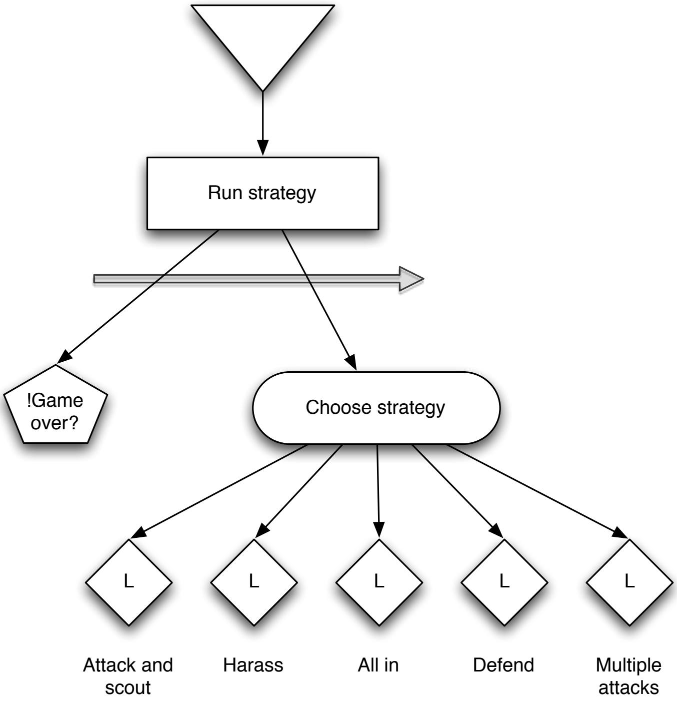


Figure 5.1: Strategy BT.

5.2 Behavior Tree: Attack and Scout

The *Attack and scout* BT depicted in Figure 5.2, shows how the *Attack* and *Scout* BTs are executed simultaneously.

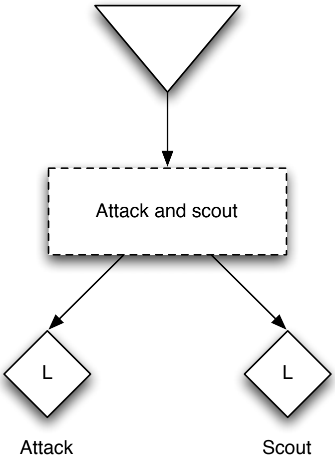


Figure 5.2: Attack and scout BT.

First, the *Attack* behavior allocates 12 marines. It then chooses one attack route among the three possible choices *Top*, *Mid* and *Bottom*, and issues the attack order to the units. The *Scout* behavior orders one unit to each lane, to scout for information about the opponent. Both BTs can be seen in Figure 5.3.

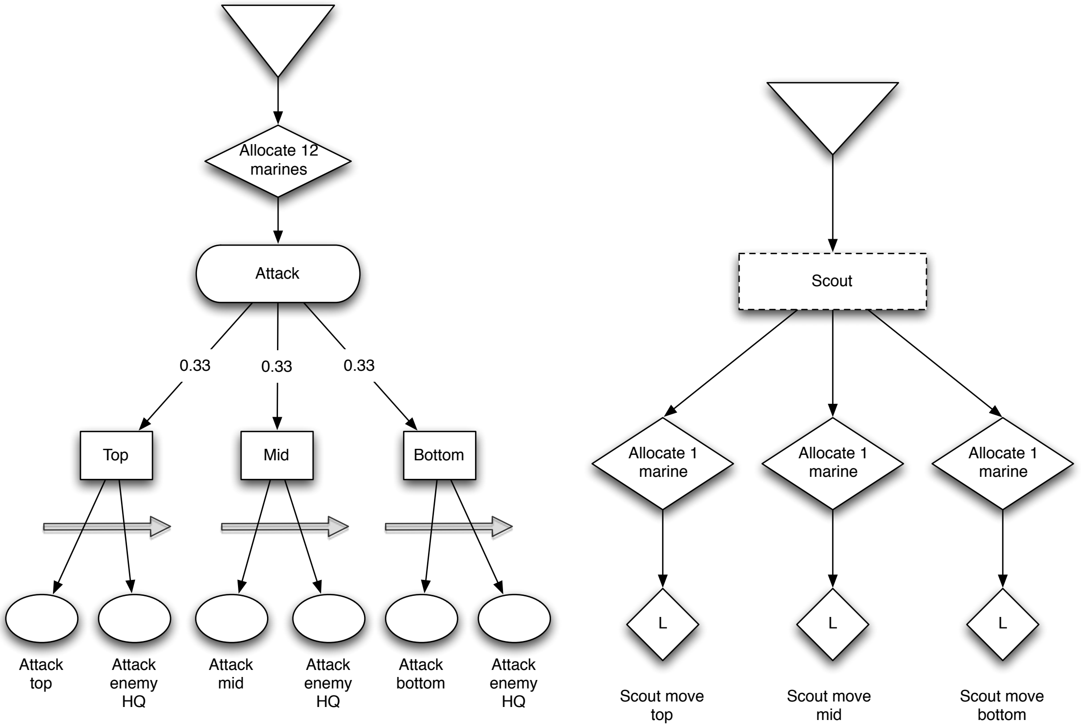


Figure 5.3: The BTs of Attack and Scout

As scouting is vital in our scenario of StarCraft, we need the scouting marines to stay alive as long as possible. Which brings us to the *Scout move top* BT that is given in Figure 5.4. Even though the *Scout* BT links to three different BTs. We choose only to describe one of them, as they are all very similar to each other. The only difference between them is the scouting route for the marine.

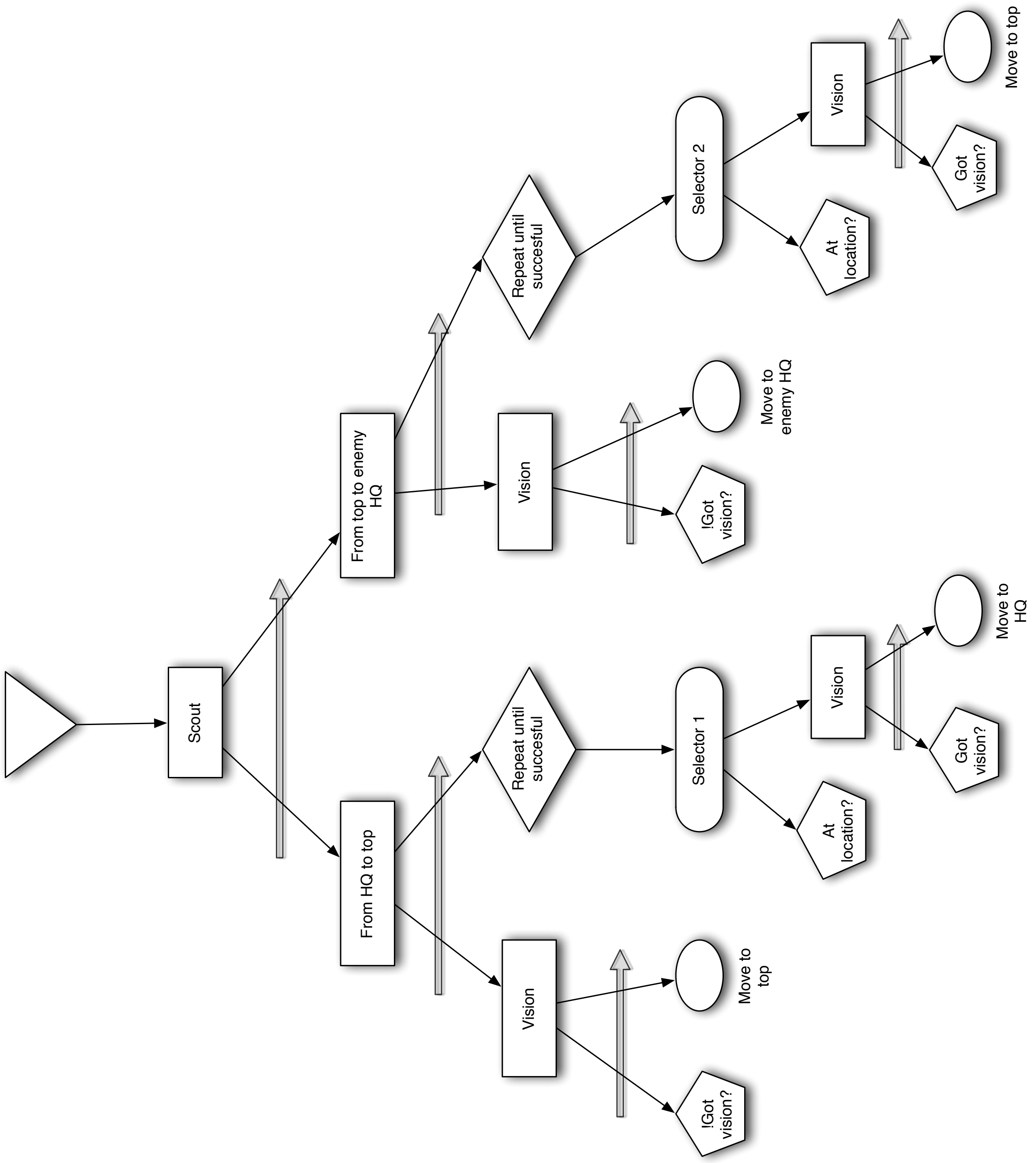


Figure 5.4: Scout move BT.

This tree handles the scouting behavior of a marine . What we want to achieve with these scouting marines, is to get information on the enemy without getting the scoting marine killed. So if the enemy tries to kill the scout, we move away. When the marine is safe, we move in again to get vision of the enemy. As we see in the tree, there are two branches. The left branch handles the route from our HQ to the top left corner of the map, which we call top. The right branch handles the route from top to the enemy's HQ. The marine will move to top if he encounters no enemy units. However, if he gets vision of enemy units on the route to top, he will move back to the HQ. The same applies for the right branch.

The marine will move to the enemy's HQ as long as he encounters no enemy units. If he does encounter enemy units, he will immediately move back to the top.

5.3 Behavior Tree: Harass

The *Harass* BT in Figure 5.5 handles the strategy for harassing the opponent. We allocate six idle marines and order them to harass the opponent through one of the three attack routes Top, Mid or Bottom. Each of these branches in the tree will execute the *Harass move* BT for their respected lane. The *Harass move mid* tree, illustrated in Figure 5.6, handles the subtle details for the harass strategy in Mid path.

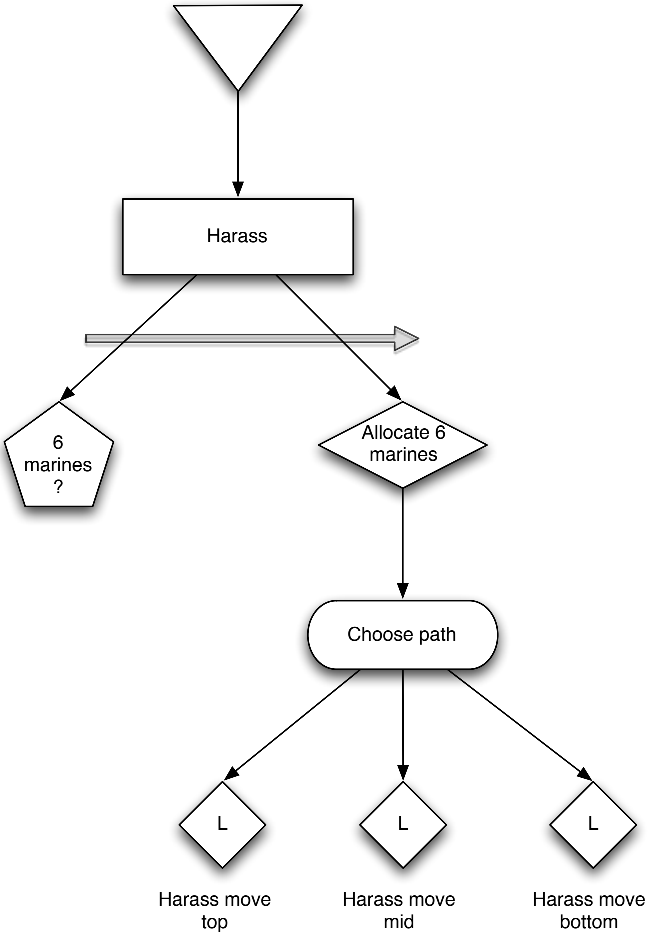


Figure 5.5: Harass BT.

The tree handles it by having a strategy we call *Hit and run*. We want the 6 marines allocated earlier to go down the mid attack route and harass the opponent without too many casualties.

First we check whether the enemy is visible or not, if he is not visible we move our marines towards the opponent's HQ. We then check again for visible enemies, if we are able to see the enemy, we check whether our marines are in range of the enemy. If not, we move in range and proceed to attack the nearest enemy units for one second. Afterwards we retreat from the enemy.

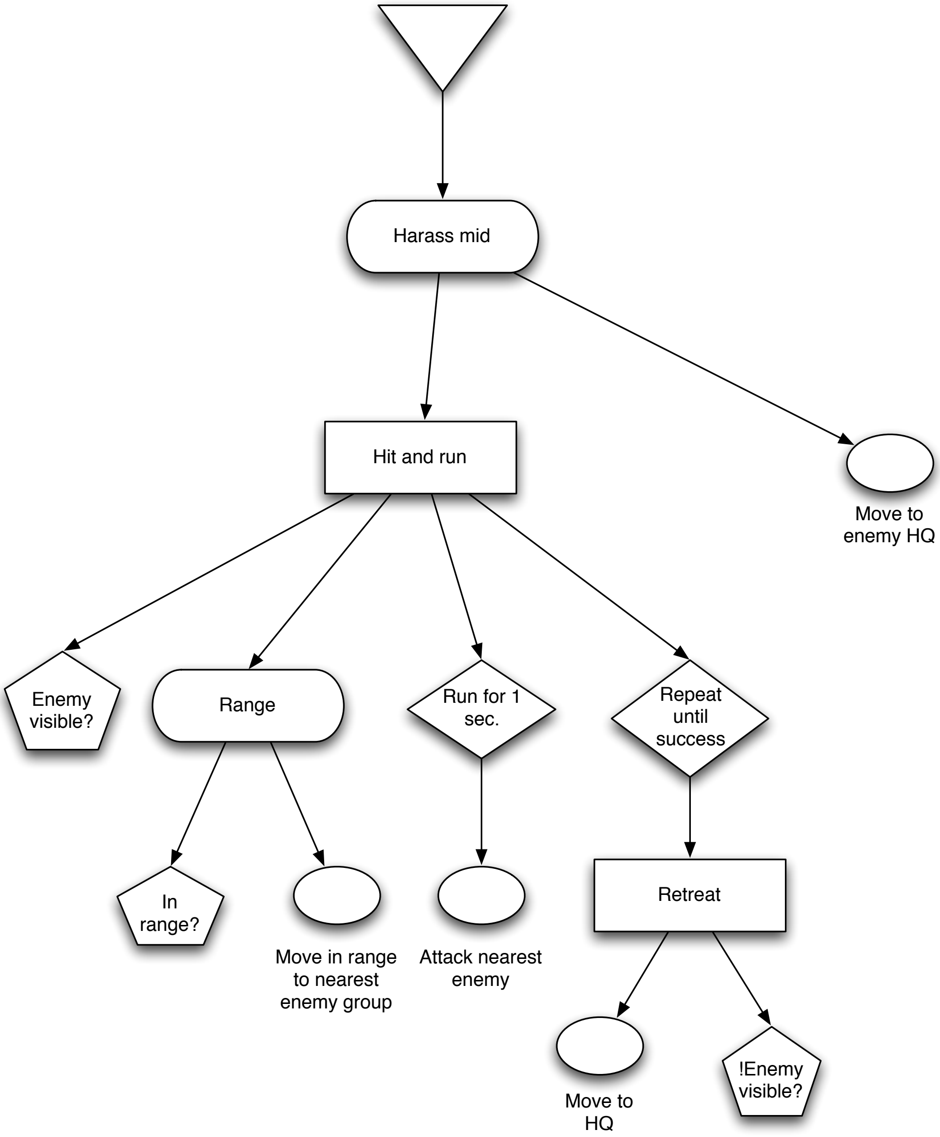


Figure 5.6: Harass attack BT.

5.4 Behavior Tree: All In

The *All in* BT in Figure 5.7 gathers all the units we have and sends them to attack the opponent through the mid lane. This strategy is risky as it leaves the HQ vulnerable to flank attacks from top and bottom. In this example we define that the BT must have 13 marines available. This is a dummy condition to show that it is necessary to have a condition here, otherwise the BT would all in with as little as one marine.

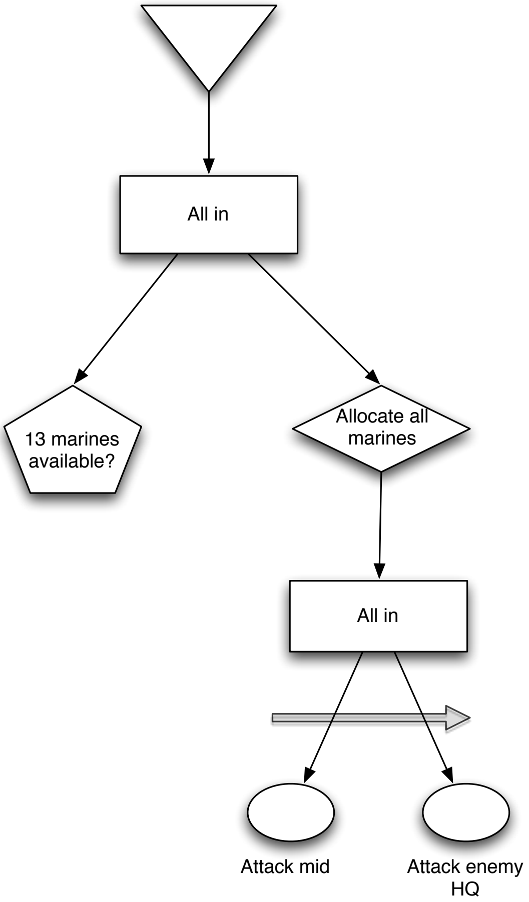


Figure 5.7: All in BT.

5.5 Behavior Tree: Defend

The *Defend* strategy handles the defence of the HQ. The *Defend* tree in Figure 5.8 has two phases of defence. In the first phase, we try to defend by using only nearby units. As a result, units that are already attacking the enemy will keep attacking the enemy, while units nearby our HQ will be gathered in front of our HQ to defend it. The second phase of our defend strategy activates when the first one fails, meaning all the defending marines died. In the second phase we send all units we have on the map back to our HQ to make a last stand.

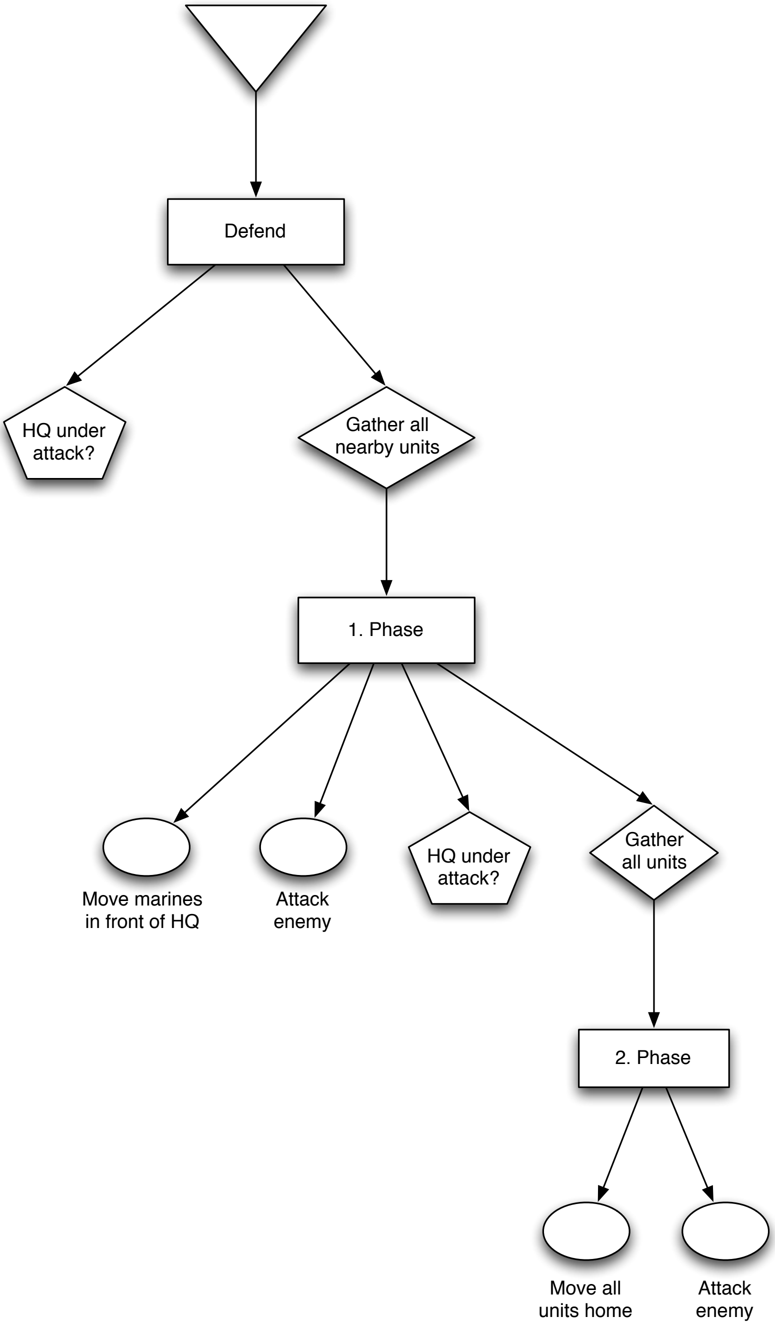


Figure 5.8: Defend BT.

5.6 Behavior Tree: Multiple attacks

The last strategy is the *Multiple attacks* BT, which is illustrated in Figure 5.9. We take our main force and split it into either two or three groups. If we split into two, there are three possible lane options to choose from. We can either attack from top and mid, from top and bottom or from mid and bottom. If we split the army into three, we will attack at all three fronts simultaneously. As we see in Figure 5.9, no matter what branch we choose to run, when we split the army in two, we will end up in a link node. The trees that are linked to are quite simple. The root node is followed by a parallel node, followed by the specific attack route node, ending with the action nodes that issues the attack order. An example of one of these BTs is given in Figure 5.10.

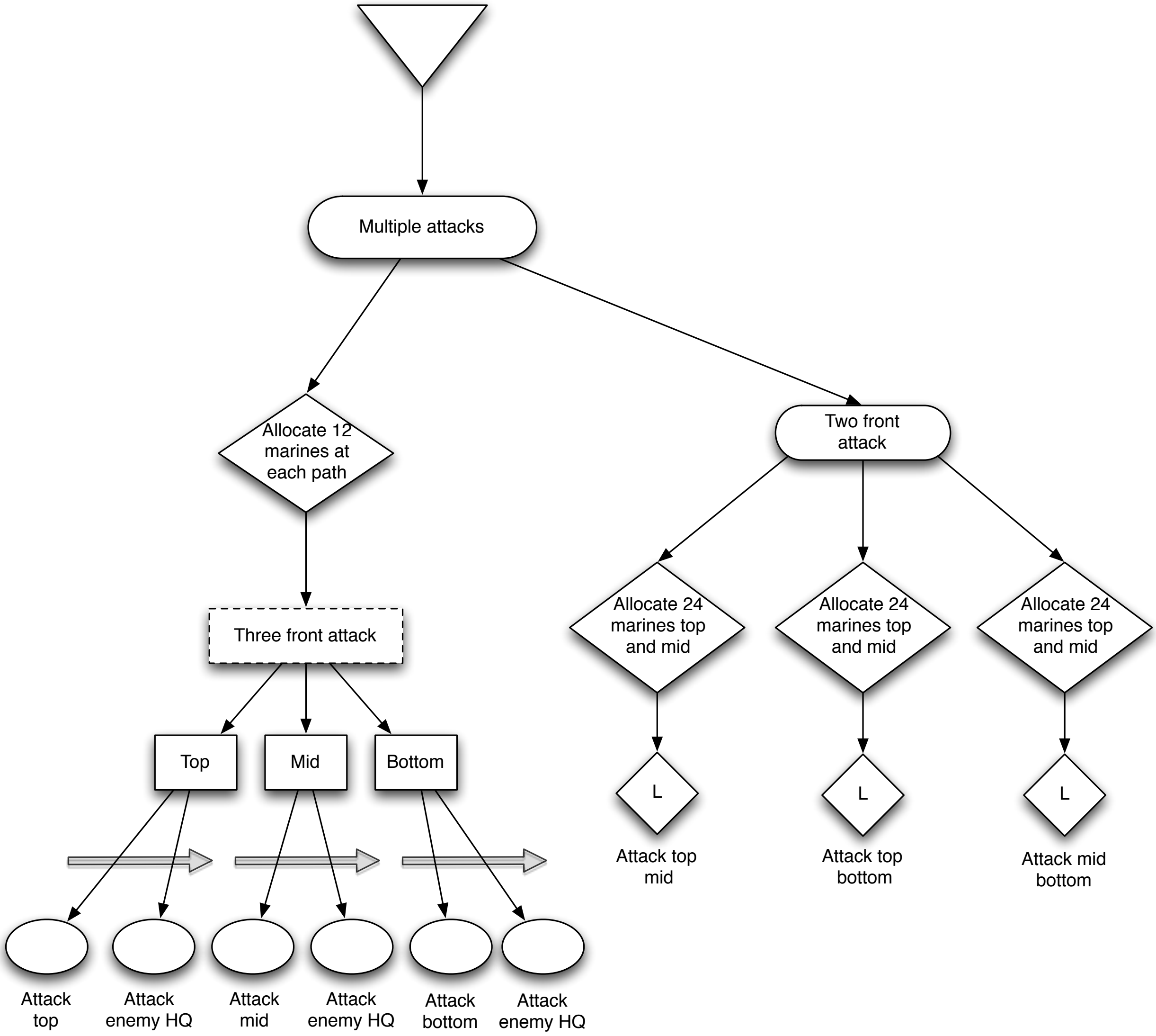


Figure 5.9: Multiple attacks BT.

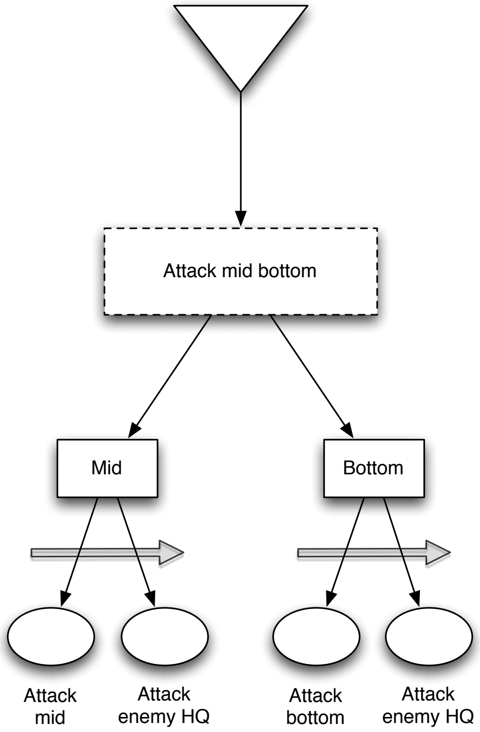


Figure 5.10: Multiple attacks through mid and bottom BT.

6 Test

We have based the tests the BTs proposed in Chapter 5 and use these to show whether it is possible to apply BTs in creating AI for StarCraft and, through this, if BTs can have a general application in RTS games.

We will test each strategy behavior by itself. Based on the performance of the behavior, we will discuss what we expected to happen, what did happen and how to correct possible mistakes in the behaviors. The results of these tests are based purely on our subjective opinion of performance and flaws in the behavior compared to the theory of BTs described in Section 3.3.

Replays from the All In, Harass Attack and Defend tests are available on the attached CD or by request from the authors.

6.1 Test: Attack and Scout

The behavior we want to test here is shown in Section 5.2. The behavior is designed to be a basic strategy where the AI will attempt to attack in one of the three paths while scouting all paths with one marine.

Expected Behavior:

We expect to see two things from this behavior. First off we expect to see the behavior send out three marines; one to scout each path. Second we expect it to assemble 12 marines and attack along one of the paths with an approximate probability of $\frac{1}{3}$ for each path.

Observed Behavior:

We observed some flaws in the scout behavior, mainly due to race conditions on the list of available marines. This is caused by the simultaneous allocation of one marine to each path. Also , when running the attack tree alone, the behavior is capable of attacking along one of the defined paths. When running one of the branches of the scout tree alone, the allocated marine was able to scout the path and run away when encountering enemy units.

Evaluation:

It was possible for the behavior to execute its components individually, but the simultaneous execution proved problematic. We will have to address the problem of simultaneous execution to improve the functionality of the framework.

6.2 Test: All In

The behavior we want to test here is shown in Section 5.4. In this behavior the AI will assemble all available forces and attack through the mid path with full force. The aim is a last resort move when nothing else is possible.

Expected Behavior:

We expect to see that the behavior will gather all available units, which in this case will be 13 marines, and attack through the mid path of the map.

Observed Behavior:

The human opponent attempted to harass the AI, but the behavior executed the all in strategy, when it had gathered 13 marines. The AI attempted to attack through the mid section of the map as intended and eliminated scouts along the path. However, the behavior crashed, when the main battle at the enemy HQ was underway. This was due to a loop inside the attack action node, which iterates through the allocated marines to check whether they are dead. This collection of marines must not be modified during iteration, which was what happened. Again we are seeing problems with the simultaneous execution of the BTs.

Evaluation:

The behavior acted as intended by design, but the crash failure is a definite concern. We will need to investigate solutions for the dynamic handling of marine deaths.

6.3 Test: Harass Attack

The behavior we want to test here is shown in Section 5.3. The harass behavior is the most complex behavior in the test and as such we have only implemented one subbranch of the tree. In this behavior we want to use six marines and attempt to take out enemy marines without losing any marines.

Expected behavior:

We expect to see the behavior gather six marines and run along the mid path. From then on it will attempt to find one common target, the closest to the group, and attack this target for one second before running away.

Observed behavior:

We saw the behavior assembling six marines and attempt to take out one marine at a time. The behavior did the intended hit and run style attack taking out, or damaging, one marine. There were some issues in the implementation, which seem to be caused by the structure of the harass BT.

Evaluation:

This test showed, how hard it can be to create a behavior that mimics a common playstyle of a human player. The complexity made the tree prone to structural errors. The nodes and the framework itself did not show any errors during this test.

6.4 Test: Defend

The behavior we want to test here is shown in Section 5.5. The behavior will, given the HQ is under attack, attempt to move nearby marines back to defend it. If unsucessfull, all units on the map will be moved back.

Expected Behavior:

We expect to see the behavior order all the units closest to the HQ back, in an attempt to defend it, if it is under attack. If all nearby friendly units are killed, we expect to see the behavior order all units back to the HQ in an attempt to defend it.

Observed Behavior:

The behavior acted as intended. When enemy units started attacking the HQ, all nearby units were ordered to attack move towards the friendly HQ, to take out the enemy units. The enemy units were eliminated, however the check to avoid phase two failed to stop the sequence, making the marines attack move towards the mid of the map.

Evaluation:

The behavior did as intended, but structural errors and lack of proper methods in BWAPI to check whether a structure is under attack, made it problematic to implement the intended behavior.

6.5 Test: Multiple Attack

The behavior we want to test here is shown in Section 5.6. This behavior is similar to the attack behavior. The difference lies in the behavior's attempt to attack the enemy on several fronts simultaneously.

Expected Behavior:

The multiple attack behavior has two distinct execution branches. One branch attempts to attack two paths at once and selects one of the three possible pairs of paths. The other execution branch will attempt to assemble enough marines to perform an attack in all three paths simultaneously. We have defined, that no matter which execution path is followed, there must be at least 12 marines allocated to each path, giving 24 marines in a two front attack and 36 in a three front attack.

Observed Behavior:

Observations show that the behavior is able to allocate both two and three groups of 12 marines for use in attacks. We could also observe that the behavior was able to choose a pair of paths for the two front attack. However, due to the way the scenario is constructed and the execution speed of the tree, the three front attack was only chosen when we disabled the two front attack option.

Evaluation:

The test showed that the behavior acted the way it was intended, but as mentioned there were problems choosing between the two types of multiple attack. The reason for the three front attack was not executed, was because it never accumulated the required 36 marines. Due to this, the condition checking whether the marines were available would fail, causing the sequential selector that chooses between the two types of multiple attack, to choose the two front attack, because it is easier to accumulate 24 marines.

This problem is in itself not caused by BTs in general, but rather a problem that must be addressed during the design of the behavior. A solution, could be to introduce a latch that would continue to attempt the selected multiple attack several times, before failing and trying the other one.

7 Improving Framework and Editor

Based on the experience recieved from the previous chapter on testing, we will in this section discuss future improvements for both the framework and the editor.

7.1 Framework Improvements

During the development and later use of our framework several key areas have been identified for improvement.

Interrupts/Exceptions

A key improvement on the current framework would be to implement interrupts and/or exceptions. These should be implemented to make it possible to reset the tree or parts of the tree when certain, perhaps unexpected, events occur. An example of its use is if a marine is being moved as part of an action node execution, the marine might encounter enemies while moving. Here an interrupt could be thrown to allow for a switch in the execution to order the marine to retreat.

As we would like to avoid game API specific code to be in the framework, a solution to implement interrupts, could be to only allow action, condition and decorator nodes to throw interrupts and only allow decorators to handle them. By doing it this way, we can implement a standard way of passing interrupts up the tree, by making the basic constructs, such as sequences and selectors simply pass the interrupt to their parents, until it reaches a decorator that is capable of handling the interrupt. If the interrupt reaches the top of the tree, two things could happen; one could be that the tree throws an unhandled interrupt error informing the game designer that something is wrong, or that the tree simply resets itself and restarts execution.

Blackboard

After using the blackboard in its current state, where each BT has its own blackboard, we have come to realize this is not an optimal solution. Since this imposes a lot of extra work during coding to make sure which blackboard each value is written. The process of making the blackboard simple will not have an impact on the overall functionality of the BT framework.

We would also like to make the blackboard type safe by ensuring that the requested value has the expected type when executing the blackboard, rather than checking the value, when it has been returned.

BehaviorTreeThreadWrapper

After using the framework for testing, we have come to the conclusion that the BT thread wrapper can be merged into the BT class itself. There is no need for a separate class. One of the things that we have concluded is that it is always necessary to have a barrier and as such this should be defined in the tree class itself. If the barrier is in the tree class, the execution, pausing and resuming of the tree can then also be merged into the class. This makes it possible to use both threaded and unthreaded BT execution from the same class.

More basic constructs

The framework could benefit from more basic constructs, such as builtin loop decorators that either iterate a number of runs, or a number of successes/failures. Another basic construct that could be useful is a wait decorator capable of waiting a given amout of time, before executing its child.

Debugging directly in the IDE

As a way of making it easier to debug constructed BTs, we would like to add the ability to debug the projects directly in an IDE, such as Microsoft Visual Studio. One solution to this could be, with permission from the author, to include the bwapi-mono-bridge projects directly in a solution, so that compiling and debugging can happen in Microsoft Visual Studio.

7.2 Editor Improvements

The current implementation of the editor does not have functionality to work together with the framework, as a few features are still missing.

Exporting Behavior Trees

It is not possible to export the BTs created in the editor to the framework, which is a key feature for the editor to be of actual use. Exporting the BTs to e.g. XML format would allow us to load it into the framework, by parsing the XML code and then constructing the BTs automatically. Currently BTs have to be coded manually in the framework, which takes considerably longer time, compared to building them in the editor and exporting them to the framework.

Node Properties

Other than the difference in the visual appearance of nodes, there are no distinct properties for the individual nodes in the editor. It is not possible to set the probability distribution for a probaility selector, edit the contents of a decorator node, defining which BT a link node links to e.g. To accommodate this, a property editor, that becomes visible when individual nodes are selected, should be implemented.

Multiple Canvases

The editor only allows for one BT to be built, because there is only one canvas with one root node. Being able to have multiple canvases open at once will remove this limitation. This can be implemented either as different canvas tabs, like webpage tabs in webbrowsers, or having multiple windows openend, each with a canvas. Working on multiple canvases at once, would also allow use of the link nodes to link to a BT on another canvas.

8 Conclusion

To summarize the work done in this project. We have:

- Discussed the game of StarCraft to address some of the challenges, creating an entertaining and challenging AI for RTS games.
- Presented a survey of scripting, FSM and BTs on their applicability in RTS games.
- Implemented a framework and editor for using BTs in behavior creation.
- Successfully applied BTs, creating an AI for an RTS game scenario.

First of all by addressing the different aspects of StarCraft, we highlighted the challenges, creating an AI for an RTS game. The overall micro management and macro management aspects of StarCraft, also exist in most other RTS games. The scenario created for the case studies served well as a simplification of a full game scenario, as it captured the important aspects of the game. The elements of resources and base building were left out of the scenario to simplify it further, such that we could focus on strategic decision making and disregard build orders and resource management.

As the evaluation of the different AI methods pointed out, scripting and FSM had their strength and weaknesses. BTs attempt to address these weaknesses, by being more structured and modular. The modularity and reusability of BTs showed its usefulness, when all the framework nodes and nodes interacting with the game environment were implemented. With all the building blocks ready, it was just a matter of building behaviors, according to the trees designed in Chapter 5.

Through tests of the proposed BTs we saw an indication that BTs can be used for creating AIs for RTS games. There are of course problems that are unique with regards to the BT and RTS game combination. Amongst these, the most apparent seems to be the handling of the fact that units may or may not exist during execution of the behavior, as opposed to a behavior driven NPC player in an FPS game.

After doing the tests, we discovered several key points for improving the BT framework. The most important problems discovered, concerns the threading execution of the trees. It introduces several possibilities for race conditions and deadlocks. These problems are of a general concern, when using threading, but it is necessary to look at solutions, to minimize the risk of encountering them. Additionally, the editor still needs further improvement for it to be functional in combination with the framework. It is currently not possible to export BTs from the editor and use them in the framework.

8.1 Further Work

The work done in this report introduces several possibilities for further study in the area of BTs.BTs have been shown, to be a possibly viable method, to handle the AI in an RTS game, namely StarCraft. However, as with the commonly used methods for AI in the game industry, scripting and FSMs, BTs only exhibit static behavior.

The framework and editor implemented and documented in this report, serve as a great foundation for developing and utilizing BTs in StarCraft. As pointed out in Chapter 7 there is still room for much improvement of both the framework and editor.

Based on the work done we propose two general suggestions for further study and research.

1. How can the framework and the editor be combined and improved to fully utilize the notion of BTs in game AI and assist in making the creation of BTs less complex?
2. Can the notion of BTs be extended with classical AI techniques to create more adaptive and human-like behavior.

As we described in Chapter 3, the goal of game AI devlopment, is to provide entertaining and challenging AI opponents, that display intelligence and human-like behavior. In other words the ideal goal of game AI research is to create AI which is indistingusiable from a human. Such AI would give players the best possible experience when playing. As Alan Turing once pointed out:

"If a machine acts as intelligently as a human being, then it is as intelligent as a human being."

We are still a long way from achieving this goal, but as new methods and theories emerge, we come ever so close to reaching this goal.

A Classdiagram

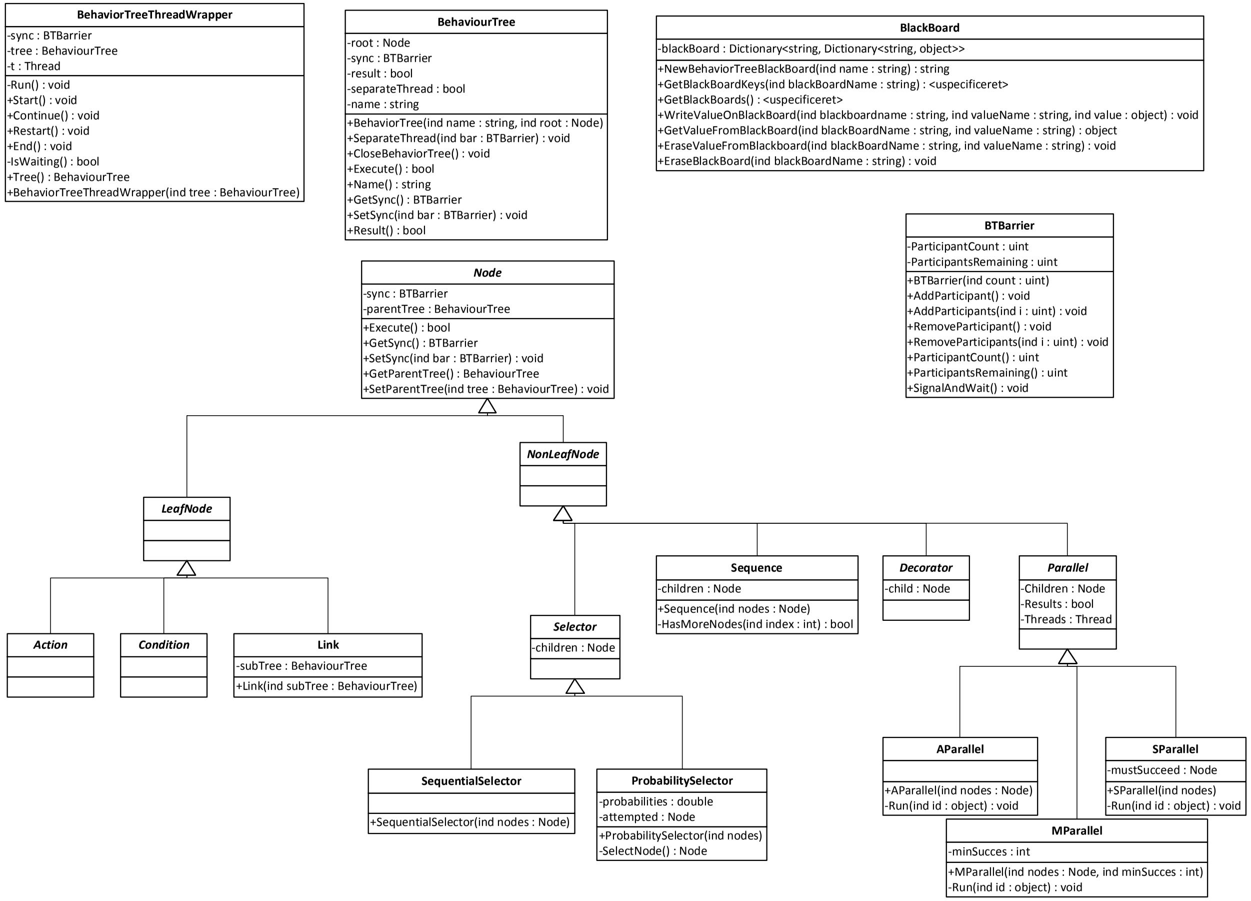

BehaviorTreeThreadWrapper
-sync : BTBarrier
-tree : BehaviourTree
-t : Thread
-Run() : void
+Start() : void
+Continue() : void
+Restart() : void
+End() : void
-IsWaiting() : bool
+Tree() : BehaviourTree
+BehaviorTreeThreadWrapper(ind tree : BehaviourTree)
BehaviourTree
-root : Node
-sync : BTBarrier
-result : bool
-separateThread : bool
-name : string
+BehaviorTree(ind name : string, ind root : Node)
+SeparateThread(ind bar : BTBarrier) : void
+CloseBehaviorTree() : void
+Execute() : bool
+Name() : string
+GetSync() : BTBarrier
+SetSync(ind bar : BTBarrier) : void
+Result() : bool
BlackBoard
-blackBoard : Dictionary<string, Dictionary<string, object>>
+NewBehaviorTreeBlackBoard(ind name : string) : string
+GetBlackBoardKeys(ind blackBoardName : string) : <uspecificeret>
+GetBlackBoards() : <uspecificeret>
+WriteValueOnBlackBoard(ind blackboardname : string, ind valueName : string, ind value : object) : void
+GetValueFromBlackBoard(ind blackBoardName : string, ind valueName : string) : object
+EraseValueFromBlackboard(ind blackBoardName : string, ind valueName : string) : void
+EraseBlackBoard(ind blackBoardName : string) : void
BTBarrier
-ParticipantCount : uint
-ParticipantsRemaining : uint
+BTBarrier(ind count : uint)
+AddParticipant() : void
+AddParticipants(ind i : uint) : void
+RemoveParticipant() : void
+RemoveParticipants(ind i : uint) : void
+ParticipantCount() : uint
+ParticipantsRemaining() : uint
+SignalAndWait() : void
Node
-sync : BTBarrier
-parentTree : BehaviourTree
+Execute() : bool
+GetSync() : BTBarrier
+SetSync(ind bar : BTBarrier) : void
+GetParentTree() : BehaviourTree
+SetParentTree(ind tree : BehaviourTree) : void
LeafNode
NonLeafNode
Action
Condition
Link
-subTree : BehaviourTree
+Link(ind subTree : BehaviourTree)
Selector
-children : Node
Sequence
-children : Node
+Sequence(ind nodes : Node)
-HasMoreNodes(ind index : int) : bool
Decorator
-child : Node
Parallel
-Children : Node
-Results : bool
-Threads : Thread
SequentialSelector
+SequentialSelector(ind nodes : Node)
ProbabilitySelector
-probabilities : double
-attempted : Node
+ProbabilitySelector(ind nodes)
-SelectNode() : Node
AParallel
+AParallel(ind nodes : Node)
-Run(ind id : object) : void
SParallel
-mustSucceed : Node
+SParallel(ind nodes)
-Run(ind id : object) : void
MParallel
-minSucces : int
+MParallel(ind nodes : Node, ind minSucces : int)
-Run(ind id : object) : void

Figure A.1: Class diagram for the Poker test environment

B Execution Flow

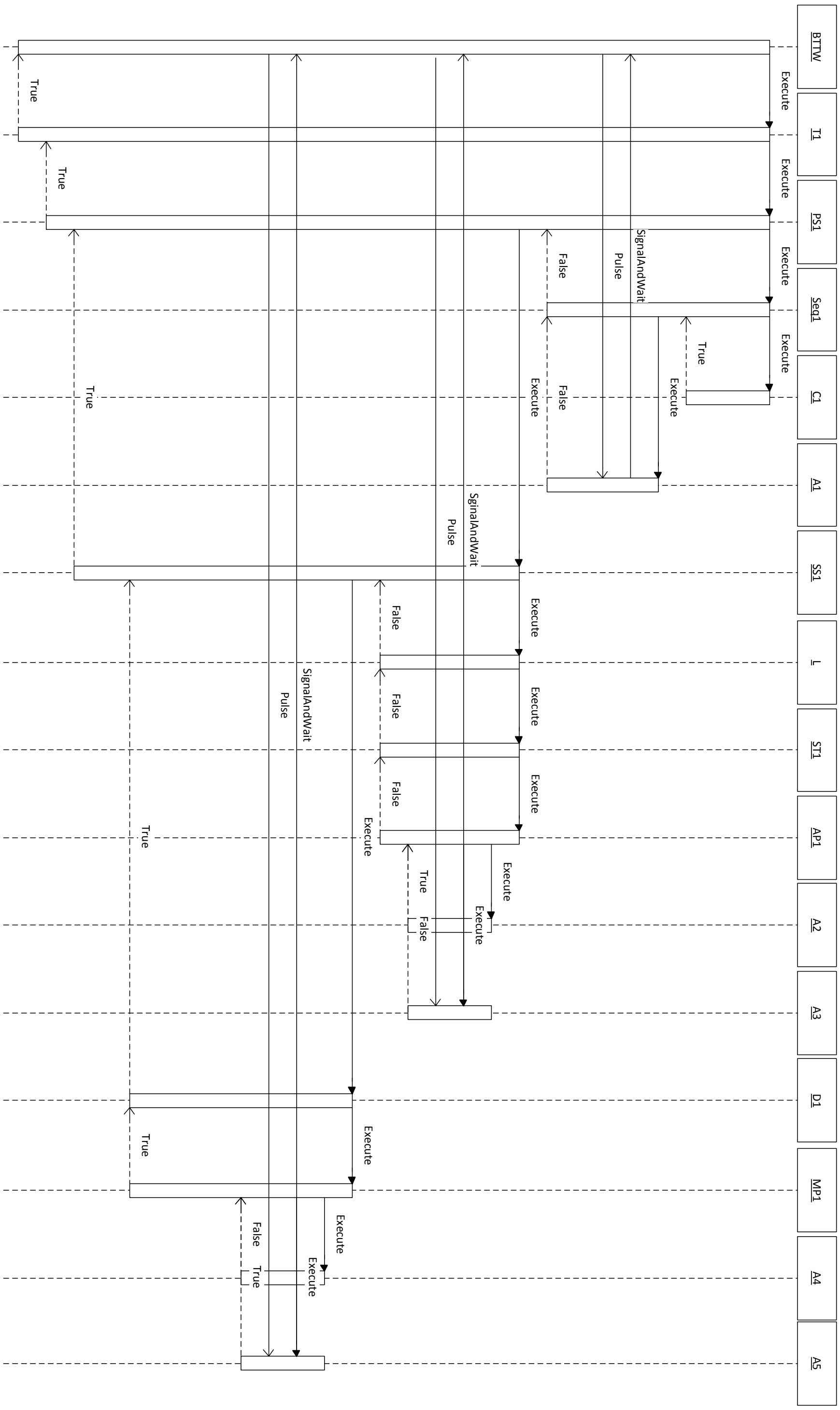


Figure B.1: Sequence diagram showing the execution of a behavior tree

Bibliography

[1] Games engines - current offerings, comparisons and research. http://www.halycopter.com/uni/gamesengines.pdf.

[2] Blizzard Entertainment. Starcraft: Brood war, 1998.

[3] Blizzard Entertainment. Starcraft 2 editor, 2010.

[4] Blizzard Headquarters Web Team. Starcraft compendium. http://classic.battle.net/scc/, 2010. 12. December.

[5] A. J. Champandard. The gist of hierarchical fsm. http://aigamedev.com/open/articles/hfsm-gist/, 2007.

[6] l. deathknight13579. Bwapi. http://code.google.com/p/bwapi/, 2010. 25. November.

[7] dpershouse. bwapi-mono-bridge. http://code.google.com/p/bwapi-mono-bridge/, 2010. 25. November.

[8] M. Dyckhoff. Evolving halo's behaviour tree ai. http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/gdc07.pdf.

[9] C. Hecker. My liner notes for spore/spore behavior tree docs. http://chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs, 2009.

[10] D. Isla. Handling complexity in the halo 2 ai. http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml, 2005.

[11] LucasArts, Telltale Games. Monkey island series, 1990.

[12] P. T. Mallik Challab, Dana Nau. *Automatic Planning - Theory and Practice*. Number ISBN: 1-55860-856-7 in 1st Edition. Morgan Kaufmann, 2004.

[13] S. S. H. Mance E. Harmon. Reinforcement learning: A tutorial, 1996.

[14] MicroProse. Sid meier's civilization, 1991.

[15] Midway Games. Mortal kombat, 1992.

[16] H. Muñoz-Avila. Game genres: First quick look. http://www.cse.lehigh.edu/ munoz/ComputerGameDesignClass/classes/FSMandScripts.pptx.

[17] Namco. Tekken, 1994.

[18] J. Orkin. Three states and a plan: The a.i. of f.e.a.r., 2006.

[19] J. Orkin. Goal-oriented action planning. http://web.media.mit.edu/j̃orkin/goap.html, 2010.

[20] R. Straatman. The ai in killzone 2's bots: Architecture and htn planning. http://aigamedev.com/premium/presentations/killzone2-planning/, 2010.

[21] P. N. Stuart Russell. *Artificial Intelligence - A Modern Approach.* Number ISBN: 0-13-207148-7 in Third Edition. Pearson, 2010.

[22] Sukram. Diagram designer. http://www.codeproject.com/KB/WPF/WPFDiagramDesigner_Part1.aspx, 2008.

[23] A. Tankred and H. M. Bøgeskov. Ai modelling: Behaviour trees. Technical report, 2010.

[24] Team Ninja. Dead or alive, 1996.