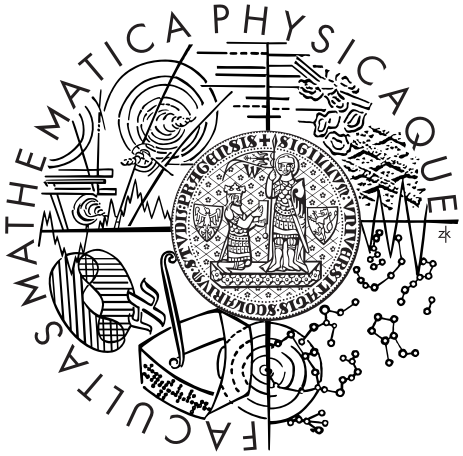
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Mikuláš Zelinka

Using yaPOSH for CTF team behaviour

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Michal Bída

Study programme: Computer Science

Specialization: Programming

Prague 2014

I would like to thank to my supervisor, Mgr. Michal Bída, for his patience and invaluable advice.

I would also like to thank to Mgr. Jakub Gemrot for helping me with resolving various Pogamut-related issues and to Bc. Vojtěch Tuma for helping me with getting his bots to work.

Finally, my thanks belong to my family and to Anna.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Specifikace týmového chování pro CTF pomocí yaPOSHe

Autor: Mikuláš Zelinka

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Michal Bída

Abstrakt: Vyhodnotili jsme vhodnost použití yaPOSHe (nástroje pro plánování virtuálních agentů) pro specifikaci složitého týmového chování v módu Capture the Flag (CTF) hry Unreal Tournament 2004. Pomocí yaPOSHe jsme vytvořili tým botů do CTF a porovnali jej s týmem napsaným V. Tumou v jazyce Java a rovněž s boty od autorů hry.

Ukázalo se, že yaPOSH má oproti samotné Javě řadu výhod (zejména co se týče čitelnosti kódu), nicméně není možné vytvořit kvalitní boty pouze s jeho výhradním použitím. Důvodem je zejména absence podpory paralelního vyhodnocování yaPOSH plánů. Proto musely být části chování (např. řešení soubojů) naprogramovány v Javě. Výsledný CTF tým je ale i tak znatelně lepší než původní boti autorů hry a ve většině případů lepší než bot V. Tumy.

Na základě těchto poznatků jsme navrhli několik vylepšení jak vyhodocovacího enginu yaPOSHe, tak i editoru jeho plánů. Výrazně by pomohlo umožnit paralelní vyhodnocování plánu nebo povolit vyhodnocování několika různých plánů jednoho agenta najednou. Naše návrhy a doporučení se neomezují na tento konkrétní mód a konkrétní hru, nýbrž jsou relevantní i pro specifikaci jiných složitých týmových chování pomocí yaPOSHe.

Klíčová slova: Umělá inteligence, yaPOSH, Pogamut, Unreal Tournament 2004, Capture the Flag

Title: Using yaPOSH for CTF team behaviour

Author: Mikuláš Zelinka

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Michal Bída

Abstract: We evaluated the suitability of yaPOSH (an action-selection system for virtual agents) for complex team behaviour development, specifically for the Capture the Flag (CTF) mode in Unreal Tournament 2004. We created a CTF team using yaPOSH and Java and compared them with a CTF team written by V. Tuma in plain Java as well as with the native UT2004 bots.

We found out that although yaPOSH does have some advantages over plain Java (mainly in code readability), one cannot create a competitive bot using yaPOSH only. That is a direct consequence of the limitations the yaPOSH planner has, with the most significant one being lack of parallelism support. Thus, some aspects of the behaviour (such as combat) were programmed in Java. Nevertheless, the resulting team is better than the native bots by a margin and slightly better than Tuma's CTF team.

As a result, we have made several suggestions for improvements of the yaPOSH engine as well as its editor, such as to add the possibility to execute multiple yaPOSH plans at once, or to enable their parallel evaluation. These suggestions and our other findings are not limited to the chosen domain of UT2004's CTF mode and instead are relevant to specification of any other complex team behaviour.

Keywords: Artificial Intelligence, yaPOSH, Pogamut, Unreal Tournament 2004, Capture the Flag

Contents

1 Introduction **4**

1.1 Unreal Tournament 2004 4

1.2 Capture the Flag 5

1.3 Pogamut 6

1.4 yaPOSH and Behaviour Oriented Design 7

1.5 Goal of the thesis 8

1.6 Structure of the thesis 8

2 yaPOSH and Behaviour Oriented Design **10**

2.1 Behaviour Oriented Design 10

2.2 yaPOSH plan 11

2.2.1 Primitives 11

2.2.2 Aggregates 11

2.2.3 Plan example 12

2.3 Comparison to Java 14

2.3.1 Code readability 14

2.3.2 Parallelism and Expressive power 14

3 Behaviour specification **16**

3.1 Desired behaviour 16

3.2 Behaviour decomposition 17

3.3 Movement decision making 18

3.3.1 Navigation in Pogamut 18

3.4 Combat 19

3.5 Communication 20

3.6 Team strategy and roles 21

4 CTF team implementation **22**

4.1 Bot update cycle 22

4.2 Roles 23

4.2.1 Attacker 23

4.2.2 Defender 23

4.2.3 Roamer 24

4.2.4 Role distribution and switching 25

4.3 Behaviour modules 25

4.3.1 Communication 26

4.3.2 Combat 27

4.3.3 Movement decision making 28

5 Experiments **31**
5.1 Opponents 31
5.1.1 Native bots 31
5.1.2 V. Tuma's bots 31
5.2 Match settings 32
5.2.1 Difficulty level 32
5.2.2 Weapons 32
5.2.3 Match limits 32
5.2.4 Team size 32
5.2.5 Translocator 33
5.2.6 Mutators 33
5.2.7 Friendly fire 33
5.2.8 Settings overview 33
5.3 Maps 34
5.3.1 CTF-Citadel 34
5.3.2 CTF-Geothermal 35
5.3.3 CTF-Lostfaith 36
5.4 Format 36

6 Results **37**
6.1 Native bots 37
6.1.1 CTF-Citadel 37
6.1.2 CTF-Geothermal 38
6.1.3 CTF-Lostfaith 38
6.2 Tuma's bots 39
6.2.1 CTF-Citadel 39
6.2.2 CTF-Geothermal 40
6.2.3 CTF-Lostfaith 40
6.2.4 Additional matches 41
6.3 Summary 41

7 Discussion **43**
7.1 Decision making versus coding 43
7.2 Importance of tool's specifics 43
7.3 Different techniques for different tasks 44
7.4 When to use yaPOSH 44
7.5 When not to use yaPOSH 45
7.6 Improvement suggestions 45

7.6.1 Parallelism . . . 46
7.6.2 Plan variables . . . 46
7.6.3 Cosmetic changes . . . 46
7.7 Final thoughts . . . 47

8 Conclusion **48**

Bibliography **49**

List of Abbreviations **51**

Appendix A: Resulting yaPOSH plan **52**
Syntax . . . 52
Drive collection . . . 53
The main Attacker competence . . . 53
The main Defender competence . . . 53
The main Roamer competence . . . 54
Competences . . . 54
Attacker's competences . . . 54
Defender's competences . . . 56
General competences . . . 57

Appendix B: CD contents **59**

Appendix C: User guide **60**
Installation . . . 60
Starting the server . . . 60
Running our bots . . . 60
Modifying our bots . . . 61
Running native bots . . . 61

1. Introduction

Recently, there has been a significant increase in demand for artificial intelligence (AI) in the computer games industry. Computer games come in a number of different genres and each of these does have specific characteristics and different needs for its AI. All the games, however, do have a common goal — to create an illusion of intelligence and human-like behaviour of the in-game characters in order to increase the game's believability, making it consequently more enjoyable to play. Our task will be to explore the possibilities of one of the AI techniques by creating an AI team of co-operating virtual characters.

Intelligent virtual agents (IVAs)[1] are virtual human-like characters that try to exhibit and imitate human-like behaviour. Naturally, creating a truly human-like virtual agent is a rather difficult task and there are many different ways to approach the whole matter.

Our goal is to create a team of intelligent virtual agents using a specific approach, Behaviour Oriented Design (BOD, introduced in 2001 by J. J. Bryson [3]), and a specific tool based on BOD, the yaPOSH reactive planner [10]. We would like to find out whether this approach and this tool are suitable for developing complex team behaviour and possibly identify their advantages, limitations and tasks that could (or alternatively could not) be fittingly solved using this method.

Our team will play the Capture the Flag (CTF) game mode of Unreal Tournament 2004 (UT2004). We chose this domain because its IVA development is supported by the Pogamut platform [12] and some CTF teams based on Pogamut have already been created. We will use these teams for comparison.

In this chapter, we will introduce the platforms and techniques we will use in the thesis, discuss the goal of the thesis and then describe its structure. We will start by introducing the environment of UT2004, which we are using for the IVA simulation.

1.1 Unreal Tournament 2004

Unreal Tournament 2004 is a first-person shooter computer game developed by Epic Games in 2004, in which each player controls a virtual character in a virtual 3D environment. The in-game characters can also be controlled by computer, in which case they are commonly called *bots*.

The game features several game modes which share some aspects of the gameplay, such as collecting weapons, shooting and moving through the map. Some

[1] We will also call them *agents* or *bots*.

Figure 1.1: A screenshot from UT2004's CTF mode on map CTF-Lostfaith.

game modes revolve solely around only these game mechanics with focus on combat, while others, such as Capture the Flag, do have other important objectives on top of the combat system. However, the combat system is still present in the CTF game mode and our objective is to come up with solid CTF decision making *on top of* specifying good combat behaviour.

Next we will describe the specific mechanics of the game mode we will be creating our bot team for, Capture the Flag.

1.2 Capture the Flag

Capture the Flag is one of the modes featured in UT2004 as well as a traditional outdoor game. In the game, two teams battle against each other for the most important objectives, *flags*, in a certain game environment (a map[2]). In UT2004, there are multiple maps for most game modes including CTF. Teams can have different number of players, but the standard team size and the size we will use is five players on each team.

On each map, a team has a base and a flag hidden within it. The goal is to capture your enemy's flag and bring it in to your own base while defending your own flag, meaning that a team cannot score when its own flag is not in its base.

In UT2004's CTF mode, each flag capture is worth one point and there is

[2]In FPS games, the areas in which the simulation takes place are commonly called *maps* and matches can be played in several different game environments, maps.

a score limit and a time limit. The game ends either when a team reaches the score limit or when the time expires. This is important because it makes certain strategies viable, for example focusing on defence when already in a lead or trying an all-out attack when losing while time is running out.

As mentioned above, the fundamental characteristics of UT2004 are the item and the combat systems. A character can collect items (such as weapons, ammunition or health packs) around the whole map and use them in combat against other characters (or, in case of CTF, against members of the opposing team). Every character has a *health* attribute that decreases when the character is hit by an enemy. When the health reaches zero, the character dies. In the game, death is not permanent. When killed, the bot or the player respawns in a predefined spot but without the items he had previously collected. More information about items and combat can be found in section 3.4.

We will focus on matches where all characters will be controlled by artificial intelligence, i.e. bots.

1.3 Pogamut

Pogamut[3] [12] (derived from POSH, Games, Unreal Tournament) is a Java middleware platform that enables its users to develop and control IVAs in several games or game engines including UT2004. The platform is designed to simplify the agent creation process and it provides a Java API with a wide variety of high-level functions, meaning that even complex tasks, such as computing a navigation path to a certain point, often require no more than a single command. As a consequence, one can then fully focus on high-level decision making without having to worry about how precisely the decisions are carried out on lower levels.

To some extent, the platform enables us to tell the bot *what* to do, instead of *how exactly* to do it.

Pogamut also comes with a NetBeans[4] plug-in that includes a number of useful graphical and debugging tools for controlling the servers and the bots in the game environment.

The platform is using the following architecture (see fig. 1.2). Data from the game (UT2004) is sent with TCP/IP using the GameBots2004 text protocol, processed by Gavialib and converted to Java objects. The end user then works only with these objects and can also debug the agents with the help of the Pogamut NetBeans plug-in.

We are using Pogamut as a tool that allows us to concentrate on implementing

[3]Pogamut is available at `http://pogamut.cuni.cz/`.

[4]NetBeans is an integrated development environment for developing primarily in Java.

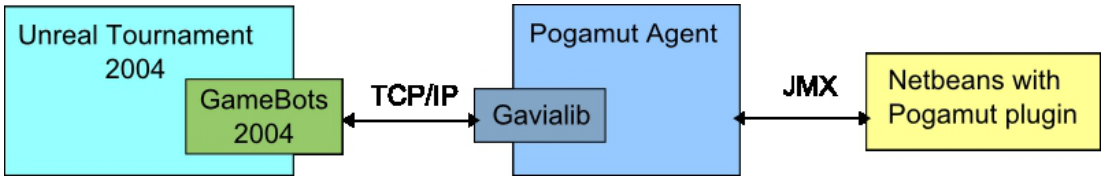


Figure 1.2: Diagram representing the architecture of Pogamut.

the actual in-game decision making of our bots rather than on the necessary but not necessarily that interesting technical problems such as communication with and connection to the game engine.

Pogamut also contains the yaPOSH planner, a tool for developing strategies for bots whose philosophy is based on Behaviour Oriented Design.

1.4 yaPOSH and Behaviour Oriented Design

Behaviour Oriented Design (BOD) [2] is a methodology focused on human-like development of IVAs. BOD uses a human-like approach of behaviour decomposition into behaviour modules and its concept is used by many planners, one of which is the yaPOSH planner [10].

The yaPOSH planner is an action-selection system tailored for the development of IVAs in UT2004. It is a dialect of POSH [3] and a part of the Pogamut platform. One might view yaPOSH as an implementation of Behaviour Trees [4] since its elementary component, the yaPOSH plan, is used for action selection and does have a tree-like structure.

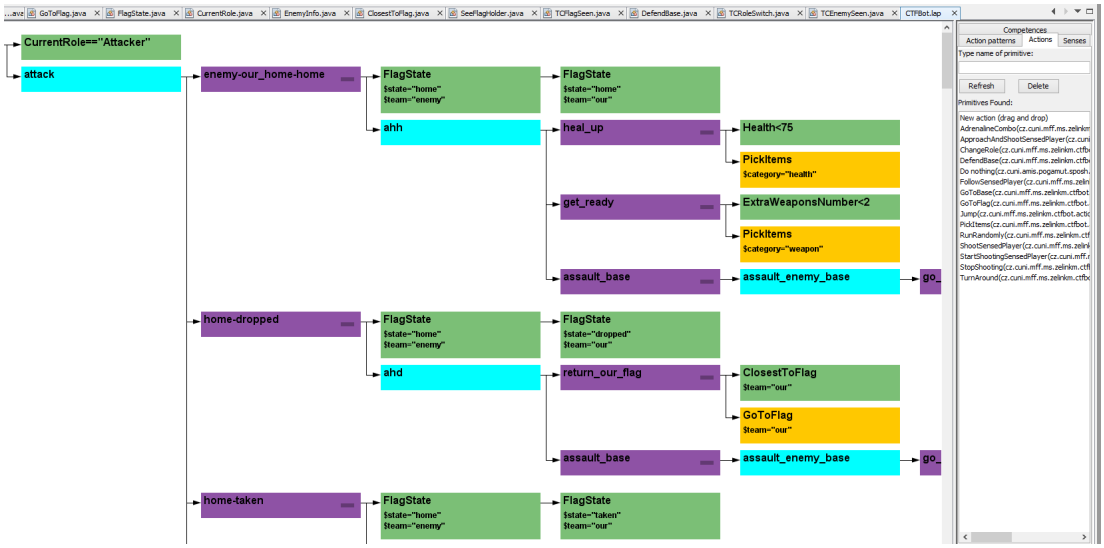


Figure 1.3: A part of a yaPOSH plan in Shed.

yaPOSH uses a human-like approach to behaviour specification since it is based on BOD and the key components of its plans are *senses* and *actions* that

correspond to human senses and actions. Each bot has a single yaPOSH plan assigned to him, and based on what he senses or perceives, he carries out a certain action.

Pogamut's recent versions also feature Shed, a graphical editor for yaPOSH (or to be precise, for its plans) and its graphical debugger, Dash. These tools as well as yaPOSH itself were introduced as a part of a diploma thesis of Jan Havlíček [13].

yaPOSH, yaPOSH plans and the philosophy of BOD are described in the next chapter 2 in detail.

1.5 Goal of the thesis

The goal of the thesis is to create a CTF bot team for UT2004. The resulting team should perform *well*, meaning that we want our team to perform at least as well or better than the native[5] bots and that our team should display human-like behaviour, which is the usual requirement when creating IVAs for any computer game, since human players do demand and enjoy human-like in-game characters. See section 3.1 for more detailed specification of the desired behaviour.

Secondary goal is to evaluate the suitability of the BOD approach and the yaPOSH planner for complex team behaviour specification and to show its advantages and disadvantages in comparison to plain Java. At the end, we would like to be able to conclude whether these tools should or should not be used for solving similar tasks and more importantly, why that is the case.

1.6 Structure of the thesis

First of all, we will introduce BOD as a concept of modelling human-like behaviour and introduce the yaPOSH planner based on BOD. Then we will describe the desired CTF behaviour and decompose it into smaller parts using the BOD approach introduced previously.

Afterwards, we will implement the behaviour using the yaPOSH tool with the help of the decomposition mentioned beforehand.

Our resulting CTF team will then be compared against other bots programmed previously as well as against the native UT bots. We will mainly focus on the differences in in-game performance and code readability.

Finally, based on this comparison, we will appraise the suitability of yaPOSH for solving similar tasks, come up with suggestions on how the yaPOSH planner

[5]Native bots are bots created by the authors of the game.

could be improved or what features could be added in the future.

More detailed description of the chapters follows.

In chapter 2, we will describe BOD and the yaPOSH reactive planner based on BOD in detail. Its characteristics will be shown mainly with regard to the behaviour specified in chapter 3. We will also discuss the differences between this using the yaPOSH planner and plain Java and the consequences of these differences.

In chapter 3, we will introduce the desired bot behaviour and decompose it into different parts and describe these individually in detail.

In chapter 4, we will present our CTF strategy. We will show how the update cycle of Pogamut bots works and introduce our implementation of the CTF team based on the chosen strategy and Pogamut's update cycle. We will also describe how we implemented each of the behaviour parts in detail and present the resulting yaPOSH plan.

Chapter 5 specifies parameters of the experiments carried out with the CTF team against other bots. The exact match settings are introduced along with the reasoning behind their selection. In addition, we will introduce the maps used in the matches and our tool for working with the maps.

In chapter 6, we will present the results of the experiments and depict the course of the matches.

In chapter 7, we will discuss our experiences during the course of behaviour development, the issues we encountered and how they were dealt with. We will present the identified limitations of yaPOSH and offer possible solutions to these limitations. We will also share our thoughts on other related topics, such as the impact of tool selection on performance of the resulting bots.

Finally, in chapter 8, we will draw conclusions based on the whole course of the CTF team development, the experiment results and the discussion in the previous chapter 7.

2. yaPOSH and Behaviour Oriented Design

In this chapter, we will introduce the concept of Behaviour Oriented Design (BOD) and the yaPOSH reactive planner based on BOD. We will also discuss its properties in relation to the behaviour specified in the next chapter 3 and the differences between using yaPOSH and plain Java when implementing the desired behaviour.

2.1 Behaviour Oriented Design

BOD is a modular technique for behaviour development and action selection. It is based on the idea that no matter how complex a behaviour is, it can be decomposed into lesser parts. These parts or smaller behaviours are called *behaviour modules*. The modularity means that a single behaviour can consist of several smaller behaviours and each of these can be decomposed further (and so on).

Then, using a collection (or library) of these modules, BOD uses a *dynamic plan*, a data structure representing relations of the modules, to process agent's action selection.

In the end, the end parts of the plan (its leaves) correspond to the simplest behaviour modules that cannot be decomposed further—and the whole behaviour, as well as its every part, consists of these leaves and their connections.

To summarize, there is a modular collection of behaviours (one behaviour can consist of several other behaviours) and there is a plan combining these behaviours.
BOD also offers a methodology specifying how the decomposition should be made. According to it, as described in the Pogamut lectures [16], one should

- name and identify the top-level behaviour the bot should be able to perform,
- identify behaviours that this higher-level behaviour consists of,
- identify senses that will trigger each of these lower-level behaviours,
- and identify these behaviours' priorities and consequently their switching conditions.

This process should then be repeated until primitive behaviours — primitives — are reached. More details about BOD as well as its decomposition methodology can be found in Bryson's Ph.D. thesis [3].

2.2 yaPOSH plan

yaPOSH is a planner based on BOD and a modification of the original Parallel-rooted, Ordered Slip-stack Hierarchical (POSH) planner introduced by J. J. Bryson [3]. yaPOSH was created and integrated into Pogamut by Jan Havlíček in 2013 [13].

As mentioned in the previous section, the planner essentially provides a way to define relations between the behaviour modules. As a consequence, the yaPOSH plan contains two different classes of nodes: *primitives*, representing the lowest-level behaviour modules, and *aggregates*, representing their relations.

2.2.1 Primitives

Primitives are methods implemented in some programming language (in our case in Java) that each carry out a certain action in the game environment. There are two kinds of primitives in yaPOSH and these also correspond to how humans perceive and interact with the environment:

- *actions*, used to execute actions, e.g. *jump*
- *senses*, used to sense information from the environment, e.g. *can I see a player?*

In the process of action selection through plan execution, the senses are used to determine which behaviour module should be executed. Actions are the leaves of the plan that are actually executed if selected. Simply put, based on what the agent perceives, he chooses an actions that he performs.

In yaPOSH, both actions and senses can be parametrized, meaning that we can have a sense `canSeeFlag(team)` instead of two senses `canSeeOurFlag` and `canSeeEnemyFlag`.

2.2.2 Aggregates

Aggregates are the inner nodes of the yaPOSH plan. They combine primitives together and thus form the whole plan. As opposed to the primitives, they do not interact with the game environment in any way. There are three different kinds of aggregates:

- *Drive collection* is the root of the yaPOSH plan. During every action-selection cycle, this root selects one of its children that will be executed. The drive collection can be viewed as a set of if-then rules with priority (the higher a child node is, the higher priority it has). The first child node

that meets its condition will be selected. During one cycle, only node node can be selected, meaning that any actions with lower priorities (those that lie lower in the plan) will not be executed in this cycle.

- *Competence* is a general modular plan. It can be also viewed as a set of if-then rules. Competences consist of multiple behaviour modules (other competences or actions) that the agent will choose from based on his senses, selecting a maximum of one module in each cycle (just as for Drive collection).

  Competence is a generalization of Drive collection in the sense that there can be multiple competences (but exactly one Drive collection). Other than that, they are the same.

- *Action patterns* are simply sequences of actions that will be executed in the given order, one by one.

Drive collection cannot be parametrized whereas competences and action patterns can. These parameters are passed onto their children, e.g. primitives. Competences and action patterns *themselves* do not use parameters because after all, they do not have or execute any code and instead, they link and point to the lower-level nodes, passing their parameters on.

The whole plan can then be viewed as a set of hierarchical if-then rules with priorities.

2.2.3 Plan example

To give an idea what the plan looks like and how one might benefit from using it, we present a plan of a dragon guarding his treasure. The plan is taken from Jan Havlíček's thesis (page 11 in [13]).

The dragon's task is to:

- Attack nearby thieves that try to steal his treasure, if there are any.
- Search of the treasure if its portion has been stolen.
- Blink if the above conditions are not fulfilled.

The plan's textual representation uses notation described on page 52 and is also included and taken from Havlíček's thesis (page 11 in [13]).

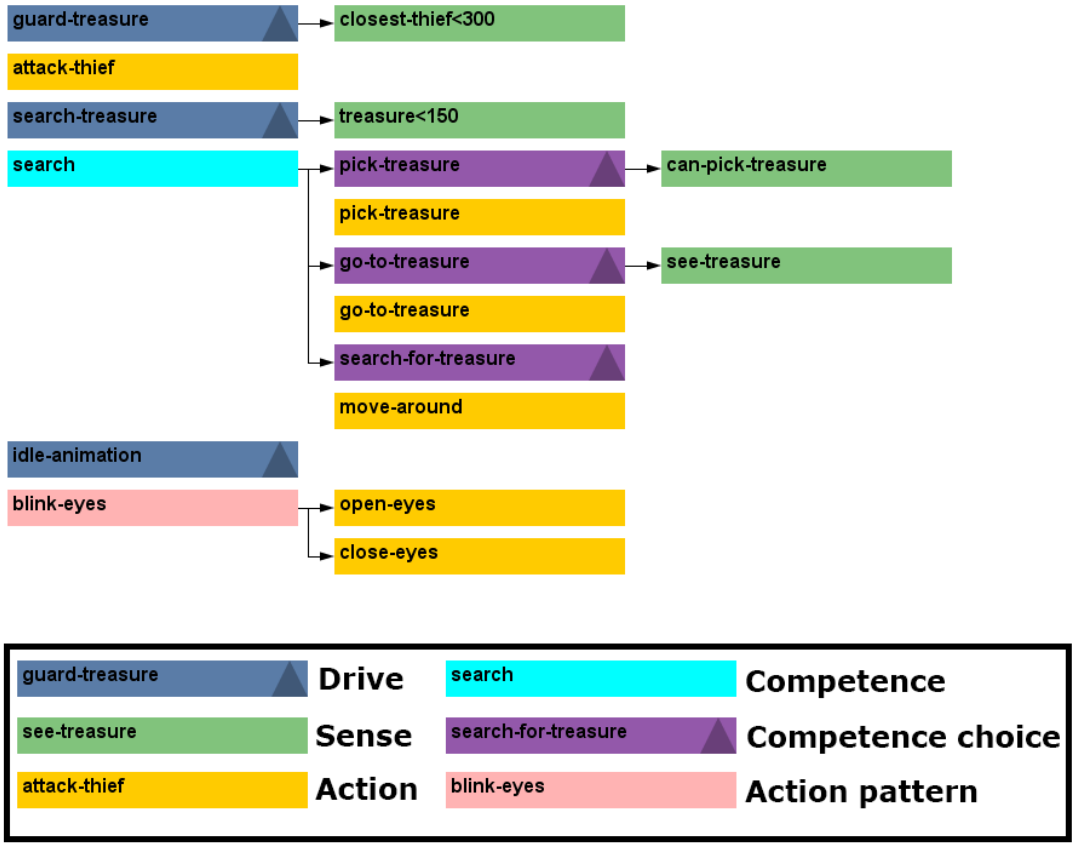


Figure 2.1: A yaPOSH plan of a dragon guarding his treasure.
Source: *Jan Havlíček, Tools for virtual agent behavior specification in POSH [13], page 11.*

```
(
    (C guard-treasure
        (elements
            ((attack-enemy (trigger ((see-hunter))) attack-hunter))
            ((search-hunter doNothing))
        )
    )
    (C search
        (elements
            ((pick-treasure (trigger ((can-pick-treasure true))) pick-treasure))
            ((go-to-treasure (trigger ((see-treasure))) go-to-treasure))
            ((search-for-treasure move-around))
        )
    )
    (AP blink-eyes (open-eyes close-eye))

    (DC life
        (drives
              ((guard-treasure (trigger ((closest-thief 300 <))) attack-thief))
              ((search-treasure (trigger ((treasure 150 <))) search))
              ((idle-animation blink-eyes))
        )
    )
)
```

Figure 2.2: A textual representation of fig. 2.1.
Source: *Jan Havlíček, Tools for virtual agent behavior specification in POSH [13], page 11.*

2.3 Comparison to Java

Using Pogamut, UT bots can be (and in most cases are) created using plain Java. These agents usually use finite state machines (FSM) or simply a set or hierarchy of if-then rules for their behaviour specification. This approach is not quite as robust as BOD and there are some noticeable differences between the two.

2.3.1 Code readability

The biggest difference, apparent at first sight, lies in code readability. Determining how the bot acts when only having Java source code at your disposal can take a lot of effort and time. On the other hand, the bot's behaviour is deductible from its yaPOSH plan very fast and easily. This is because the BOD approach uses an intuitive way of behaviour specification, based on perception (senses) and action execution.

However, this advantage is not granted as it requires the behaviour's developer to make use of one of the aspects of the BOD approach, that is a strong encouragement of proper naming conventions of both primitives and competences. It is highly recommended to name the behaviour modules by their actual meaning instead of with shortcuts or vague descriptions. The yaPOSH planner benefits from this even more, since its graphical editor displays the names of the behaviour modules and their relations immediately in the yaPOSH plan. Which is not something that one could typically get when using plain Java.

As a consequence, provided that the behaviour's author follows the BOD methodology, the behaviour is much easier to get a grasp of when specified with yaPOSH. Among other things, this plays significant role when a behaviour is being developed by multiple people, in which case the robustness and modularity makes efficient behaviour specification, maintenance and extension possible. As shown in several experiments presented in the study of AI tool evaluation [9], development tools matter and can greatly help with AI development.

2.3.2 Parallelism and Expressive power

Can everything expressible in plain Java be also expressed in yaPOSH? Technically it can, since the yaPOSH primitives are in fact Java functions and in the very least, one could simply create a yaPOSH plan with one action, the Java program. However, that would be very much against the concept of BOD.

When actually having BOD in mind and obeying its methods, yaPOSH does have some limitations, as also mentioned by Havlíček [13]. These limitations stem from the fact that a single bot can only have a single plan assigned to

him and there is no aggregate that would evaluate its children parallelly (or simultaneously). Also, during every update of the plan (see section 4.1), only one final action or aggregate is called.

This is a severe limitation, especially in the context of BOD, where a behaviour module consists of other modules whose simultaneous execution is often required. For example, just like a real soldier would move through the battlefield, communicate with his squad and fight all at once, the bot has to perform all these actions simultaneously as well. In our case, simultaneously does not mean at the *very* same time, but rather in one update iteration (one cycle of plan evaluation). Still, that might be a problem, because a bot can not be commanded to shoot and move during one update iteration as long as the shoot and move commands are separate actions (which they are required to be by design).

Theoretically, the problem could be solved by alternating behaviour modules in cycles, i.e. by executing navigation in one update iteration, combat in the other, and creating a cycle of behaviour module execution in this way. However, that solution would be far from ideal since in action games like UT2004, one cannot afford to delay any of behaviour modules by several update cycles — the reactions need to be instant.

One of our goals is to find out whether this restriction will prevent us from specifying a complex team behaviour in yaPOSH. If it will not be possible, we are not going to break the principles of BOD and use primitives that carry out different behaviour aspects. Instead, we will implement some of the behaviour modules in Java and execute this parallelly (in reality sequentially, but in one update iteration) to the yaPOSH plan.

3. Behaviour specification

In this chapter, we will describe what the resulting CTF behaviour should look like (or in other words, what we want the bots to do), the three parts it could be divided into and most importantly, why the decomposition is not only useful but also needed.

The three pillars of the behaviour are movement decision making 3.3, combat 3.4 and communication 3.5. We will describe these in detail.

3.1 Desired behaviour

We want the resulting CTF team to be able to play the game *well* and *believably*. Playing *well* means being equally good as or better than the native UT2004 bots on the same difficulty level[1].

Playing *believably* means to play similarly to how human players would, i.e. collect weapons before fighting, prioritize, capture and defend the flags, assist teammates and have different roles (with different priorities) assigned to different players. Ideally, when observing the game, one should not be able to tell the difference between human players and bots.

In effect, believability and level of play do have a lot in common and in some cases, the effort to create more believable bots can yield better results. Generally speaking, the more human-like the team plays, the better results it is going to achieve. Obviously, there are some exceptions to this rule (after all, making mistakes is one of the characteristics of human behaviour), but in complex games like UT2004's CTF mode, humans do have the upper hand on AI and team-based competitive FPS matches are only played between human teams, since bots are not challenging enough.

For example, a CTF team of native UT2004 bots of similar skill level as a group of human players poses hardly any challenge to the human team *if* the human players cooperate. Albeit on higher difficulty levels the bots might have an advantage in terms of aim precision and reaction times even over the best human individuals, their teamwork and cooperation levels are nowhere close to the cooperation effectiveness of human players.

To give an idea of what behaviour exactly we aim for, here is a list that summarizes the key behaviour aspects that the resulting bots should display and that are commonly observed when watching human players play CTF. Note that

[1]For Pogamut bots, the difficulty setting only changes their aim accuracy. On the other hand, difficulty affects many more attributes of the native bots. See section 5.2.1 for details.

it does not include all of behaviour aspects and those mentioned below could consist of several lesser parts.

- Before engaging in a fight, players collect weapons and other useful items upon spawning in the game environment.
- Players prioritize flags. For instance, when seeing a dropped flag, they try to get it instead of collecting items or chasing enemies.
- Players communicate and inform each other about flag status, enemy positions, important items, etc.
- Players on the same team have different roles and different priorities. For example, one player defends the base while another tries to steal the enemy flag.
- The roles may (and usually do) change dynamically depending on the in-game situation. For instance, an attacker will help with defence when in a more suitable position than a defender and vice versa.

3.2 Behaviour decomposition

The desired CTF team behaviour is immensely complex. In order to be able to design and implement it successfully, we need to break it down into several smaller and more easily graspable parts. We need these parts to be *independent* of each other, so that we can approach them individually and one by one. In our case, they are not *completely* independent as their certain elements are related. For example, dodging the incoming projectiles while fighting (combat module) would affect the navigation module a bit. However, we can deal with these cases separately as there are not too many of them.

The three identified parts can then be further decomposed using essentially the same approach. For instance, the combat system can be divided into picking a target, shooting and dodging. And again, each of these behaviour modules can be solved or approached independently and be decomposed further. The decomposition can continue until we reach a truly simple behaviour that corresponds to a single command in the context of Pogamut. Examples of these actions are *shoot*, *jump*, *move* or *turn*. In fact, these four basic commands are almost everything the bots need and a solid complex behaviour can be created using only a certain connection of these few actions.

3.3 Movement decision making

Determining where each bot should go at any given moment is both the most difficult and the most important problem of the CTF game mode. The reason being that first, a bot does not have complete knowledge about the current state of the game or map, second, there are dozens of locations he would benefit from going to and third, the optimal target location is also dependant on where the bot's teammates are currently going.

The task of this module is to select a target location that the bot should go to at any given time. Movement decision making will be the most complex of the three modules and will be handled by a yaPOSH plan.

Our movement decision making will be functioning on top of Pogamut's navigation system described below.

3.3.1 Navigation in Pogamut

The UT2004 navigation, as handled by Pogamut, is much more simple than one might imagine. This is because the bots do not actually *see* their surroundings. Instead, they use a navigation graph that is predefined for each map.

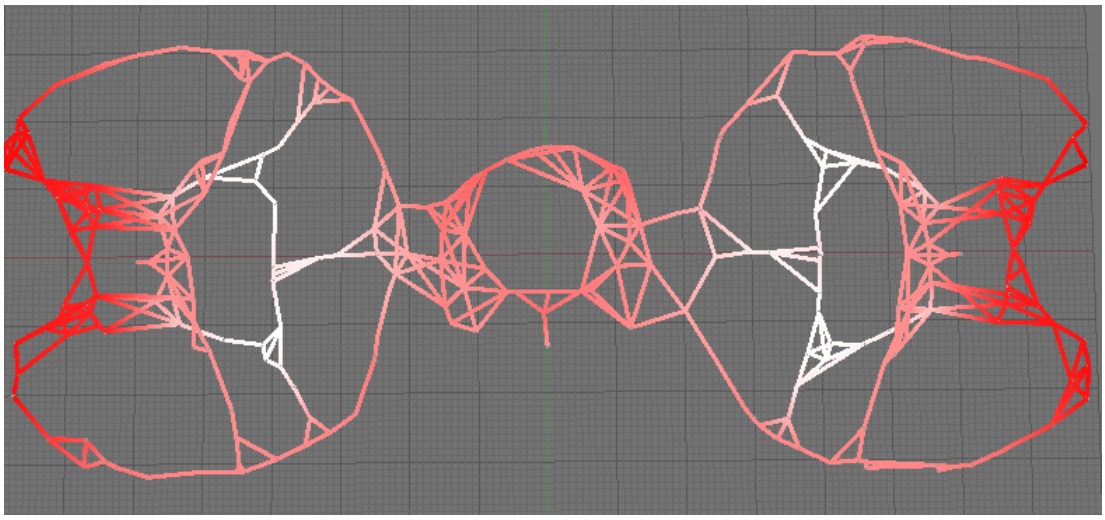


Figure 3.1: A navigation graph of the CTF-Lostfaith map as shown in the Pogamut NetBeans plug-in.

Bots move along the edges of the graph from one navigation point (a vertex of the graph) to another navigation point. Consequently, the number of possible destinations is severely limited, making the choices much more straightforward. Nevertheless, it is possible to have a bot move to a location outside the graph, but then the navigation is not quite that reliable.

Generally, when a bot finds himself out of the graph, he tries to get back to the nearest vertex and since he does not actually see, this often results in him

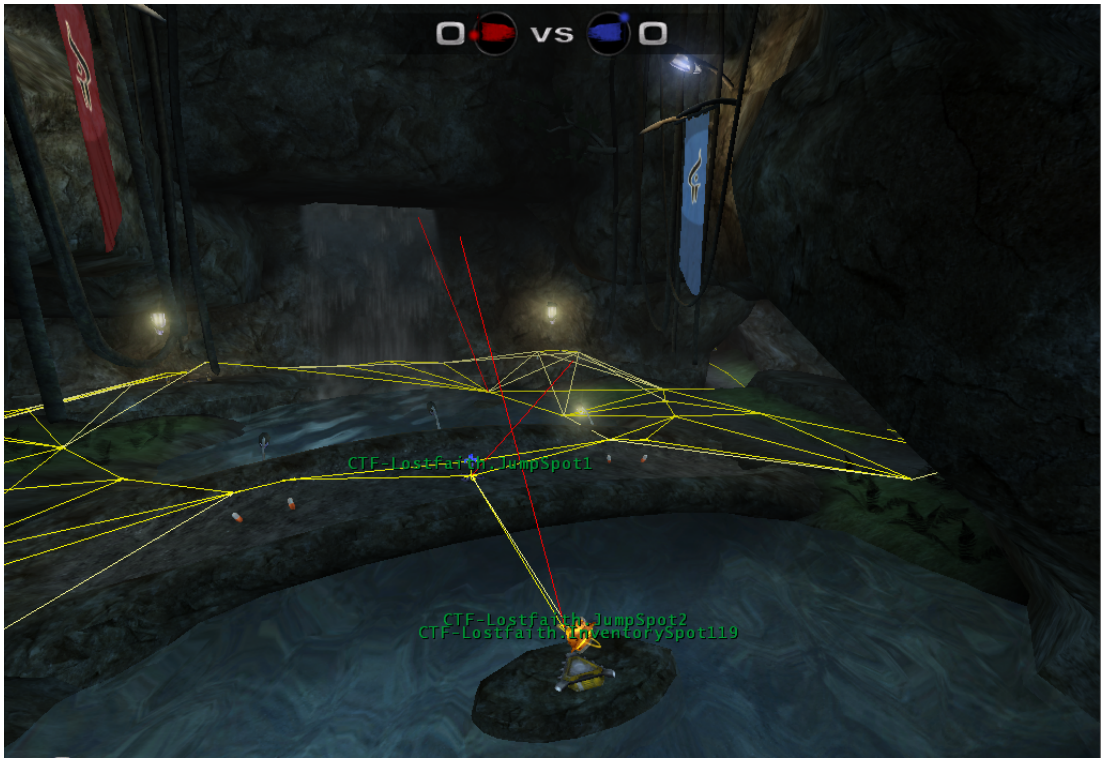


Figure 3.2: A navigation graph of the CTF-Lostfaith map shown in-game, formed by the yellow lines.

getting stuck in the environment. This is why certain actions, such as jumping or dodging, might cause problems and we try to avoid these using these actions when not necessary (often times, it might be better not to avoid incoming projectiles since it could result in problems with navigation).

As already mentioned, Pogamut offers high-level functions including those for pathfinding and path computation. To make a bot go to any location, one only has to call one function with one parameter, the target location, though it is still possible to specify the whole path manually. Pogamut also takes care of stuck detection and resolution, modifying the navigation graph in the process.

3.4 Combat

The combat module will take care of fighting and engaging the enemies. It is important to realize that since we decomposed our chosen behaviour into independent modules, combat will not be able to decide where the bot will go. This will be handled by movement decision making instead, which will also be taking care of weapon collecting. In a way, we might say that a part of combat behaviour handling will be delegated to the movement module.

Consequently, the combat module will be only in charge of shooting and weapon selection. Although at first it might not seem like it, weapon selec-

tion is actually much more complex and difficult to assess than the shooting process itself, which is because the weapon system of UT2004 is quite complex. UT2004 features about ten different weapon types, each with different parameters, strengths and weaknesses. Luckily, Pogamut does have built-in tools that help us select weapons automatically by specifying different weapon priorities for different ranges (see combat's implementation section 4.3.2 for details).

Even though we will focus on the CTF team behaviour and creating a perfect combat behaviour will not be our main priority (which would be the case in the Deathmatch mode), combat is still very important in CTF and the combat strength greatly affects the individual's and consequently the team's performance and results.

3.5 Communication

Communication between players is not necessary, not even in a team-based game mode like CTF. Actually, using communication might result in worse performance in some cases [18] than when not communicating.

On the other hand, its *effective* usage may improve a team's performance significantly. Our bots will communicate a number of different things, mostly those which human players would inform each other about as well. The communicated information include the following:

- flag locations;
- positions of visible enemies;
- position of the bot itself;
- information on items' availability;
- role swapping (the concept of roles is introduced in the next section 3.6).

As a result, any bot should at any time know positions of all of his teammates, know what items have been recently picked up and know the locations of enemies and both flags if recently visible. All this information should help the bot make better decisions if used properly.

In certain cases, however, not all the information has to be necessarily useful. For example, when a dropped flag finds itself close to the bot, the bot does not need to take positions of recently picked-up items into account to conclude that he needs to rush to the flag.

In fact, there is a big difference between the ways humans and our bots communicate. Typically, when a human player needs to know a specific piece of

information, he simply asks his teammates whether they happen to possess it. Which is quite unlike how our bots communicate — they simply send messages any time a specific event occurs, regardless of whether the recipients will need or use the information any time soon. The communication only occurs in one direction; there is no question and an answer, the messages are just one-way notifications.

3.6 Team strategy and roles

The three behaviour modules described above correspond to actions of individuals. But what about co-operation and differentiation inside the team? Human players surely do differ from each other and they do have dissimilar playstyles, which are all needed for a complex, universal and complete team performance. How to make our individual bots play not in the completely same way as the others and form a good team whose members would complement each other?

The answer to this question is by dividing the team members into certain groups with *different requirements* and *different priorities*. We will call these assignments *roles*. As commonly seen in game strategy guides [7] or observed in professional e-sports matches (as a matter of fact, real-life sports are no exception), players in every game specialize in specific roles that require specific skills and are different from other roles in that particular sport or game.

We will utilize this approach by assigning three different roles to our bots. The three roles are Attacker, Defender and Roamer. Each of the roles will have different objectives and priorities and as a result, our team's performance as well as its believability should be on significantly higher level than when not using roles. Roles are further described in section 4.2.

4. CTF team implementation

In this chapter, we will outline our chosen CTF strategy and its implementation. We will describe how Pogamut bots are updated, introduce the different roles and their priorities and how each of the main behaviour modules were specified. For complete information about the implementation, see the source code and most importantly the resulting yaPOSH plan.

The final strategy has been created with the help of the Garvelous' CTF guide [7], match replays from the UT2004 international ClanBase Nations Cup 2011, V. Tuma's bachelor thesis [18] and of course, personal experience. The strategy has been tested, modified and improved iteratively while focusing on the team CTF strategy rather than combat behaviour of individual bots.

4.1 Bot update cycle

Pogamut bots are updated periodically (rather than continuously), approximately every 250 milliseconds, i.e. four times a second. During one update cycle, the method `logic()` is called, executing the bot's behaviour. When using yaPOSH, which is our case, `logic()` is replaced by plan evaluation and in addition, methods `logicBeforePlan()` and `logicAfterPlan()` are called during the update cycle just before and just after the plan evaluation.

This allows us to use yaPOSH *in addition* to plain Java instead of as its replacement. By parallel evaluation, we mean sequential evaluation in one update cycle, i.e. command execution in `logicBeforePlan()`, in the yaPOSH plan and in `logicAfterPlan()` during one cycle. The three methods are called one immediately after the other and the time discrepancy between these is so small that it is not noticeable in the game environment. Thus, "parallel" evaluation is in fact possible but only with the help of Java methods surrounding the plan evaluation.

Pogamut also features custom *event listeners* that provide information about the game state. Event listeners are methods that are called when and only when a specific event occurs, meaning they are called asynchronously to the update cycle.

Every event listener is bound to a certain event class representing a change of either bot's inner state or the game environment. In our case, the events are specific to UT2004 and CTF and we use such event listeners as `botKilled()`, `targetReached()` or `itemPickedUp()`.

4.2 Roles

Our bots will use different roles, just as human players would when playing CTF. Roles define priorities and main goals of an individual and the whole CTF team strategy is essentially determined by the individual roles and their distribution.

Roles also allow us to decompose the desired behaviour even further, making its development considerably easier and even more user-friendly and comprehensible.

There are three basic roles, Attacker, Defender and Roamer, that differ mainly in terms of flag priorities and consequently, map movement and navigation.

The roles are not rigidly tied to specific agents and their assignments can change during the course of a match for several reasons (see section 4.2.4).

Next we will briefly describe priorities and characteristics of all the roles. For complete information about their logic implementation, refer to the easily comprehensible resulting yaPOSH plan whose textual representation can be found on page 52. However, it is best viewed in the Shed editor (page 59).

4.2.1 Attacker

Attacker's main priority is to get the enemy flag and eventually capture it.

Before heading to the enemy base, Attacker arms up to be able to fight his way all the way to the enemy base. When carrying the enemy flag, Attacker chooses the shortest path to his base and performs defensive or speed combos when he has enough adrenaline (see section 4.3.2 for information about adrenaline and combos).

When there are more Attackers on a team and one of them is carrying the enemy flag, the other Attackers follow and protect him on his way to the base. When the team is close to scoring, the other Attackers stop following the flag carry and go for the enemy flag instead.

Attackers also try to get their own team's dropped flag when they are closest to it, helping with the defence. When any of Attacker's teammates are closer to the flag than he is, he tries to get the enemy flag instead.

4.2.2 Defender

Defender defends his flag and his flag-base and if the flag is taken or dropped, he chases the enemy flag carry (EFC) and tries to return the flag back to base.

Upon spawning, Defender picks items near his base and then finds a defending point to defend the flag. We have experimented with various algorithms for finding a good defending point, but in the end the best (and more importantly, most

universal) results were obtained by simply staying on top of the flag once having enough items. This is mainly because there are more paths to the flag in all maps and it is not possible to guard all of them with one bot. In addition, Defender cannot afford to stay behind and chase the EFC, since Pogamut bots cannot use Translocators[1], which makes the chasing very hard and nearly impossible.

When the flag is taken, Defender either chases the EFC when visible, or runs straight to the enemy base if not visible. He performs offensive combos that increase his damage output when chasing the EFC.

When the flag is dropped, Defender goes to its last known location or to the enemy base if the enemy flag is there and the location of our flag is unknown (or visible with the flag not being there). Quite often, Defender gets to the enemy base sooner than the EFC and runs away with the enemy flag, preventing the enemy team from scoring.

4.2.3 Roamer

Roamer is a bit different from both Attacker and Defender — his goal is to control the middle grounds and often the key points of the map, since when navigating to the other team's base, one must inevitably pass through this territory. To identify the key points as well as other regions of the map correctly, we use map division (see section 4.3.3).

Roamer guards the middle and takes control of the items and power-ups there, which are important items that significantly boost a player's defence or offence capabilities temporarily. Consequently, Roamer should be able to take down or at least significantly hurt all enemies trying to get to his base, making Defender's task much easier. Roamer can also weaken the enemy Defenders and consequently help his Attackers.

Although flags are not Roamer's main priorities, when there is a flag nearby, he helps his team by either chasing the enemy flag carry, protecting his flag carry or picking up dropped flags. Since Roamer should often find himself near the centre of the map, he is often in a good position to aid his teammates.

When Roamer successfully takes control of important power-ups and is high on health as well, he tries to get the enemy flag when in base.

As opposed to Attackers and Defenders, it is not necessary to have any Roamers on a team, but having one can improve the team's performance quite substantially, especially on maps with choke points near their centre. It is important to realize that this role is not suitable for every CTF map. The less different paths there are between the bases, the more effective is Roamer going

[1]Translocator is a weapon that allows the player to teleport around the map.

to be.

4.2.4 Role distribution and switching

The optimal role distribution, i.e. the ratio of the roles to each other, is of course dependant on the map and on the enemy. We wanted our bots to be universal instead of optimized for a specific map or a specific enemy.

Through a number of experiments, we concluded that the best role ratio is 3 Attackers : 1 Defender : 1 Roamer. We will use this distribution ratio by default, since it is the most robust and universal of those that were tested and most importantly, can be used in any situation and on any map and still yield good results.

Nevertheless, our CTF team does not use completely rigid role distribution and the bots may switch roles in certain situations. Our bots can switch roles with a teammate as well as change roles without having the original role replaced by a teammate, resulting in a change of the role ratio and consequently the global team strategy.

Role switching occurs when a Defender or a Roamer picks up the enemy flag. In fact, holding and capturing the enemy flag is not even defined for Defender or Roamer and consequently, they change their role to Attacker and have a nearby teammate switch to their former role. When both flags are in their bases, bots that have previously switched roles return to their original roles. This step would not be necessary (in effect, it does not change a whole lot, since all bots can fulfil all roles equally well) but this is what human players would do as well — when needed, they switch their roles momentarily, but return back to original roles when possible.

Role changing (without replacements of the original roles) can occur when the end of the match is near (*near* is defined by a constant time value) and the team is winning or losing by some defined constant score value. When winning, Roamers switch to Defenders but Attackers do not, since especially in CTF, where your enemy cannot score when you possess their flag, the best defence is a good offence. When losing, however, everyone except the Defender switches to Attacker since, well, the best offence is a good offence too.

4.3 Behaviour modules

Our goal is to implement the CTF team using mainly yaPOSH. However, as we concluded in the previous chapter, it is unfortunately not possible to use yaPOSH *only* because the yaPOSH planner does not support parallel evaluation of its plans

and we *do* need to execute the three behaviour modules simultaneously.

Thus, we will use yaPOSH for the crucial behaviour module, movement decision making. The other two behaviour modules could be handled by yaPOSH just as well and we chose navigation mainly because it is the most complex module of the three and also since it greatly benefits from the decomposition that yaPOSH brings.

We will handle the other behaviour types — combat and communication — separately by using plain Java. As we have already mentioned, the behaviour execution is not truly parallel, but rather semi-parallel, in the sense that during every update of the bot, each module is executed exactly once.

4.3.1 Communication

The communication module consists of sending and receiving messages. Bots communicate with the help of the `TeamComm` Pogamut module that enables us to define our own message classes with serialized Java objects and send them using a TeamComm server. The server also uses channels, meaning that we can specify the recipient or recipients of the message.

Our bots send these messages:

- During every update cycle, they send their own location to all teammates.
- If carrying the flag, they send its location to all teammates.
- When seeing a dropped flag or a flag held by enemy, they send its location to all teammates.
- Upon seeing an enemy, they send his location to all teammates.
- Upon picking up an item that disappears temporarily, they send the item information to all teammates.
- When switching roles and in need of a substitute player for the original role, they send the role switch request to a specific teammate.

Communication significantly affects and improves navigation because it provides teammates with additional information that is used for movement target selection. For example, our bots do not navigate to items that have been recently picked up by a teammate because they know the item will not be there yet. More importantly, bots send messages about flags and are able to recover them or help with their capture very quickly.

Sending messages

Messages are sent in the `logicBeforePlan()` method before the combat module is executed and obviously before the plan is evaluated. This is to provide the information to teammates as soon as possible, preferably before the plan is evaluated.

Receiving messages

Messages are received asynchronously with the help of message event listeners and every bot is executed in a different thread. As a consequence, there is no guarantee that a message sent in `logicBeforePlan()` will be received and processed before the bot's plan evaluation.

4.3.2 Combat

Since the combat module does not interfere with navigation target selection, it is rather simple. Its main tasks are to identify visible enemy targets, pick a target to shoot and choose a weapon to shoot with.

Target selection

Target selection is based on flag status and bots will prioritize the EFC if there is one. When there is not a visible EFC, bots will try to hit the closest enemy, since the closer an enemy is, the bigger the chance of hitting him gets. Most weapons are also more effective in closer range.

Weapon selection

Choosing a weapon is a very important task as it has a great impact on bot's performance and consequently, on the performance of the whole team. In Pogamut, weapon selection is done quite easily thanks to its `weaponPreferences` module. Using this module, one can specify different weapon preferences (priorities) for different ranges (distances to target). For example, in close range, weapons with lower precision requirements and higher area of effect damage are preferred over weapons similar to the sniper rifle. We have implemented weapon preferences for 5 different ranges (for details, see method `setWeaponPreferences()`. When actually shooting, one only has to call the `shoot()` method with the weapon preferences as an argument and Pogamut takes care of automatic weapon selection.

Not only do different weapons have different uses and optimal distances to the target, but the optimal weapon selection is also largely dependant on the game difficulty setting. The higher the difficulty setting is, the better the slower,

single-target and high-damage weapons tend to perform. On the other hand, weapons with larger area of effect and weapon spread are usually better on lower difficulties, where the bots' aim is not that precise. More information about how the game difficulty affects the gameplay is in section 5.2.1.

Combat also includes dodging to nearby objectives — pickable items or flags, to speed up their collection. This is the only way in which the combat module affects movement. However, there is a drawback to this. Occasionally, the bot might end up outside the navigation graph after dodging and that might slow his navigation process temporarily.

Adrenaline and Combos

Adrenaline is an attribute (just like health or armour) that can be increased by collecting adrenaline pills in the game environment. Adrenaline can then be used for temporary power-ups — *combos* — getting consumed in the process.

Adrenaline is also gained on enemy kills and flag actions (recovery or capture). Nevertheless, its collection is comparatively slow since each adrenaline pill only yields 2 adrenaline points.

Adrenaline combos cost 100 adrenaline each. There are a total of six different combos, of which the by far most useful are Speed (increases run speed temporarily, useful for running away with the flag) and Booster (heals the bot over a period of time). Our bots will use these two combos in appropriate situations as well as the Berserk combo that increases the rate of fire temporarily (useful for chasing down the enemy flag carry).

4.3.3 Movement decision making

Movement decision making is handled by a yaPOSH plan. The plan's drive collection has three choices, one for each role. In practice, there should be a separate plan for each role, but since yaPOSH does not support multiple plans, we have to have one plan with multiple choices, one for each role. The choices themselves are competences with choices consisting of different flag states (in case of Attacker and Defender) and miscellaneous senses (in case of Roamer, whose main priorities are not flags).

To get an idea what the plan looks like and what it consists of, it is best viewed in NetBeans with the yaPOSH plug-in (see page 59). The plan's structure is described in 2.2 and its textual representation can be found on page 52.

Pogamut's navigation system in UT2004 is not ideal and does have some significant flaws. As a part of movement logic specification, we tried to improve the Pogamut navigation system by manual navigation graph changes for the tested

maps and by improving the stuck handling. We also added a tool called Map division that divides the map into different regions.

Map division

Map division is an algorithm that divides the navigation graph of any UT2004 CTF map into three different regions. These regions can then be used by the bots to improve their gameplay and their movement movement decision making.

Every navigation point is assigned to at least one map region based on its distance to both bases. The resulting map regions are *ourBase*, *middle*, *enemyBase* and *all*, spanning the whole map.

The base regions contain those points that are close to the respective base. *Close* means that the distance from the base to the point is at most 40% of the distance between the two bases. The base region is used by Defender in order to be close to his flag at all times.

The middle region contains points that are not too far away from both bases, i.e. the distance between the points and the two bases is 60% of the distance between the two bases or less. The middle part is used by Roamer to take control of the important points that are on the way between both bases.

The resulting map division graphs for the maps we used in the experimental matches can be seen in the Experiments chapter 5.3.

Altering Pogamut's navigation system

Since Pogamut bots do not actually see the environment and all they use is the navigation graph, they do get stuck relatively often, especially after being pushed away from the graph itself. Being stuck means being unable to move and it is most frequently caused by trying to get navigation back to it with an obstacle in the bot's way (which he obviously cannot see and instead tries to move through it, resulting in him getting stuck in place).

Pogamut does have some methods for stuck detection which work very well (in terms of the actual detection), but do not *handle* stuck bots properly. That is why we implemented some basic stuck handling with the help of the built-in stuck detectors.

Proper stuck handling is important especially for longer matches, where bots cannot afford to be stuck in one place forever, which is what can happen quite often if not properly handled. We added a simple stuck counter that respawns the bot when stuck for too long and we also use the following method of stuck protection.

Pogamut features the `UT2004PathAutoFixer` class that automatically alters

the navigation graph by removing edges that cause navigation problems reported by Pogamut's stuck detectors. We used and modified the `UT2004PathAutoFixer` specifically for CTF, where it now detects missing paths to flag bases and respawn navigation points and can reset the navigation graph appropriately.

We also manually tweaked the navigation graphs of some of the tested maps by removing the edges that often caused the bots to stuck and by adding some safe edges to compensate for their overall decrease. In addition, we implemented a navigation enhancer that uses Pogamut's `TabooSet` class to temporarily prevent the bot from navigating to locations that have recently caused him to get stuck.

As a result, our bots should never get stuck indefinitely and their navigation works without any problems even after 12 hours of continuous gameplay on a single map.

5. Experiments

In this chapter, we will introduce the opponents of our bots for the experimental matches, specify the exact match settings and introduce the maps the matches will be played on.

5.1 Opponents

We would like to see how our resulting bots perform in comparison to the native UT2004 bots and to the CTF team created by V. Tuma in his bachelor thesis [18].

The matches will be played by two teams, each of size 5.

5.1.1 Native bots

Native bots are the in-game bots created by the developers of UT2004. As we mentioned earlier, our goal is to make a team that is better than native bot CTF teams, which would require actually making our bots smarter than native bots, since on higher difficulty levels, native bots do have some advantages over our bots (see section 5.2.1).

5.1.2 V. Tuma's bots

Bots by Vojtěch Tuma [18] are a part of his bachelor thesis, in which Tuma studied the influence of communication on CTF bot teams and consequently implemented several different teams with different properties and strategies. In our experiments, we will use Tuma's most successful CTF team, i.e. the team without communication whose individuals are based on GladiatorBot from David Holaň.

GladiatorBot is a very successful Deathmatch bot and a two-time winner of Pogamut Cup[1].

Tuma's experiments showed that his best team was able to beat native bots convincingly, not losing a single match. Consequently, we would like to find out whether our bots will perform better against the native bots than against Tuma's bots.

[1] Pogamut Cup (`http://www.pogamutcup.com/`) is an annual competition of UT2004 Deathmatch bots organized by the Artifical Minds for Intelligent Systems group (`http://amis.mff.cuni.cz/`).

5.2 Match settings

Our general approach is to use the standard CTF settings when possible. By standard we mean the settings used in past UT2004 CTF competitive matches. Complete match settings can be seen in section 5.2.8.

5.2.1 Difficulty level

The difficulty (or skill level) ranges from 0 to 7 and for Pogamut bots, this setting only affects their aim accuracy. On the other hand, the native bots' parameters vary significantly more on different difficulty levels because even attributes like field of view or movement speed depend on the difficulty level.

Native bots below level 5 are handicapped in some way (such as by having lower movement speed) and when above level 5, they do have unfair advantages (they can see through obstacles or have a 360° field of view, etc.). For detailed overview, see the Unreal Engine documentation [19]. The lower the skill level is, the easier it gets for our bots to compete against native bots.

Even for Pogamut bots, however, the difficulty level can affect their performance significantly, since weapon selection and weapon effectiveness is greatly influenced by aim accuracy. For these reasons, we will carry out our experiments on different difficulty levels, ranging from 5 to 7 to identify and possibly eliminate the impact of difficulty level on match results. Our bots were not optimized for any specific difficulty level and their weapon selection should be balanced across the skill range.

5.2.2 Weapons

All weapons are enabled and the *Weapons stay* option is enabled, meaning that weapons do not disappear temporarily after being picked up. This setting is standard for the CTF mode to allow for fast game-pace based on objectives (flags) instead of on weapon collecting.

5.2.3 Match limits

We will use a standard Time limit of 20 minutes per match and we will not be using a Score limit.

5.2.4 Team size

Both teams will have 5 players, which is a standard CTF team size. Luckily, Pogamut is able to handle simultaneous execution of 10 different agents without

any problems.

5.2.5 Translocator

Since Pogamut bots can not use Translocator, Translocator is disabled for all matches. Translocator's unavailability will change the whole CTF game considerably compared to competitive matches, since it is one of the main elements of CTF gameplay when played by human players.

5.2.6 Mutators

Mutators are game effects that change environment settings or add other effects, such as gravity, game speed or healing over time to the game. We will not use any Mutators.

5.2.7 Friendly fire

The friendly fire multiplicative ratio (from 0.0 to 1.0) determines how much can a player hurt his teammates. The 1.0 value would mean that one can damage his teammates in the same way as his opponents, 0.5 would mean half the total damage, and so on. We will use a value of zero, meaning that the bots cannot damage their teammates at all. However, friendly fire does not affect the damage the player can inflict to himself (one can always damage himself, for example by shooting with area of effect weapons in extremely close range).

5.2.8 Settings overview

This is a summary of the exact game and server settings we will use. In all matches, both bot teams will be using the same skill level setting (5, 6 or 7).

Table 5.1: Match settings overview

Team Size	5
Time Limit	20 (minutes)
Goal Score	none
Bot Skill	5 to 7
Weapons stay	ON
Friendly Fire Scale	0.0
Spawn Protection Time	2.0 (seconds)
Allow Weapon Throwing	ON

5.3 Maps

In order to avoid map-specific bias, we will test the bots on three different maps. We will use the maps that were used by V. Tuma to make sure his bots run without any problems there. The three maps are CTF-Citadel, CTF-Geothermal and CTF-Lostfaith and their description follows.

We also included a graph created using the Map division tool for each map. Red and blue nodes belong to the respective team bases and green nodes are the middle parts of given maps. Note that the graph is a 2D projection of a 3D environment since we neglected the z-coordinate (the height dimension).

5.3.1 CTF-Citadel

Citadel is the smallest and most open of the three maps. The bases are towers facing each other with three narrow bridges between them. This map is also a bit asymmetrical, since the shortest paths between the two bases are different for each side and lie on the opposite sides of the map.

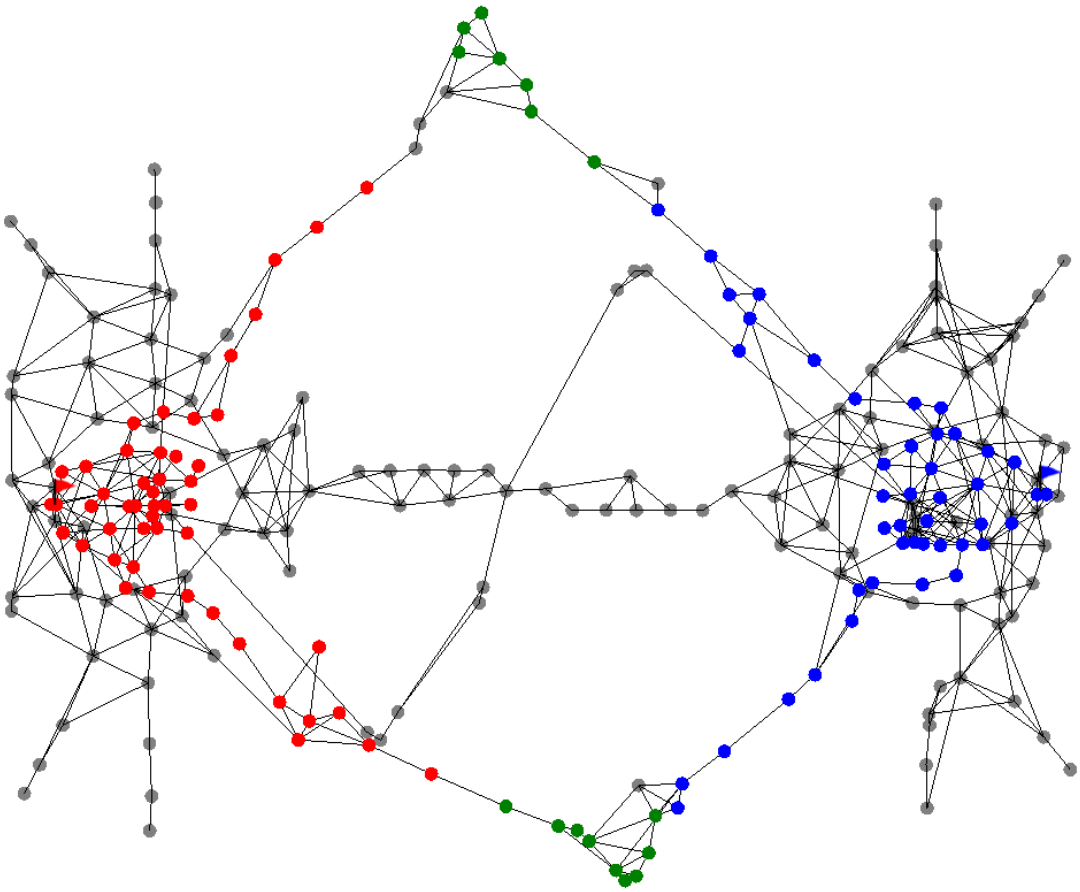


Figure 5.1: CTF-Citadel's navigation graph created with the Map division tool. Red: red base, blue: blue base, green: middle grounds.

Citadel is the only selected map with noticeable differences in the height

dimension, meaning that the map division graph might be slightly misleading. For example, the middle bridge is well above the level of flag bases and one cannot get the enemy flag and run straight back home using this bridge (this is why its centre does not belong to the middle parts according to Map division).

5.3.2 CTF-Geothermal

Geothermal is a closed map with a number of smaller rooms and tunnels. It is nearly symmetrical and probably the most balanced map of the three. It contains all the power-ups and weapons.

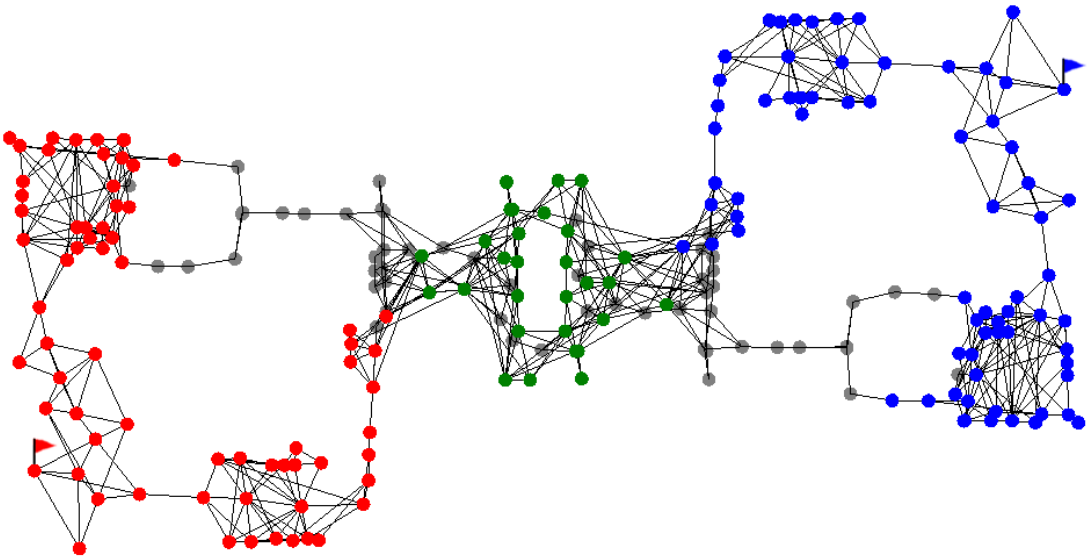


Figure 5.2: CTF-Geothermal's navigation graph created with the Map division tool.

5.3.3 CTF-Lostfaith

Lostfaith is by far the largest of the three maps and it does have a characteristic open-spaced middle region that cannot be circumvented when going from one base to the other.

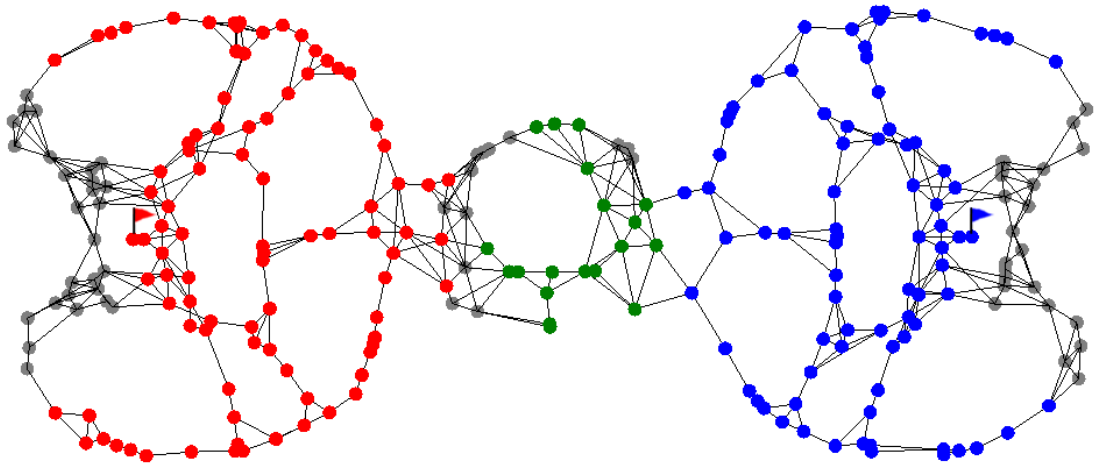


Figure 5.3: CTF-Lostfaith's navigation graph created with the Map division tool.

The pace of the game is often much slower here and flags can be recovered easily thanks to the key middle grounds.

Lostfaith is also the map whose navigation graph had to be altered the most in order to prevent the bots from getting stuck. We also added some safe edges to the navigation graph that are not present by default there.

5.4 Format

Our bots will play three matches with the specified settings against each opponent on each difficulty, each map and each side. Since there are two opponents, three selected difficulty levels three maps and two sides, our bots will play a total of 108 20-minute matches which is equal to 36 hours of gameplay.

If the results will be very close in any specific match configuration, we will run further tests to determine their outcome with greater precision. We expect our bots to be able to beat the native bots and to hopefully stand up to Tuma's bots as well. We also think that our bots will perform much better on lower difficulties than the native bots, who do have some advantages on difficulties 6 and 7.

6. Results

This chapter presents the results of our experiments. We also offer an insight into the course of the matches.

The results are written from our bots' perspective, meaning that the score is always represented as *our bots' score*:*enemy bots' score* and Team represents the current side of our bots. Score is the total number of successful flag captures throughout the match.

6.1 Native bots

6.1.1 CTF-Citadel

As one can see in the table in table 6.1, two surprising phenomena occurred in matches against native bots on Citadel: firstly, our bots' performance compared to the performance of native bots was very similar throughout the three difficulties and secondly, the Red side apparently has a significant advantage on Citadel.

Table 6.1: Results of matches against native bots on CTF-Citadel

	Skill level		
Team	5	6	7
Red	17 : 0	14 : 2	18 : 0
	15 : 0	13 : 0	20 : 0
	20 : 0	13 : 1	12 : 1
Blue	12 : 4	7 : 2	7 : 1
	11 : 4	6 : 2	7 : 1
	11 : 4	8 : 3	10 : 0

We expected our bots to do better on difficulty 5 than on difficulties 6 and 7, which was not really the case. That can be attributed to weapon selection, since in long range (which is most often the case on this Citadel), our bots use high-range and high-damage weapons similar to the sniper rifle, that are most effective on high difficulties (6 and 7).

The Red side's advantage probably stems from the fact that the map is asymmetrical and the path from blue base to red base is much more covered than the other one.

6.1.2 CTF-Geothermal

Table 6.2: Results of matches against native bots on CTF-Geothermal

	Skill level		
Team	5	6	7
Red	12 : 2	10 : 3	8 : 4
	10 : 2	10 : 2	6 : 1
	10 : 3	9 : 2	6 : 1
Blue	13 : 2	7 : 1	7 : 0
	12 : 1	8 : 1	3 : 1
	10 : 2	8 : 1	6 : 0

Matches on Geothermal went as expected, meaning that our bots outperformed the native bots but the difference grew smaller as the difficulty got higher. Nonetheless, our bots were still able to beat native bots quite handily and reliably.

6.1.3 CTF-Lostfaith

Table 6.3: Results of matches against native bots on CTF-Lostfaith

	Skill level		
Team	5	6	7
Red	6 : 2	4 : 1	3 : 0
	6 : 1	3 : 0	3 : 0
	7 : 1	3 : 1	3 : 0
Blue	7 : 2	3 : 0	1 : 0
	6 : 1	3 : 0	2 : 0
	5 : 1	4 : 1	5 : 1

In terms of results, Lostfaith was similar to Geothermal, except the game-pace was much slower and there were fewer total flag captures.

Our Roamer bot also had a significant impact on the game by controlling the very important armour power-up in the middle grounds.

6.2 Tuma's bots

6.2.1 CTF-Citadel

Just like we observed in matches against the native bots, red side does appear to have the upper hand on Citadel. Matches against Tuma's bots also clearly show that the lower the skill level of both teams is, the more the results are in favour of our bots.

Table 6.4: Results of matches against Tuma's bots on CTF-Citadel

	Skill level		
Team	5	6	7
Red	18 : 0 15 : 1 13 : 2	11 : 0 12 : 0 11 : 1	6 : 1 4 : 0 5 : 0
Blue	10 : 1 12 : 0 5 : 0	8 : 0 7 : 0 8 : 0	4 : 0 3 : 0 3 : 0

That is most likely a consequence of Tuma's bots being optimized for skill level 7, rather than our bots performing worse there, since the difference between our and native bots was rather insignificant and can be mostly attributed to the advantages that Tuma's bots do not possess (such as the ability to see through obstacles). It is worth mentioning that in his thesis, Tuma only ran the experiments on difficulty level 7 [18].

The results on Citadel are quite regular except for the 5 : 0 match on difficulty 5, in which there was an about 10-minute long stalemate when both teams held the opposing team's flag in their respective bases. In fact, in the end neither team was able to retrieve the respective flag and the game ended during the stalemate.

6.2.2 CTF-Geothermal

Once again, the different skill level scaling was quite apparent in matches on Geothermal. Nevertheless, the game was only drawn once and our bots secured victory in all the other matches.

Table 6.5: Results of matches against Tuma's bots on CTF-Geothermal

	Skill level		
Team	5	6	7
Red	11 : 0 12 : 0 8 : 0	4 : 0 3 : 0 6 : 1	0 : 0 4 : 0 2 : 0
Blue	9 : 0 13 : 0 7 : 1	5 : 0 6 : 0 7 : 0	2 : 0 2 : 0 1 : 0

It seems that although our strategy only included one Defender, our defensive capabilities were more than sufficient and in some cases, it might be possibly worth focusing on offence even more.

6.2.3 CTF-Lostfaith

Since Lostfaith is the largest and consequently the slowest-paced chosen map, it should not come as a surprise that the results were closest here. At any rate, our bots managed to score in every game on levels 5 and 6 whereas Tuma's bots only achieved one capture.

Table 6.6: Results of matches against Tuma's bots on CTF-Lostfaith

	Skill level		
Team	5	6	7
Red	2 : 0 3 : 0 2 : 0	2 : 0 6 : 0 2 : 0	0 : 0 0 : 0 0 : 0
Blue	2 : 0 3 : 0 2 : 1	1 : 0 2 : 0 1 : 0	0 : 0 0 : 0 0 : 0

The most intense and close matches were played on the highest difficulty, where all matches ended in a 0 : 0 tie. This is mainly a consequence of the open middle grounds, where bots on the highest skill level kill each other almost always and almost instantly, resulting in very few of them getting to the enemy base.

Actually, there were a couple of flag steals, some of which even occurred simultaneously, but in the end, none of them resulted in a flag capture. Why is

that? Well, when having the enemy flag, one still has to get back to his base and once again cannot avoid the open-spaced middle grounds, in which one is an easy target.

6.2.4 Additional matches

Since the results on CTF-Lostfaith were very close, we decided to run additional tests to find out which bots perform better in longer matches (20 minutes is often not enough for a team to decisively prove it is better).

We ran 8-hour matches against Tuma's bots on both sides and all three difficulties. Here are the results:

Table 6.7: Results of the additional 8-hour matches on CTF-Lostfaith

	Skill level		
Team	5	6	7
Red	58 : 3	51 : 4	14 : 6
Blue	48 : 6	32 : 4	7 : 3

The results were quite clear except for the highest difficulty, which we would like to talk about further.

Given the length of the matches (which would be equal to 24 20-minute matches), they were still extremely close with very few captures. In the long run, nonetheless, our bots performed better.

The deciding factor might have well been the use of adrenaline (4.3.2). Whereas our bots do make use of it, especially when trying to capture the flag, Tuma's bots did not appear to use adrenaline at all. Since adrenaline is collected slowly and over time, its usage does not play that big of a role in the shorter 20-minute matches. In the longer 8-hour matches, however, it had a significant impact and probably decided their outcome.

6.3 Summary

Our bots performed very well against both native and Tuma's bots, did not lose a single match and won their vast majority. Overall, our bots performed better than expected, proving that good communication and teamwork can beat strong individual skills.

While Tuma's bots are more proficient in one-on-one combat and weapon selection, our bots managed to exploit their lack of communication and team-based objective focus and score more points despite not fighting so effectively. One of the main reasons that our bots beat Tuma's bots was simply a higher

prioritization of flag actions: in some cases, our bot got to the flag before one of Tuma's bots despite the latter being closer to the flag at first.

We also find it interesting that in few cases, our bots had better results against the bots by Tuma than against the native bots. That is a bit surprising, since Tuma's bots were able to beat the native bots in their every game [18]. Then again, that was on a different difficulty level, which shows how important it is to carry out multiple experiments under different conditions.

7. Discussion

In this chapter, we will describe the lessons learned during the whole course of the CTF team development and discuss the limitations, use cases and possible improvements of yaPOSH.

7.1 Decision making versus coding

Our goal was simply to create a CTF team using yaPOSH and BOD. At first, the task seemed fairly complex, but with the help of decomposition that BOD uses, designing the desired CTF behaviour was rather straightforward.

What helped us immensely was the separation of specification of decision making from the actual low-level Java coding. This is often mentioned as beneficial for the decision making specification, where one can forget about coding and fully focus on the AI itself. That is surely true, but we believe that it also works the other way around — and that it helped us greatly with programming the primitives library. Another advantage of this approach that the collection (or library) of primitives is a standalone entity that can be extended independently, is not necessarily restricted to a single project or a piece of software and can be reused in the future.

Once we had a library of actions and senses which we essentially could think of as little pieces of behaviour, we could fully focus on the actual logic implementation and the decision making, done by simply connecting the already finished pieces.

7.2 Importance of tool's specifics

Then, however, we encountered a serious problem — that yaPOSH does not support parallel evaluation of its plans or execution of multiple plans at once. By design, we wanted and needed to evaluate our behaviour modules (communication, combat and map movement) simultaneously. We solved (or more precisely, avoided) this problem by specifying the two smaller and less-demanding modules in plain Java and by using yaPOSH for map movement specification only.

We purposely first designed the behaviour and only then started to think how it could be implemented with yaPOSH, in order to discover its boundaries and limitations and in order to find out whether the tool is universal and usable in all cases (which did not prove to be the case).

However, when the implementation is the actual goal (instead of evaluating a tool's suitability), one should definitely first assess the tool itself, identify its properties and limitations and only then work on the problem, having the tool's characteristics in mind and building the solution utilizing the tool's qualities.

7.3 Different techniques for different tasks

Developing different modules using different techniques (or tools) helped us realize that *each of the techniques is more suitable for different tasks.* For example, the combat module was comparatively small and simple and we had no problems implementing it using plain Java. In fact, using yaPOSH for this task would probably take more effort and would not be as efficient.

On the other hand, the complex movement module could not have been written in plain Java as easily as it was done with yaPOSH. Our observation is that *the more complex a behaviour is, the more it benefits from BOD's robust and modular approach.* As a consequence, we believe that yaPOSH is a suitable tool for developing complex behaviour, but it is not necessarily that effective when the desired behaviour is comparatively simple. Additionally, there is a fine line between *simple* and *complex* that most likely varies individually and it cannot be ultimately determined whether a given task or behaviour is simple or complex.

In reality, selecting a suitable tool for a given task is a problem of its own and in order to choose the proper tool or technique, one must be aware of his own capabilities and preferences as well as the task's and potential tools' characteristics.

7.4 When to use yaPOSH

Although we cannot say when exactly yaPOSH is the best or the most suitable tool, we can still identify in which cases one would *benefit* from using it. These cases include:

- complex behaviour that is decomposable into smaller parts,
- behaviour that in the end consists of few simple actions,
- behaviour whose modules can or need to be parametrized,
- behaviour that should be transparent and easily comprehensible,
- behaviour that is or will be developed by multiple subjects,

- behaviour whose iterative development and improvement is required,
- behaviour whose logic specification and actual implementation is done by different subjects,
- behaviour for which a library of primitives already exists,
- behaviour with human-like characteristics.

In our case, we used yaPOSH for a complex human-like behaviour module that made great use of decomposition as introduced by BOD as well as parametrization and iterative development.

Without any doubts, creating the movement logic specification using yaPOSH was the right decision as it made its continuous development considerably easier and more comfortable than would be the case when using plain Java.

7.5 When not to use yaPOSH

As we have already mentioned, yaPOSH is not the ultimate solution to behaviour specification and in some cases, it's usage might not suitable or helpful. These include mainly:

- behaviour with only one layer of decisions,
- behaviour that cannot be decomposed into smaller parts,
- behaviour that requires parallel evaluation of its modules.

We specified our combat and communication modules in plain Java and it proved effective. On the other hand, combat was not our main focus and potentially, this module could be further improved and extended, in which case using yaPOSH *from the very beginning* would have been the more appropriate choice.

Then again, the lack of support of multiple plans or parallel evaluation is what prevented us from doing so.

7.6 Improvement suggestions

Some of the limitations of yaPOSH (such as that it is not suitable for specification of very simple or flat-structured behaviours) stem from the fact that yaPOSH is based on a robust and modular methodology, BOD. These limitations should be considered its properties rather than its flaws and there is no point trying to remove them.

However, there are certain disadvantages that come from its implementation and *are not enforced by design.*

7.6.1 Parallelism

Most importantly, yaPOSH does not support parallel evaluation or evaluation of multiple plans at once. At least one of these features is required for complete and complex behaviour specification to be possible.

It would be easier to implement the latter, that is the *ability to execute multiple plans at once.* However, that would require the behaviour developer to only execute independent actions in each of the plans, so that they would not interrupt each other.

This is by far the most significant issue of yaPOSH as the other we have identified would not open up new possibilities or improve the planner's expressive power.

7.6.2 Plan variables

Another improvement could be achieved by introducing *plan variables.* At the moment, the plan's primitives and aggregates can only be parametrized by constants directly present in the individual nodes.

An addition of plan-wide variables that would be valid for all the plan nodes would make the behaviour development considerably cleaner and more transparent. Consequently, one would be able to parametrize multiple aggregates or primitives using only one easily accessible and modifiable constant.

7.6.3 Cosmetic changes

The other suggested changes are purely cosmetic and are related to the user interface of the Shed editor:

- When debugging, display the bot's name in the plan (currently one cannot match the plans to the in-game agents when there are more of them).
- Allow selection of multiple nodes.
- Allow operations on the selected nodes (copy, cut, paste, delete).
- Allow repositioning of the nodes.

Although the importance of the suggestions above is nowhere near to the importance of multiple plan support, they would make the behaviour specification much more user-friendly and would perhaps increase the planner's popularity.

7.7 Final thoughts

Even though we were only able to use yaPOSH for a part (albeit the most important part) of the final behaviour, the tool helped us immensely to shape a solid CTF team with very good performance results.

We have observed that the more complex the desired behaviour is, the more can its development benefit from the BOD methodology as well as from its realization by the yaPOSH planner.

Ironically enough, the probably most significant disadvantage of yaPOSH is that in some cases, it cannot be used for everything one might like to — that is when the behaviour requires parallelism support.

Should this limitation be removed in the future and should perhaps some of the suggested quality of life improvements be introduced, yaPOSH would become a truly universal tool for complex behaviour specification.

8. Conclusion

We created a competitive CTF bot team for UT2004 whose development was driven by Behaviour Oriented Design (BOD), a robust and modular technique for behaviour specification. The CTF team was implemented using the Pogamut platform mainly with the yaPOSH planner, a tool based on BOD.

Selection of the BOD approach made the behaviour development very convenient and led to effortless behaviour debugging, modification and extension, which is what allowed us to create a competitive CTF team.

The resulting CTF team performed very well in game, beating all its opponents including the native UT2004 bots and the CTF bots by V. Tuma [18].

The yaPOSH planner we used for CTF team specification does have some limitations, such as the inability to have or execute multiple plans at once and the lack of parallelism support. In spite of that, we strongly believe that even in its current state, yaPOSH is a suitable tool for complex behaviour specification that can make AI development considerably easier and much more user-friendly.

We have identified and suggested a number of improvements whose future implementation would extend the usability of yaPOSH even further, namely the ability to have and execute multiple yaPOSH plans at once and the addition of plan variables.

The modularity of the BOD approach and the high level of code readability of the yaPOSH planner also make future improvements and extensions of our resulting CTF team possible.

Bibliography

[1] Brom, C., Gemrot, J., Bida, M., Burkert, O., Partington, S. J., Bryson, J. J.: *POSH Tools for Game Agent Development by Students and Non-Programmers*. In: Mehdi, Q., Mtenzi, F., Duggan, B. and McAtamney, H., eds. The Nineth International Computer Games Conference: AI, Mobile, Educational and Serious Games, 2006-11-01, Dublin.

[2] Bryson, Joanna Joy: *Behavior-Oriented Design of Modular Agent Intelligence*. In: Agent Technologies, Infrastructures, Tools, and Applications for e-Services, R. Kowalszyk, J. P. Müller, H. Tianfield and R. Unland, eds. Springer, 2003, pp. 61-76.

[3] Bryson, Joanna Joy: *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD thesis, MIT, Department of EECS. Cambridge, MA, 2001.

[4] Champandard, Alex J.: *Behavior Trees for Next-Gen Game AI* [online]. Internet presentation. 2008 [cit. 2014-07-28]. `http://aigamedev.com/insider/presentations/behavior-trees`

[5] Champandard, Alex J.: *AI Game Development*. Indianapolis, Ind: New Riders, 2003.

[6] DeLoura, Mark A.: *Game Programming Gems*. Rockland, MA: Charles River Media, 2000.

[7] Garvelous: *Unreal Tournament 2004 Capture the Flag Strategy Guide* [online]. Guide. 2009 [cit. 2014-07-05]. `http://www.oocities.org/garvelous2004/`

[8] Gemrot, J., Brom., C., Bryson, J. J., Bída, M.: *How to Compare Usability of Techniques for the Specification of Virtual Agents' Behavior? An Experimental Pilot Study with Human Subjects*. In: Proc. Agents for Educational Games and Simulations - International Workshop, AEGS 2011, LNCS 7471, Springer, 2012, pp. 38-62.

[9] Gemrot, J., Černý, M., Brom, C.: *Why you should empirically evaluate your AI tool: From SPOSH to yaPOSH*. In: Proceedings of 6th International Conference on Agents and Artificial Intelligence (ICAART 2014). 2014, pp. 461-468.

[10] Gemrot, J., Havlíček, J., Bída, M., Kadlec, R., Brom, C.: *yaPOSH Action Selection*. In: Intelligent Virtual Agents. LNCS 8108, Springer, 2013, pp. 458-459.

[11] Gemrot, J., Hlavka, Z., Brom, C.: *Does high-level behavior specification tool make production of virtual agent behaviors better?* In: Proc. of Cognitive Agents for Virtual Environments, LNCS 7764, Springer, Heidelberg, 2013, pp. 167-183.

[12] Gemrot, J., Kadlec, R., Bída, M., Burkert, O., Píbil, R., Havlíček, J., Zemčák, L., Šimlovič, J., Vansa, R., Štolba, M., Plch, T., Brom C.: *Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents*. In: Agents for Games and Simulations, LNCS 5920, Springer, 2009, pp. 1-15.

[13] Havlíček, Jan: *Tools for virtual agent behavior specification in POSH*. Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics. Prague, 2013.

[14] Lewinski, M., Demir, C., Anantharam, R.: *Incorporating Team Strategies in Bots for Capture the Flag mode of Unreal Tournament 2004*. Utrecht University, Faculty of Science, Department of Information and Computing Sciences. 2009.

[15] Partington, S. J., Bryson, J. J.: *The Behavior Oriented Design of an Unreal Tournament Character*. In: Proceedings of IVA'05, LNAI 3661, Springer, 2005.

[16] *Pogamut 3 Lectures 2013/2014* [online]. Practice lessons website of the Human-like Artificial Agents course. 2014 [cit. 2014-05-26].
`http://pogamut.cuni.cz/pogamut-devel/doku.php?id=human-like_artifical_agents_2013-14_summer_semester`

[17] Rabin, Steve: *AI Game Programming Wisdom*. Hingham, MA: Charles River Media, 2002.

[18] Tuma, Vojtěch: *Role of communication in team-oriented FPS games*. Bachelor's thesis, Charles University in Prague, Faculty of Mathematics and Physics. Prague, 2013.

[19] Unreal Engine Wiki: *Bot Skill Levels* [online]. 2007 [cit. 2014-07-05].
`http://wiki.beyondunreal.com/Legacy:Bot_Mind#Skill_Levels`
`http://liandri.beyondunreal.com/Bot`

List of Abbreviations

- AI – Artificial Intelligence
- FPS – first-person shooter, computer games genre
- UT2004 – Unreal Tournament 2004, an FPS computer game
- CTF – Capture the Flag, a game mode for UT2004
- BOD – Behaviour Oriented Design, a methodology for AI specification
- POSH [planner] – Parallel-rooted, Ordered Slip-stack Hierarchical [planner]

Appendix A: Resulting yaPOSH plan

The resulting yaPOSH plan is too large for its visual representation to fit in the document and is best viewed in the Shed editor in NetBeans (see Attachment B on page 59).

Here, we present the resulting yaPOSH plan in its text form. Refer to section 2.2 for yaPOSH plan documentation, description of the plan elements and a plan example with both visual and textual representations.

Syntax

yaPOSH uses a Lisp-like postfix notation and the elements of the aggregates are listed by their priority — with higher priority first. The lower the priority gets, the lower the respective element is.

For example the Drive collection (the root of the plan) is written as:

```
(DC life
  (drives
    ((attack (trigger ((senses.CurrentRole "Attacker" ==))) attack))
    ((defend (trigger ((senses.CurrentRole "Defender" ==))) defend))
    ((roam (trigger ((senses.CurrentRole "Roamer" ==))) roam))
    ((nothing (trigger ((Succeed))) DoNothing))
  )
)
```

Which in the more common infix notation with if-then rules would correspond roughly to:

```
(DC life
  (drives
    if (senses.CurrentRole == "Attacker") trigger attack        (choice attack)
    else if (senses.CurrentRole == "Defender") trigger defend   (choice defend)
    else if (senses.CurrentRole == "Roamer") trigger roam       (choice roam)
    else trigger DoNothing                                      (choice nothing)
  )
)
```

Nevertheless, the main advantage of yaPOSH is the visualization of its plans in NetBeans and the Shed editor. It is highly recommended to view the plan there, as one can see the relations of the plan's nodes and consequently understand its meaning immediately.

Next, we include the textual representation of all the aggregates present in our resulting yaPOSH plan in the yaPOSH notation.

Drive collection

```
(DC life
  (drives
    ((attack (trigger ((senses.CurrentRole "Attacker" ==))) attack))
    ((defend (trigger ((senses.CurrentRole "Defender" ==))) defend))
    ((roam (trigger ((senses.CurrentRole "Roamer" ==))) roam))
    ((nothing (trigger ((Succeed))) DoNothing))
  )
)
```

The main Attacker competence

```
(C attack
  (elements
    ((enemyFlag-ourFlag_home-home (trigger
        ((senses.FlagState($team="enemy",$state="home") true ==)
        (senses.FlagState($state="home",$team="our") true ==))) attack-home-home))
    ((home-dropped (trigger ((senses.FlagState($state="home",$team="enemy") true ==)
        (senses.FlagState($state="dropped",$team="our") true ==)))
        attack-home-dropped))
    ((home-taken (trigger ((senses.FlagState($state="home",$team="enemy") true ==)
        (senses.FlagState($state="taken",$team="our") true ==))) attack-home-taken))
    ((dropped-home (trigger ((senses.FlagState($state="dropped",$team="enemy") true
        ==) (senses.FlagState($state="home",$team="our") true ==)))
        actions.GoToFlag($team="enemy")))
    ((dropped-dropped (trigger ((senses.FlagState($state="dropped",$team="enemy")
        true ==) (senses.FlagState($state="dropped",$team="our") true ==)))
        attack-dropped-dropped))
    ((dropped-taken (trigger ((senses.FlagState($state="dropped",$team="enemy") true
        ==) (senses.FlagState($state="taken",$team="our") true ==)))
        attack-dropped-taken))
    ((taken-home (trigger ((senses.FlagState($state="taken",$team="enemy") true ==)
        (senses.FlagState($state="home",$team="our") true ==))) attack-taken-home))
    ((taken-dropped (trigger ((senses.FlagState($state="taken",$team="enemy") true
        ==) (senses.FlagState($state="dropped",$team="our") true ==)))
        attack-taken-dropped))
    ((taken-taken (trigger ((senses.FlagState($state="taken",$team="enemy") true ==)
        (senses.FlagState($state="taken",$team="our") true ==))) attack-taken-taken))
  )
)
```

The main Defender competence

```
(C defend
  (elements
    ((have_enemy_flag (trigger ((senses.MyBotHoldsEnemyFlag)))
        actions.ChangeRole($role="attacker",$requestSwitch=true)))
    ((ourFlag-enemyFlag_home-home (trigger
        ((senses.FlagState($team="our",$state="home") true ==)
        (senses.FlagState($state="home",$team="enemy") true ==))) defend-home-home))
    ((home-dropped (trigger ((senses.FlagState($state="home",$team="our") true ==)
        (senses.FlagState($state="dropped",$team="enemy") true ==)))
        defend-home-dropped))
    ((home-taken (trigger ((senses.FlagState($state="home",$team="our") true ==)
        (senses.FlagState($state="taken",$team="enemy") true ==)))
        defend-home-taken))
```

```
    ((dropped-home (trigger ((senses.FlagState($state="dropped",$team="our") true
        ==) (senses.FlagState($state="home",$team="enemy") true ==)))
        defend-dropped-home))
    ((dropped-dropped (trigger ((senses.FlagState($state="dropped",$team="our") true
        ==) (senses.FlagState($state="dropped",$team="enemy") true ==)))
        defend-dropped-dropped))
    ((dropped-taken (trigger ((senses.FlagState($state="dropped",$team="our") true
        ==) (senses.FlagState($state="taken",$team="enemy") true ==)))
        defend-dropped-taken))
    ((taken-home (trigger ((senses.FlagState($state="taken",$team="our") true ==)
        (senses.FlagState($state="home",$team="enemy") true ==))) retrieve_our_flag))
    ((taken-dropped (trigger ((senses.FlagState($state="taken",$team="our") true ==)
        (senses.FlagState($state="dropped",$team="enemy") true ==)))
        retrieve_our_flag))
    ((taken-taken (trigger ((senses.FlagState($state="taken",$team="our") true ==)
        (senses.FlagState($state="taken",$team="enemy") true ==)))
        defend-taken-taken))
  )
)
```

The main Roamer competence

```
(C roam
  (elements
    ((attack (trigger ((senses.CloseToLosing)))
        actions.ChangeRole($role="attacker",$requestSwitch=false)))
    ((defend (trigger ((senses.CloseToWinning)))
        actions.ChangeRole($role="defender",$requestSwitch=false)))
    ((have_enemy_flag (trigger ((senses.MyBotHoldsEnemyFlag)))
        actions.ChangeRole($role="attacker",$requestSwitch=true)))
    ((chase_enemy_flag_carry (trigger ((senses.SeeFlagHolder($team="enemy") true
        ==))) chase_enemy_flag_carry))
    ((get_our_flag (trigger ((senses.FlagState($team="our",$state="home") true !=)
        (senses.ClosestToFlag($team="our") true ==))) actions.GoToFlag($team="our")))
    ((get_enemy_flag_closest (trigger ((senses.ClosestToFlag($team="enemy") true
        ==))) actions.GoToFlag($team="enemy")))
    ((stalemate (trigger ((senses.FlagState($state="held",$team="our") true ==)
        (senses.FlagState($state="held",$team="enemy") true ==)))
        actions.GoToBase($team="enemy")))
    ((retrieve_our_flag (trigger ((senses.FlagState($state="home",$team="our") true
        !=) (senses.FlagLocationKnown($team="our") true ==)))
        actions.GoToFlag($team="our")))
    ((attack_when_armed_up (trigger ((senses.FlagState($state="home",$team="enemy")
        true ==) (senses.Health 100 >=) (senses.Armor 100 >=)))
        actions.GoToBase($team="enemy")))
    ((roam pick_items))
  )
)
```

Competences

Attacker's competences

```
(C attack-home-home
  (elements
```

```
    ((heal_up (trigger ((senses.Health 75 <)))
        actions.PickItems($category="health",$part="any")))
    ((get_ready (trigger ((senses.ExtraWeaponsNumber 2 <)))
        actions.PickItems($category="weapon",$part="any")))
    ((assault_base assault_enemy_base))
  )
)

(C attack-home-dropped
  (elements
    ((return_our_flag (trigger ((senses.ClosestToFlag($team="our") true ==)))
        actions.GoToFlag($team="our")))
    ((assault_base assault_enemy_base))
  )
)

(C attack-home-taken
  (elements
    ((chase_enemy_flag_carry (trigger ((senses.SeeFlagHolder($team="enemy") true
        ==))) actions.FollowPlayer))
    ((assault_base assault_enemy_base))
  )
)

(C attack-dropped-dropped
  (elements
    ((return_our_flag (trigger ((senses.ClosestToFlag($team="our") true ==)))
        actions.GoToFlag($team="our")))
    ((get_enemy_flag actions.GoToFlag($team="enemy")))
  )
)

(C attack-dropped-taken
  (elements
    ((get_enemy_flag (trigger ((senses.ClosestToFlag($team="enemy") true ==)))
        actions.GoToFlag($team="enemy")))
    ((chase_enemy_flag_carry (trigger ((senses.SeeFlagHolder($team="enemy") true
        ==))) actions.FollowPlayer))
    ((get_enemy_flag_anyway actions.GoToFlag($team="enemy")))
  )
)

(C attack-taken-home
  (elements
    ((capture_enemy_flag (trigger ((senses.MyBotHoldsEnemyFlag))) capture_fast))
    ((attack (trigger ((senses.CloseToScoring))) assault_enemy_base))
    ((assist_our_flag_carry assist_our_flag_carry))
  )
)

(C attack-taken-dropped
  (elements
    ((get_our_flag (trigger ((senses.ClosestToFlag($team="our") true ==)))
        actions.GoToFlag($team="our")))
    ((capture (trigger ((senses.MyBotHoldsEnemyFlag))) capture_heal))
    ((assist_our_flag_carry assist_our_flag_carry))
  )
)
```

```
(C attack-taken-taken
  (elements
    ((capture (trigger ((senses.MyBotHoldsEnemyFlag))) capture_heal))
    ((chase_enemy_flag_carry (trigger ((senses.SeeFlagHolder($team="enemy") true
        ==))) actions.FollowPlayer))
    ((assist_our_flag_carry assist_our_flag_carry))
  )
)
```

Defender's competences

```
(C defend-home-home
  (elements
    ((heal_up (trigger ((senses.Health 75 <)))
        actions.PickItems($category="health",$part="our")))
    ((get_ready (trigger ((senses.ExtraWeaponsNumber 3 <)))
        actions.PickItems($category="weapons",$part="our")))
    ((defend_base actions.DefendBase))
  )
)

(C defend-home-dropped
  (elements
    ((get_enemy_flag (trigger ((senses.FlagVisible($team="enemy") true ==)))
        actions.GoToFlag($team="enemy")))
    ((go_to_enemy_flag (trigger ((senses.ClosestToFlag($team="enemy") true ==)))
        actions.GoToFlag($team="enemy")))
    ((defend_base actions.DefendBase))
  )
)
(C defend-home-taken
  (elements
    ((defend_base actions.DefendBase))
  )
)

(C defend-dropped-home
  (elements
    ((get_our_flag (trigger ((senses.FlagLocationKnown($team="our") true ==)))
        actions.GoToFlag($team="our")))
    ((assault_enemy_base actions.GoToBase($team="enemy")))
  )
)

(C defend-dropped-dropped
  (elements
    ((get_our_flag (trigger ((senses.FlagVisible($team="our") true ==)))
        actions.GoToFlag($team="our")))
    ((get_enemy_flag (trigger ((senses.ClosestToFlag($team="enemy") true ==)))
        actions.GoToFlag($team="enemy")))
    ((retrieve_our_flag (trigger ((senses.FlagLocationKnown($team="our") true ==)))
        actions.GoToFlag($team="our")))
    ((find_enemy_flag (trigger ((senses.FlagLocationKnown($team="enemy") true ==)))
        actions.GoToFlag($team="enemy")))
    ((assault_base actions.GoToBase($team="enemy")))
  )
)
```

```
(C defend-dropped-taken
  (elements
    ((get_our_flag (trigger ((senses.FlagLocationKnown($team="our") true ==)))
        actions.GoToFlag($team="our")))
    ((chase_enemy (trigger ((senses.SeePlayer($team="enemy") true ==)))
        actions.FollowPlayer))
    ((assault_base actions.GoToBase($team="enemy")))
  )
)

(C defend-taken-taken
  (elements
    ((chase_enemy_flag_carry (trigger ((senses.SeeFlagHolder($team="enemy") true
        ==))) actions.FollowPlayer))
    ((find_our_flag (trigger ((senses.FlagLocationKnown($team="our") true ==)))
        actions.GoToBase($team="enemy")))
    ((assault_base actions.GoToBase($team="enemy")))
  )
)
```

General competences

```
(C assault_enemy_base
  (elements
    ((go_to_base actions.GoToBase($team="enemy")))
  )
)

(C assist_our_flag_carry
  (elements
    ((follow_our_flag_holder (trigger ((senses.SeeFlagHolder($team="our") true ==)))
        actions.FollowPlayer))
    ((go_to_enemy_flag actions.GoToFlag($team="enemy")))
  )
)

(C retrieve_our_flag
  (elements
    ((chase_enemy_flag_carry (trigger ((senses.SeeFlagHolder($team="enemy") true
        ==))) chase_enemy_flag_carry))
    ((get_enemy_flag (trigger ((senses.FlagVisible($team="enemy") true ==)))
        actions.GoToFlag($team="enemy")))
    ((find_our_flag (trigger ((senses.FlagLocationKnown($team="our") true ==)))
        actions.GoToBase($team="enemy")))
    ((assault_base actions.GoToBase($team="enemy")))
  )
)

(C capture_fast
  (elements
    ((adrenalin (trigger ((senses.Adrenaline 100 >=)))
        actions.AdrenalineCombo($type="defensive")))
    ((capture actions.GoToBase($team="our")))
  )
)

(C capture_heal
  (elements
```

```
    ((adrenalin (trigger ((senses.Adrenaline 100 >=)))
        actions.AdrenalineCombo($type="defensive")))
    ((capture actions.GoToBase($team="our")))
  )
)

(C chase_enemy_flag_carry
  (elements
    ((adrenalin (trigger ((senses.Adrenaline 100 >=)))
        actions.AdrenalineCombo($type="berserk")))
    ((chase actions.FollowPlayer))
  )
)

(C pick_items
  (elements
    ((get_powerups (trigger ((senses.CanGetPowerup($part="middle") true ==)))
        actions.PickItems($category="powerup",$part="middle")))
    ((get_weapons (trigger ((senses.ExtraWeaponsNumber 3 <)))
        actions.PickItems($category="weapon",$part="middle")))
    ((get_health (trigger ((senses.Health 75 <)))
        actions.PickItems($category="health",$part="middle")))
    ((roam actions.GoToMapPart($part="middle")))
  )
)
```

Appendix B: CD contents

The attached CD contains:

- `Thesis.pdf` – this thesis.
- `/netbeans/` – NetBeans 7.4 installation package.
- `/pogamut/` – Pogamut 3.6.1 installation package.
- `/src/` – Source code of the resulting CTF bot team, its executable .jar files and software documentation.

Appendix C: User guide

In this attachment, we will describe how our CTF team can be installed, run and modified.

Installation

1. Install NetBeans 7.4 (`/netbeans/` folder on the CD).
2. Install Unreal Tournament 2004[1].
3. Install Pogamut 3.6.1 (`/pogamut/` folder).

Starting the server

1. Run UT2004. Select Host Game, GameBots CTF Game, specify map (the tested maps were CTF-Citadel, CTF-Geothermal and CTF-Lostfaith), and specify Game Rules (the rules we used are described in table 5.1).
   The Bot Skill setting can be set from 0 (Novice) to 7 (Godlike) and will only affect the native bots.
2. Click on Listen to start the server.
3. When in the game, press Escape and select Spectate to spectate the game.

Only after starting the server can any bots be run.

Running our bots

Our bots can be launched by running their compiled .jar file: `/src/CTFBot/target/CTFBot-3.6.1.one-jar.jar`. By default, their skill level is 7, team size is 5 and their side is red.

To launch the .jar file, use command `java -jar CTFBot-3.6.1.one-jar.jar` in the `/target/` folder.

The bots can be also run by opening the project folder in NetBeans (`/src/CTFBot`) and by pressing the F6 key or selecting Run Project.

For additional information about the Pogamut platform and basic tutorials, refer to the Pogamut lectures website [16].

[1] As UT2004 is a commercial application, it cannot be provided with the thesis. UT2004 can be bought for example at
`http://store.steampowered.com/app/13230/` or
`http://www.gog.com/game/unreal_tournament_2004_ece`.

Modifying our bots

In order to modify our bots' parameters, open the project in NetBeans (`/src/CTFBot`) and open the `CTFBotLogic.java` class file.

Modify the following parameters (all these parameters only affect our bots and not the native bots):

- `SKILL_LEVEL` - skill level of our bots, 0 to 7 (default is 7).
- `TEAM_SIZE` - number of bots in our team, recommended and default value is 5.
- `START_TWO_TEAMS` - whether our bots should play against another team of our bots. True for two teams, false for one team when wanting to play against other (native) bots.
- `TEAM` - whether our bots should be playing as the red or the blue team. Possible values: 0 and 1 or `AgentInfo.TEAM_RED` and `AgentInfo.TEAM_BLUE`.

When modified, press the F6 key or select Run Project to run the modified bots.

Running native bots

The native bots are most comfortably run by opening the in-game console which is done by pressing the '`' or ';' key above the Tab key.

Then, one can launch `n` native bots by typing `ADDBOTS n` and pressing Enter. To remove all the bots, write `KILLBOTS` into the console.

Skill level of the native bots is determined by the server game settings.