Citation for published version:
Partington, SJ 2005, *A critical analysis of Behaviour-Oriented Design (BOD), based on experiences in using it to create an Unreal Tournament Capture-the-Flag (CTF) team*. Computer Science Technical Reports, no. CSBU-2005-05, University of Bath, Department of Computer Science.

Publication date:
2005

Link to publication

©The Author May 2005

University of Bath

Alternative formats

If you require this document in an alternative format, please contact:
openaccess@bath.ac.uk

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 15. Oct. 2024

Department of Computer Science

Technical Report

Undergraduate Dissertation: A critical analysis of Behaviour-Oriented Design (BOD), based on experiences in using it to create an Unreal Tournament Capture-the-Flag (CTF) team.

Samuel J. Partington

Technical Report 2005-05 **May 2005**

ISSN 1740-9497

Copyright ©May 2005 by the authors.

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

A critical analysis of Behaviour-Oriented Design (BOD), based on experiences in using it to create an Unreal Tournament Capture-the-Flag (CTF) team.

Samuel J. Partington

BSc (Hons) in Computer Science, 2005

A CRITICAL ANALYSIS OF BEHAVIOUR-ORIENTED DESIGN (BOD), BASED ON EXPERIENCES IN USING IT TO CREATE AN UNREAL TOURNAMENT CAPTURE-THE-FLAG (CTF) TEAM.

submitted by Samuel Partington

COPYRIGHT

Attention is drawn to the fact that the copyright of this thesis rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).

This copy of the thesis is has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

Declaration

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

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Abstract

Computer Science is full of interesting ideas, but without evaluation and testing there is no way of determining a good idea from one which has simply been well-publicised. This dissertation provides an evaluation of Bryson's (2001) Behaviour Oriented Design [BOD], a methodology for the development of complex agents. This evaluation is based on experiences gained whilst developing a computer-controlled player ("bot") for the game Unreal Tournament, and on comparisons with existing architectures and methodologies. BOD is shown to be a useful and powerful methodology, applicable to a wide range of situations. The document also describes the development of the bot and its eventual performance, highlighting issues relating to BOD and to game-agent development in general.

The development process made use of the Gamebots interface (Kaminka et al., 2002) and the PyPOSH implementation of BOD's Action Selection mechanism (Kwong, 2003), both of which are described and commented upon here. Improvements made to PyPOSH are also discussed.

Acknowledgements

My thanks to my supervisor Dr. Bryson for her encouragement, support and useful ideas, and to Andy Kwong whose work was invaluable to me. I'd also like to thank contributors at WordPerfect Universe for their help with some document formatting issues and, on a similar note, to apologise to those who felt offended by my decision not to use LaTeX. (Don't take of it as a criticism of LaTeX; it's not meant as such.) Finally, I'd like to thank my parents and friends for all their support.

Contents

1. Introduction 10
   1.1. Contributions, Key Concepts, and Justification 10
      1.1.1. Behaviour-Oriented Design [BOD] 10
      1.1.2. Unreal Tournament [UT] and Capture the Flag. 11
      1.1.3. Justification 11
   1.2. Document Structure 12
      1.2.1. Roadmaps 12

2. Introduction to Behaviour-Oriented Design 14
   2.1. Introduction 14
   2.2. Development Process and Development Principles 14
   2.3. Behaviours 15
   2.4. Action Selection 16
      2.4.1. Basic Reactive Plans 16
      2.4.2. Sensory and Action Primitives 16
      2.4.3. Action Patterns 17
      2.4.4. Competences 17
      2.4.5. Drive Collections 17
   2.5. A defence of pre-written plans 17
      2.5.1. Pre-written plans are both biologically plausible and sufficiently reactive. 17
      2.5.2. Existing AI is not yet sufficiently advanced 18
   2.6. PyPOSH and Python 18

3. Introduction to Other Architectures 19
   3.1. Introduction 19
   3.2. Soar 19
      3.2.1. Goal Context 19
      3.2.2. Working Memory 20
      3.2.3. Long-Term Memory 20
      3.2.4. The Perception / Motor Interface 20

3.2.5. The Decision Cycle 20
3.2.6. Impasses 21
3.2.7. Chunking 21
3.2.8. Summary 21
3.3. EPIC [Executive Process-Interactive Control] 22
3.3.1. EPIC in Theory 22
3.3.2. EPIC in Practice 22
3.3.3. Production Rule example 23
3.3.4. Summary 23
3.4. ACT-R 24
3.4.1. Declarative Memory and Pattern Matching 24
3.4.2. Procedural Memory 26
3.4.3. Learning in ACT-R 27
3.5. Summary 28

4. The Bot in Action 29
4.1. Introduction 29
4.2. The Scenarios 29
4.2.1. Walking To Navigation Points 29
4.2.2. A Greater Awareness of Flags 31
4.2.3. Responding to Attack 34
4.3. Summary 36

5. The Development Process 37
5.1. Introduction 37
5.2. Initial Behaviour Decomposition 37
5.2.1. The Process 37
5.2.2. My Experience of the Process 37
5.3. Evolutionary Design and Development 38
5.3.1. The Process 38
5.3.2. Behaviour Modules, State and Utility Functions 38
5.3.3. The Primitives 40
5.3.4. Problems Encountered 44

5.3.5. Evaluation of the BOD Process 46
5.3.6. POSH's Contribution to Development 48
5.4. Summary 50

6. Comparison with Other Architectures 51
6.1. Introduction 51
6.2. The Rational Rose Approach 51
6.3. Getting Started with Development 51
6.4. Evolutionary Design and Development 53
6.5. Goal-Driven Development 54
6.6. The Architecture 54
6.6.1. POSH's Hierarchical Structure 54
6.6.2. Priorities and Emergent Behaviour 55
6.6.3. Frequency and Retries 55
6.6.4. Modularity and Re-use 55
6.6.5. Explicit Goals 57
6.7. Further Analysis of the Rational Rose Approach 58
6.8. Comparisons with the Subsumption Architecture 58
6.8.1. Incremental Development and Behaviour-Interaction 58
6.8.2. Robustness 60
6.8.3. Overall Comparison 60

7. General BOD Evaluation 61
7.1. Introduction 61
7.2. The Methodology 61
7.2.1. Lessons from Extreme Programming 61
7.3. POSH Plans 63
7.3.1. Syntax 63
7.3.2. Sharing and Redundancy 64
7.3.3. Parallel Plan Elements 67
7.3.4. Improvements to Documentation 67
7.4. Testing POSH against Action-Selection Criteria 68
7.4.1. Dealing with all types of sub-problem 68

7.4.2. Contiguous action sequences 69
7.4.3. Compromise candidates 69
7.4.4. Conclusion 71

8. PyPOSH: Problems, Alterations, Corrections and Recommendations 72
8.1. Multiple Behaviour Files 72
8.1.1. How to Write a Behaviour Module 72
8.2. Problems Corrected 73
8.2.1. Timeouts for Drive Collection Elements 73
8.2.2. Retry Limits for Competence Elements 74
8.3. Other Issues and Suggestions 74
8.3.1. Debugging 74
8.3.2. Profiling 74
8.3.3. Issues with Primitives 75
8.4. Distribution 75

9. Conclusions 76
9.1. Summary of Achievements 76
9.1.1. A Discussion of Architectures 76
9.1.2. The Bodbot Project 76
9.1.3. An Evaluation of BOD, and Improvements to PyPOSH 76
9.2. Future Work, and Limitations of this Dissertation 76
9.2.1. Significant Developments 76
9.2.2. Tweaks 78
9.2.3. Further Evaluation and Development of BOD 78
9.2.4. Other Work 79
9.3. Summary of Evaluation 79
9.3.1. Methodology 79
9.3.2. POSH Action Selection 79
9.3.3. Plan Files 80
9.3.4. Overall Summary 80

Bibliography 81

Appendix A: Sample Plan Files 85

Appendix B: Initial Behaviour Decomposition 90

Appendix C: E-mail from John Laird 94

Appendix D: Readme from Distribution 95

Appendix E: Code Listings and CD Contents 99

1. Introduction

Computer Science is full of interesting ideas, but without evaluation and testing there is no way of determining a good idea from one which has simply been well-publicised. This dissertation provides an evaluation of Bryson's (2001) Behaviour Oriented Design [BOD], a methodology for the development of complex agents. This evaluation is based on experiences gained whilst developing a computer-controlled player ("bot") for the game Unreal Tournament, and on comparisons with existing architectures and methodologies. BOD is shown to be a useful and powerful methodology, applicable to a wide range of situations. The document also describes the development of the bot and its eventual performance, highlighting issues relating to BOD and to game-agent development in general.

The development process made use of the Gamebots interface (Kaminka et al., 2002) and the PyPOSH implementation of BOD's Action Selection mechanism (Kwong, 2003), both of which are described and commented upon here. Improvements made to PyPOSH are also discussed.

1.1. Contributions, Key Concepts, and Justification

The principal contributions of this dissertation are:

- The development, using Behaviour-Oriented Design, of a bot for Unreal Tournament's Capture-the-Flag mode, and a discussion of this development. The POSH plan developed as part of this is more complex than any existing published POSH plan. (POSH is BOD's Action Selection Mechanism, introduced in section 2.4).
- An evaluation of BOD in light of this development and with reference to existing architectures and methodologies for Agents and Artificial Intelligence. These include Soar (Lehman et al. 1996), EPIC (Kieras and Meyer, 1997), ACT-R (ACT-R Research Group, 2004a), the Subsumption Architecture (Brooks, 1986), JACK (Howden et al., 2001), agent modelling via UML and Rational Rose (section 6.2), Kinny et al.'s (1996) Beliefs-Desires-Intentions model, and Tyrrell's (1993) criteria for Action Selection Mechanisms.

Additional contributions of this dissertation include the following:

- A summary and evaluation of a number of Artificial Intelligence / Agents architectures and methodologies.
- Improvements to the PyPOSH implementation of BOD's Action Selection mechanism.

1.1.1. Behaviour-Oriented Design [BOD]

Behaviour-Oriented Design (Bryson, 2001) is a methodology for the development of complex agents. There is much debate over the exact definition of an "agent", but a well-respected definition is that given by Wooldridge (2002, p.15; his emphasis):

> An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives.

Bryson (2001) defines a *complex agent* as one which must deal with goals and behaviours which may be conflicting.

BOD is both methodological and architectural: it specifies an iterative development process, a system of behaviour decomposition and a mechanism for Action Selection (POSH). BOD draws on ideas from Behaviour-Based AI (e.g., Matarić, 1997) and Object-Oriented Design (Booch, 1990). It is introduced in detail in chapter 2.

1.1.2. Unreal Tournament [UT] and Capture the Flag.

Unreal Tournament (Epic Games, 2004) is a First-Person Shooter [FPS] game. As the name suggests, the viewpoint adopted by the player in FPS games is that of the character he or she is controlling: the player sees the world through the character's eyes. (Some FPS games also offer a third-person "over the shoulder" view option.)

The single-player version of Unreal Tournament pits the human player against computer-controlled players ("bots") in kill-or-be-killed deathmatches spread over a wide range of expansive 3D environments. The aim of the game is to defeat all competitors in every arena. The multi-player mode is similar, except that it is other human players who are fought, rather than computer-controlled ones. The complete storyline for the game can be found at *http://www.planetunreal.com/utguide/story.shtml*.

This project, however, concentrates on a specific game-mode within UT: Capture the Flag [CTF]. In this mode, two teams (or possibly two single players) compete against each other. Each team has a base in which their flag is located. The object of the game is to obtain your opponents' flag (done by running into it), and return with it to your flag. This counts as a flag capture. Once a specified number of captures have been achieved, the game is won.

However, the opposing team can obviously capture your flag also. In this case, you must recover it before you can make a successful capture, as returning to your base with the enemies' flag achieves nothing if your own team's flag is not there. Once a player has captured a flag, s/he may be forced to drop it by being killed (using the usual UT weaponry). The flag then lies on the ground waiting for someone (of either team) to pick it up. If you pick up your own flag dropped by an escaping enemy, it returns to your base instantly.

Teams in CTF may be composed of human players alone, or of a mixture of human and computer players.

1.1.3. Justification

This project is significant for a number of reasons. The primary reason is that BOD has not previously been evaluated in detail except by its developer. It is important that developments in any scientific field are critically evaluated if they are to be truly understood and reach their full potential. Similarly, the PyPOSH implementation of POSH Action Selection has not previously been evaluated by a third-party.

Furthermore, the bot developed as part of the implementation phase of this project has a more complex POSH plan than any published to date (see section 5.3.6). The development therefore provides much insight into the effectiveness and scalability of all aspects of the BOD process and acts as evidence of the power of the methodology.

Finally, this dissertation provides information useful in a number of areas: Evaluation of BOD is a major part, but this project also contributes to existing literature on the

development of agents and specifically of agents for computer games (e.g. Laird and Duchi, 2000). The evaluation itself is relevant from both a theoretical and a practical perspective.

Are computer games really a valid field for the exploration of Artificial Intelligence?

Although computer gaming may appear a trivial use of computers, computer games' contribution to AI research is far from insignificant. Laird and van Lent (2000) go so far as to describe interactive computer games as the "Killer Application" for pursuing the goal of Human-level AI. However, the usefulness of computer games extends far beyond just pursuing this particular goal. McCarthy (1998; p.1), for example, describes using the game *Lemmings* as an ideal model for "AI research connecting logical formalizations with information that is incompletely formalizable in practice", whilst Veksler and Gray (2004) have made use of the *Tetris* game for experiments with reinforcement learning.

A key motivation for the use of computer games in AI research is that they offer complex interactive environments in which artificially-intelligent agents can operate. These environments are usually highly customizable and can often be created from scratch to suit particular needs. The cost and complexity of using such an environment in comparison to, say, robotics are negligible.

Readers interested in further justification are encouraged to read Laird and van Lent's article (2000). In particular, it offers a list of "reasons for AI researchers to take the computer game industry seriously" (p. 1172).

1.2. Document Structure

This document is made up of three main sections:

- Chapters 2 and 3 summarise and comment on most of the **architectures** studied. Chapter 2 introduces BOD whilst other architectures are introduced in Chapter 3.
- Chapters 4 and 5 discuss the **development process and the bot produced**, including some brief evaluation.
- Chapters 6, 7 and 8 provide the **evaluation** of BOD and a brief evaluation of PyPOSH, the Python implementation of POSH. The evaluation is summarised as part of the **conclusions** in chapter 9.

This document contains no explicit "Literature Review" section. Rather, the work done for my Literature Survey is spread throughout the document, much of it in chapters 2 and 3.

1.2.1. Roadmaps

If you just wish to read about **different architectures**, chapters 2 and 3 are the most important ones to read. You should also read section 6.8 (the Subsumption Architecture). You may also like to read about JACK in section 6.6.4, and the "Rational Rose approach" in section 6.2.

If your main interest is **the development parts of this project**, you should read the scenarios in chapter 4, as these describe the behaviour of the created bot. You should definitely read chapter 5, the primary place where the development process is discussed. You may find section 2.4 (an explanation of POSH action selection) useful in your understanding of chapter 4, as this chapter makes much reference to POSH plans and action selection. You should also read Appendix B which describes the bot's specification, and possibly Appendix D, as this discusses part of the development of the Soar Quakebot (Laird

and Duchi, 2000). Chapter 8 is probably worth reading too, as part of the development time was spent working on modifications to PyPOSH.

If your interest is only in **the methodological side of BOD**, you should read sections 2.1 and 2.2 (although these might make more sense if you understand BOD fully, so you may find it useful to read all of that chapter). You should then read chapter 5, probably omitting the section which discusses POSH (5.3.6), and read the sections of evaluation which detail specifically with the methodology (6.3 and 6.4). The discussion with reference to Extreme Programming in section 7.2.1 would also be worth reading.

Finally, if your interest in BOD is limited to **POSH action selection**, you should begin by reading chapter 2 (probably omitting section 2.2). Chapter 4 describes the behaviour of the bot in relation to the plans created and so should definitely be read. POSH's contribution to the development process is discussed in section 5.3.6 whilst the comparisons in section 6.6 deal with action selection. POSH plan files are evaluated in section 7.3, whilst section 7.4 tests POSH against Tyrrell's (1993) criteria for Action Selection mechanisms. Section 8.2 of the chapter on PyPOSH might also be useful: it discusses some of the problems corrected with this implementation of POSH. Finally, Appendix A gives a number of sample plan files.

2. Introduction to Behaviour-Oriented Design

2.1. Introduction

Introduced in Bryson (2001), Behaviour-Oriented Design [BOD] is a methodology for developing complex agents. Bryson (2001, p. 59) summarises the components of BOD as follows (her emphasis):

> Behaviour-oriented design consists of three equally important elements:
>
> - an iterative [development] process
> - parallel, modular *behaviours*, which determine **how** an agent behaves, and
> - *action selection*, which determines **when** a behaviour is expressed.

This chapter explores these three elements in more detail. It also provides a theoretical defence of pre-written plan files, introduces the PyPOSH Python implementation of POSH and introduces the Python language itself.

2.2. Development Process and Development Principles

BOD aims to facilitate rapid, efficient and simple development. These goals are the driving force of the development process itself, and are the *raison d'etre* of BOD's development principles.

The design process begins with the initial behaviour decomposition, discussed in section 2.3 below. The core of the design process, however, is the following iterative sequence (Bryson, 2001, p. 120):

> 1. Select a part of the specification to implement next.
> 2. Extend the agent with that implementation:
>   - code behaviors [sic] and reactive plans, and
>   - test and debug that code.
> 3. Revise the current specification.

The iterative nature of this process allows for *rapid prototyping*: different approaches can be tried, rolling back to the previous iteration if some development is found to be ineffective.

The key principle for revision of specifications is "when in doubt, favour simplicity" (*ibid*, p. 121). In practice, this means using the simplest action-selection component possible in any given situation, refactoring action-selection components to avoid overly complex or overly long elements, and both eliminating redundancy and maximising re-use.

Bryson (2001) gives three further recommendations to make developing with BOD as efficient and effective as possible:

1. Document the agent specification in program code
2. Use a Revision-Control system
3. Use debugging tools.

These recommendations and others are discussed in more detail in Chapter 8 of *ibid*.

2.3. Behaviours

Separating a large system into more manageable subsystems is a long-established software-engineering principle, and behaviour decomposition is an effective way of breaking down the capabilities of an agent. Brooks (1991; p. 87) describes it succinctly (his emphasis):

> [The] fundamental slicing up of an intelligent system is ... dividing it into *activity* producing subsystems. Each activity, or behaviour producing system, individually connects sensing to action.

Contrast this with applying the principle of breaking down a system into areas of specific *functionality*: "One needs a long chain of modules to connect perception to action. In order to test any of them they must all first be built. But until realistic modules are built it is highly unlikely that we can predict exactly what modules will be needed or what interfaces they will need." (*ibid*; p. 87).

However, this activity-based approach to decomposition is not without its problems, primarily in the interaction between the behaviour modules. This can be seen in Behaviour-Based AI (for example, Matarić, 1997) where a high level of coupling is required to ensure that the right module runs at the right time (by having these modules interact to suppress or activate each other, for example). BOD offers a way around this problem: its hierarchical reactive plans mean that behaviours can run without worrying about the interactions of other behaviours, yet with the assurance that should some more important situation arise, the relevant behaviour will be triggered to handle it. This is particularly significant as it means that the developer is free to concentrate on what the agent actually does.

As with other aspects of the design process, BOD provides guidelines to follow for initial behaviour decomposition (Bryson, 2001; pp. 119-120):

> 1. Specify at a high level what the agent is intended to do.
> 2. Describe likely activities in terms of sequences of actions. These sequences are the basis of the initial reactive plans.
> 3. Identify an initial list of sensory and action primitives from the previous list of actions.
> 4. Identify the state necessary to enable the described primitives and drives. Cluster related state elements and their primitives into specifications for behaviors [sic]. This is the basis of the behavior library.
> 5. Identify and prioritize goals or drives that the agent may need to attend to. This describes the initial roots for the POSH action selection hierarchy [explained below].
> 6. Select a first behavior to implement.

An important point is that BOD's emphasis on revision of specifications means that no assumption is made that the initial decomposition is perfect. The developer is free to modify the decomposition if new issues arise, and the lack of coupling between components means that this can be easily achieved.

The use of behaviours in BOD draws inspiration from Object-Oriented Design [OOD] (Booch, 1990). Specifically, behaviours are usually coded as objects, and the basic actions in Reactive Plans (see section 2.4.2) are then method calls on these objects. OOD also

influences BOD's iterative design process where, as described above, cyclic design with rapid prototyping is emphasised.

I imagine that this use of OOD would be a contributing factor to the acceptance of BOD as a methodology: it is a technique with which the majority of software engineers would already be familiar, and one which has proven successful and useful. The advantages (and disadvantages) of drawing on OOD for agent methodologies is discussed further in Inglesias et al. (1999).

2.4. Action Selection

Action Selection is the mechanism which controls when actions are executed. In BOD this is controlled by Parallel-rooted Slip-stack Hierarchical [POSH] reactive plans, described below. Note that "reactive planning" is something of a misnomer as reactive planning is not planning in the traditional sense (Nilsson, 1998 pp. 117-214). A more accurate phrase would be "reactive action selection" (Bryson, 2001; p. 60).

2.4.1. Basic Reactive Plans

Basic Reactive Plans [BRPs] are the theoretical basis for many of the elements used in POSH plans. BRPs are ordered groups of *sense*→*action* pairs (production rules). Figure 2.1 is an example.

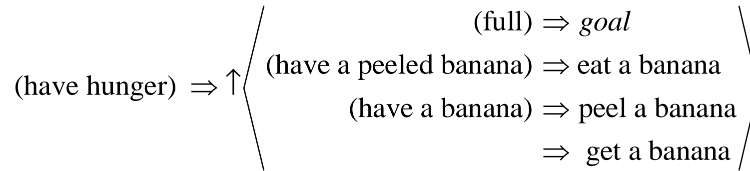

Figure 2.1 (from Bryson, 2001)

This BRP groups together all the rules to do with the actions to be performed when the agent has hunger. The rules have a priority (increasing in the direction of the vertical arrow). The rule which fires is that which has the highest priority of all those whose preconditions are satisfied. Note the *goal* element. This indicates that the task is complete, and the BRP terminates when it fires. BRPs can also terminate due to none of their conditions being able to fire.

Plans which contain sub-plans are called *hierarchical*. It has been argued (e.g. Maes, 1991) that hierarchical approaches to action selection are overly restrictive, and that full parallelism is a superior approach. This argument is refuted in Bryson (2000c).

POSH plans themselves are made up of 5 types of component: Sensory Primitives, Action Primitives, Action Patterns, Competences and Drive Collections. The plans are written in Bryson's own LAP [Learnable Action Pattern] format, which uses a Lisp-like syntax.

2.4.2. Sensory and Action Primitives

Sensory and Action primitives are the lowest-level components in POSH, specifying how the available actions and senses should be carried out. For example, these may be coded as methods in a behaviour module.

2.4.3. Action Patterns

Action Patterns are the simplest type of plan, being simply a list of actions. For example:

⟨ get a banana → peel a banana → eat a banana ⟩ *Figure 2.2 (from Bryson, 2001)*

2.4.4. Competences

Competences are a special form of BRP with two additions: (1) competences offer the option to specify a maximum number of retries for each action, and (2) competences return a value: $\top$ if the competence ends because the goal-trigger fires, and $\bot$ if it ends because no element can fire.

2.4.5. Drive Collections

A Drive Collection is also a special type of BRP, usable only as the top level of the hierarchy. Whilst each BRP element contains a priority, releaser (i.e. preconditions) and the action itself, Drive Collection elements also contain a maximum frequency at which the element fires (e.g. the element may only fire at most once per two minutes, even if its preconditions are met more often than that). Drive Collection elements also contain a fifth item: A, the root of a BRP Hierarchy.

The action element, α, in a Drive Collection points to the *currently active* element of A. This allows Drive Collections to force the plan execution to remain focussed on a lower-level plan element: if α turns out to be a competence which then triggers a competence or action-pattern β, then action α is set to point to the root of β (hence the "Slip-stack" in "POSH"). Altering where α points means that next time the competence element fires, control jumps straight to this sub-element. Control is returned to the higher element when β terminates. This return of control is accomplished by setting α to point to the root of A, which is where it points when then element runs for the first time.

One further difference between Drive Collections and BRPs or Competences is that multiple drives (i.e. Drive Collection elements) may effectively be active simultaneously (hence "Parallel-rooted" in "POSH"). For example, whilst one drive is pointing to some lower-level task which is part way through being executed, a higher-priority drive may fire instead, temporarily moving control into that drive.

2.5. A defence of pre-written plans

Some might argue that by pre-writing reactive plans, we are not creating a system which is "artificially intelligent" at all. (For instance, it could be argued that the system itself should write its own plans.) However, this is not the case. There are two key points here:

2.5.1. Pre-written plans are both biologically plausible and sufficiently reactive.

Section 4.2.2 of Bryson (2001) cites much evidence to support these points, which I shall not repeat here. Bryson (2000a) also studies the issue of biological plausibility in great detail; its central hypothesis is that "natural, appropriate and alert behaviour can be exhibited by agents executing actions prestored as hierarchies and sequences" (p. 33).

Furthermore, it is important to note that pre-coding of plans in no way compromises an agent's *autonomy*. Consider Barber and Martin's (1999; p. 9) comprehensive definition of autonomy, which draws together ideas from a wide range of existing sources:

> An agent's degree of autonomy, with respect to some goal that it actively uses its capabilities to pursue, is the degree to which the decision-making process, used to

determine how that goal should be pursued, is free from intervention by any other agent.

In choosing actions via POSH plans, it is clear that the agent is acting without any intervention and thus is autonomous.

2.5.2. Existing AI is not yet sufficiently advanced

Whilst it is a dream for many to be able to release a basic agent into an environment and have it learn everything necessary without any pre-written plans, the science of AI is still a long way from achieving this (after all, it took humans four billion years of evolution!). In fact, BOD is arguably an excellent compromise of realism and idealism, as it does enable a large amount of learning (e.g. Bryson, 2001, Ch. 7) and, through the non-linear execution of actions brought about by structures such as competences, facilitates much in the way of emergent behaviour.

Furthermore, the "learn everything" approach would require a huge amount of search to create its own plans. Wolpert and Macready's (1996) "no free lunch" theorems claim that for any algorithm (and thus for search such as this), a general-purpose optimal solution is impossible: an improvement in performance on one set of problems is matched exactly by a decrease in performance for some other set. This places severe constraints on the feasability of agents which do not use any pre-written plans.

2.6. PyPOSH and Python

PyPOSH (Kwong, 2003) is "an agent framework built in Python (Davids, 1997) which supplies POSH action selection mechanisms for agents" (p. 25). Using PyPOSH with the Gamebots interface (Kaminka et al., 2002) allows agents in Unreal Tournament to be guided by POSH plans to perform actions coded in the Python programming language. A simple agent, "poshbot", was developed as part of PyPOSH's development, and the agent I have developed builds on this. My choice of Python as a programming language is driven by the fact that PyPOSH is written to use behaviours coded in Python. A rationale behind the use of Python for PyPOSH can be found in Kwong (2003), pp. 17-21.

Python is a general purpose Object-Oriented language which includes garbage collection. Python programs are highly portable, as (like programs written in Java, for example) they compile to an intermediate language which is then run by an interpreter. Python is also embeddable and extendable, able to load compiled files, source files or object code dynamically.

One interesting syntactic aspect of Python is that whitespace is not ignored. Rather, indentation is used to define blocks. Another is the interesting aspect is Python's expression evaluation, which allows expressions such as the following (Davids, 1997):

`a = b = c = 0` *multiple assignments*

`[a, b] = ['a', 'b']` *multiple different assignments*

`w < x < y < z` *multiple range testing*

3. Introduction to Other Architectures

3.1. Introduction

Since one of the primary parts to this project is an evaluation, it is desirable to have alternatives to compare against, as well as just experience in the methodology to be evaluated. Some of these alternatives, primarily Soar, EPIC and ACT-R, are outlined below. These architectures are those studied as part of the initial Literature Survey I performed. Although, unlike BOD, these are all cognitive architectures, I chose to study in these detail as they are more complex than most other architectures.

Other architectures were studied as part of the later comparisons and these are introduced at the relevant stages of this document. These other architectures include the Subsumption Architecture, JACK, agent modelling via UML and Rational Rose, and Kinny et al.'s (1996) Beliefs-Desires-Intentions model.

3.2. Soar

The Soar architecture (Lehman et al. 1996) was designed for modelling human cognition, with the specific aim of creating a unified theory of cognition [UTC]. Work on Soar began around 1980. Soar projects have included at least one similar to my work with UT: The Soar Quakebot (Laird and Duchi, 2000).

The Soar architecture is designed around a particular view of cognition. This view specifies that cognition:

- is goal-oriented
- reflects a rich, complex, detailed environment
- requires a large amount of knowledge
- requires the use of symbols and abstractions
- is flexible, and a function of the environment
- requires learning from the environment and experience.

(This list and the subheadings below adapted from Lehman et al., 1996). The view of cognition is important as it affects the structures and mechanisms which underlie Soar. These are outlined below.

3.2.1. Goal Context

The Goal Context structure is central to Soar. Soar deals with four kinds of conceptual object: goals (why something is being done), problem spaces (a method of partitioning knowledge), states (internal representation of the situation) and operators (maps from state to state), and a goal context encapsulates a particular instance of these.

For example, consider a model of a student working on a dissertation. In this case, the model could contain a goal context with objects that were something like the following:

- *Goal:* produce an optimal dissertation report
- *Problem space:* the set of knowledge related to report creation
- *States:* report title, deadline, supervisor, etc

- *Operators:* schedule meeting with supervisor, drink coffee, read paper, etc

The above may seem very general. However, Soar is hierarchical (see *Impasses* below) and so the goal context outlined above would have several sub-contexts which would be more specific.

3.2.2. Working Memory

Working Memory [WM] contains the current situation (which may include past states and hypothetical states as required for reasoning), in the form of one or more goal contexts. The results of perception are also held here.

3.2.3. Long-Term Memory

The knowledge in Long Term Memory [LTM] is processed by the architecture to produce behaviour. The knowledge is stored in the form of associations which map the current goal context (in WM) to a new goal context.

With the student example above, some associations might include the following:

(a1) If I perceive that I have finished reading a paper

then suggest a goal to summarise that paper.

(a2) If there is a goal in WM to summarise a paper

then suggest achieving it using the Summary Creation problem space with

an initial state having **num_words** *0* and **amount_summarised** *0*.

(a3) If using the Summary Creation problem space and the **amount_summarised** is < *100*

then ...

Note that the exact format of the associations is unimportant. I have used if-then structures as that is the style adopted in Lehman et al. (1996). More details on how the "then" parts of associations are handled are given in Section 3.2.5 below.

A further point is that "Soar's long-term memory is *impenetrable*. This means that a Soar system cannot examine its own associations directly; its only window into LTM is through the changes to working memory that are the results of associations firing." (*ibid*, p. 34; their emphasis). I believe that this is likely to lead to a more human-like model of cognition but at the expense of ease of programming and debugging.

3.2.4. The Perception / Motor Interface

The Perception / Motor Interface defines mappings from the external world to internal representations in Working Memory, and from internal representations to action in the external world.

3.2.5. The Decision Cycle

The Decision Cycle is the primary process underlying Soar's cognition. The cycle is made up of two phases: elaboration and decision. In elaboration, the architecture attempts to match the contents of WM against associations held in LTM. Any that match fire in parallel. Any changes to the state happen immediately but changes to the context (e.g. changing to a new problem space, adding a new goal context) are simply added to a list of suggestions. Elaboration repeats (as changes to state may mean that new associations can fire) until no further associations match.

At this point the decision phase begins. This phase decides which one (and only one) of the suggested context changes to perform and performs it. The process uses those associations from LTM which suggest which context change is the “best” to enable it to make a decision.

If the decision process cannot decide, then we have an *impasse*:

3.2.6. Impasses

As outlined above, impasses occur when a decision cannot be made about which context change should be applied. Note that since knowledge in Soar is compartmentalised into problem spaces, the solution may be found in another problem space. When an impasse arises, a new sub-goal is created to find the knowledge necessary to make this decision. Impasses therefore provide an opportunity for *learning*. The model would include associations suggesting how impasses could be resolved (by looking in a specific problem space, for example).

When a solution is found, a new association is added to the original problem space in LTM so that next time the situation is encountered an impasse will no longer occur.

Inability to make a decision is just one type of impasse (another may be the lack of information on how to perform a required action, for example). However, all impasses are solved in the same way.

Impasses are what give Soar its hierarchical structure: models are given an initial goal context, and any further sub-goals are generated solely by impasses. For example, our student might have a goal of writing a dissertation which might lead him to determine that he must read papers. However, if the knowledge on how to do this is not available in his report creation problem space then an impasse will be generated and sub-goals generated in the technical reading problem space.

3.2.7. Chunking

Chunking is the name given to the procedure by which new associations are added to LTM. These new associations are created automatically whenever results are generated from an impasse. This involves looking at all contextual information which generated the impasse’s result. The process is explained in more detail in section 8 of Lehman et al. (1996).

The concept of Problem Spaces seems to me to be a useful one, as partitioning knowledge can make search more effective. However, I would be concerned that this could result in a lot of redundancy once chunking has been performed several times, as similar or related associations would be found in different problem spaces.

3.2.8. Summary

In summary, Soar uses goal contexts (goals, problem spaces, states and operators) to store knowledge. Knowledge in Working Memory holds details of the current situation, whilst Long-Term Memory holds associations which map one goal-context to another. Learning comes about as a result of impasses, i.e. situations where the current problem space does not provide a solution to the current problem. Once a solution has been found, chunking is used to add relevant new associations to Long-Term Memory.

3.3. EPIC [Executive Process-Interactive Control]

3.3.1. EPIC in Theory

The EPIC architecture is being developed for the modelling of human cognition and performance (Kieras and Meyer, 1997). A key theoretical aspect of EPIC is that it is uses embodied cognition: i.e. constraints imposed by the perceptual-motor system are considered. In short, "how long it takes an EPIC model to do a task depends intimately on how EPIC's eyes, perceptual mechanisms, and effectors are used in the task" (Kieras and Meyer, 1997; p. 395). This allows EPIC to model human behaviour more effectively and more accurately.

Another important aspect of EPIC is that of multiple-task performance. It is known that the capacity a human has for information-processing is limited, and this has been traditionally expressed by a single-channel bottleneck. Despite this, however, it appears that humans *are* able to perform multiple tasks simultaneously. EPIC takes a fairly radical approach, described by Kieras and Meyer (1997; p. 397) as follows:

> With EPIC ... we do not make the assumption that central capacity for cognitive processing is limited. Such an assumption is traditional but lacks both empirical and metatheoretical justification. In contrast, we assume that limitations on human ability are all structural; that is, performance of tasks may be limited by constraints on peripheral-perceptual and motor mechanisms or by limited verbal working memory capacity, rather than by a pervasive limit on cognitive-processing capacity.

3.3.2. EPIC in Practice

The EPIC architecture includes perceptual processors (visual and auditory), cognitive processors and motor processors. The details of perception and motor processing are not relevant to this project and so these processors will not be examined here. Cognitive processing, by contrast, is of interest:

Cognitive Processing

EPIC's cognitive processor is programmed via production rules which use the Parsimonious Production System [PPS] interpreter (Bovair et al., 1990). The rules have the following format: `(<rule-name> IF <condition> THEN <actions>)`. Rules are limited in what they can do: "[conditions] can test only the contents of the production-system working memory. The rule actions can add and remove items from working memory or send a command to a motor processor" (Kieras and Meyer, 1997; p. 402). Working memory is explained below.

EPIC's cognitive processing operates in cycles. At the start of a cycle, working memory is updated with the output from perceptual processors and with the modifications from the previous cycle; at the end of a cycle, the production-system working memory is updated and any required commands are sent to the motor processors.

As alluded to above, EPIC allows more than one production rule to fire at a time, so-called "Cognitive Parallelism" (*ibid*; p. 403). In fact, *all rules* whose conditions match the contents of working memory are fired on each cycle, and thus all their actions are executed.

The current version of EPIC does not offer support for learning.

Working Memory

Although PPS has only one working memory structure, EPIC treats it as several partitions. The first four of these relate to particular processors: visual memory, auditory memory and tactile memory contain information from perceptual processors, whilst motor memory contains "information about the current state of the motor processors, such as whether a hand movement is in progress" (*ibid*, p. 404).

Another form of working memory is the control store. The control store contains items to represent the current goals and the steps for accomplishing these goals. As this shows, control information in PPS is a type of working memory item like anything else which the model "knows" (and can therefore be manipulated by rule actions). This greatly assists cognitive parallelism, as it allows running tasks to control other tasks by manipulating their goals and actions.

The final form of working memory is simply referred to as General WM and is used to store miscellaneous information for and about tasks.

3.3.3. Production Rule example

The following example is part of a model of the selection of items from a drop-down list (pull-down menu):

```
(if-not-target-then-saccade-one-item
if
(goal do menu task)
      (step visual-search)
      (wm current-item is ?object)
      (visual ?object is-above ?next-object)
      (not (visual ?object is-above nothing))
      (motor ocular processor free)
      (visual ?object label ?nt)
      (not (wm target-text is ?nt)))
then
((deldb (wm current-item is ?object))
      (adddb (wm current-item is ?next-object))
      ((send-to-motor ocular move ?next-object)))
```

Figure 3.1 (adapted slightly from Kieras and Meyer, 1997, p. 414)

This rule causes the model to move its eye down the list (`(send-to-motor ocular move ?next-object)`) until the object currently being looked at (stored in the variable `?object`) is that with label `?nt` or the end of the list is reached (`(not (visual ?object is-above nothing))`). As well as moving the eye, the `then` actions update working memory to show that the current object is now the next object (the `deldb` and `adddb` instructions).

3.3.4. Summary

In summary, EPIC is designed to model human cognition. Unlike many other models, it does not assume a cognitive bottleneck. Rather, performance is limited by the size of working memory, and the fact that only one task may use a given motor (e.g. eyes) at a time. Cognitive processing is done via Production Rules which use the PPS interpreter and test against working memory. Rules' actions may modify the memory and send commands to the motors.

I find the EPIC architecture very interesting in that it does not have this in-built limit on cognitive performance. Similarly, the idea of cognitive parallelism is an interesting one, although I imagine that it could lead to problems when a number of incompatible actions attempt to control the motors simultaneously – interleaving these actions could lead to very strange movement sequences, for example. Attempting to solve this problem by allowing tasks to manipulate the goals and actions of other tasks could lead to interaction problems similar to those encountered with the Subsumption Architecture (see section 6.8.1).

3.4. ACT-R

Like EPIC, ACT-R is "a cognitive architecture, a theory for simulating and understanding human cognition" (ACT-R Research Group, 2004a). In fact, ACT-R's modules which interact with the environment are adapted from those of EPIC. However, as will become clear, there are differences between the two.

The ACT-R architecture has two types of memory: declarative (facts) and procedural (rules). It is goal-driven, goals being placed in a specific goal buffer. All other modules in the system also have buffers, this being the only way that the production system (see below) can interact with them. The system's modules are shown in the overview diagram (Figure 3.2).

3.4.1. Declarative Memory and Pattern Matching

Items in declarative memory are called chunks. Chunks have a name, a category (specified by an "isa" slot) and any number of additional slots containing further information. For example, the following chunk holds the fact 3 + 4 = 7. This example is taken from Anderson et al. (1997; p. 441).

```
fact3+4
      isa         addition-fact
      addend1     three
      addened2    four
      sum         seven
```

ACT-R attempts to simulate the working of the human mind in its knowledge-recall process. This is done by pattern-matching of facts and the idea of *activation*.

Items in declarative memory ("chunks") are retrieved from declarative memory via pattern matching. For example, to help decide what to do next a model may request an instance from declarative memory (using the retrieval buffer) which describes a situation matching the one it currently finds itself in. If no exact match is found, a partial match may be returned (see below).

Which instance is returned depends on activation, how readily available the fact is in memory. Chunks that have been used recently or chunks that are used very often end up with a high activation. Activation decays over time as the chunk is not used. Activation models the human recall process whereby knowledge used regularly (e.g. using a kettle) or recently (the previous key I have pressed as I type this) is easily accessed but other knowledge (e.g. a discussion from several weeks ago) is hard or even impossible to retrieve.

The activation of a chunk depends on its usefulness in the past (base-level activation) and its relevance to the current context (associative activation). The equation for this, and further explanation, is given in Taatgen et al. (in press). Noise is added to the activation, meaning that alternative strategies may be tried at times, increasing the overall quality of solutions chosen and providing further opportunities for learning.

I believe that the use of noise could produce more human-seeming behaviour, as humans do not always solve problems in the same way, even when a known effective solution exists. It could be argued that an agent which was specified in enough detail would do this anyway, as the complexity of the environment would result in new emergent behaviours. However, I do not believe that this could be relied on completely, especially with relatively simple environments or agents.

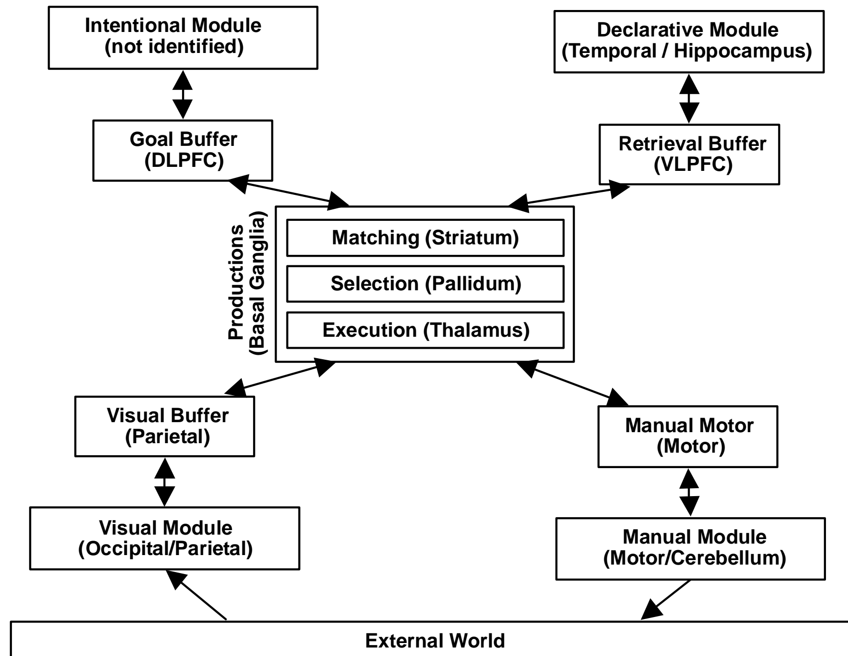

Figure 3.2: An overview of ACT-R 5.0 (adapted from Taatgen et al., in press).

Partial Matching

If partial matching is required, then this is made possible by decreasing the activations of chunks which do not match. The greater the mismatch, the higher the penalty and thus the lower the likelihood that the chunk will be returned. Enabling partial matching means that, whilst it is still the chunk with highest activation which is returned, the chunk's slot values need not match those requested exactly. Taatgen et al. (in press) give an example of the use of partial matching with the *Sugar Factory* example. In this, the model was able to improve performance by analysing past settings, actions and their effects (even if the previous settings did not match the current situation exactly), without needing to know the rules which governed success.

Partial Matching appears a very useful procedure for learning which I believe could go some way towards removing the need to completely specify every situation that an agent might encounter. However, it does seem a little haphazard (see the discussion of learning past-tenses in section 3.4.3).

3.4.2. Procedural Memory

Rules in procedural memory (known as *productions*) drive behaviour in ACT-R by specifying what should be done and when (i.e. in response to what conditions) it should be done.

The general form of a production rule is as follows (from ACT-R Research Group, 2004b, Unit 1):

```
(p Name
list of buffer tests
==>
list of buffer changes
)
```

The following example shows the rule to count to a particular number. An English description is on the right. (Example and description from *ibid*):

```
(P increment
      =goal>
            ISA
            count-from
            start =num1
            - end =num1
            step  count
            ing
      =retrieval>
            ISA
            count-order
            first =num1
            second=
            num2
==>
      =goal>
            start =
            num2
      +retrieval>
            ISA
            count-order
            first =
            num2
      !output! (=num1)
)
```

If the goal is
to count from
the number =num1
and =num1 is not the final digit
and one is counting
and a chunk has been retrieved
of type count-order
where the first number is =num1
and it is followed by =num2

Then
change the goal
to continue counting from =num2
and request a retrieval
of a count-order fact
for the number that follows =num2
and output =num1 in the trace

Like chunks, productions are also chosen by pattern-matching. (I.e. attempting to find rules whose conditions match the contents of buffers.) When more than one production rule is applicable the *utility* attribute is used.

Conceptually, utility is how useful a production is in the current context. ACT-R automatically keeps track of the estimated cost and estimated probability of success for each rule, and the utility of a rule *i* is calculated by the following equation (from Taatgen et al., in press; p. 13):

$$U_i = P_iG - C_i$$

P_i is the probability of success, G is the value assigned to the current goal and C_i is the estimated cost. The actual production choice equation (which determines the probability of choosing production i from a matched set of n productions) is given in *ibid* and, as with pattern matching for chunks, includes a noise element.

Probability is calculated as the ratio of successes to attempts (i.e. the sum of successes and failures). Cost is similar: “the sum of the efforts invested in a goal ... divided by the total number of experiences (both successes and failures)” (*ibid*, p. 14). Equations for these elements and a discussion of their initial values can be found in p. 14 of *ibid*.

The concept of utility for action selection is found in other architectures. For example, the idea of a option’s “score” in Stone and McAllester (2001).

3.4.3. Learning in ACT-R

As shown above, the values of a production’s Probability and Cost alter as successes and failures are experienced. This *Utility Learning* is one type of learning in ACT-R.

Learning is also accomplished via production compilation, described neatly in Taatgen et al. (in press; p. 23) as follows:

> Production compilation ... learns new rules by combining two existing rules that fire in sequence into one new rule. If the first of the two rules makes a request to declarative memory the result of which is used by the second rule, then the retrieved chunk is submitted into the new rule, effectively eliminating the retrieval.

Although it may not be obvious at first reading, this procedure enables genuinely useful learning and not just the creation of ever-more specific and efficient rules. This is often shown by the example of learning the past tense of verbs (described briefly in *ibid* and in detail in Taatgen and Anderson, 2002).

In this example, the past tenses of verbs are originally generated by two rules: One of these searches for a chunk stating the past tense of a particular verb, whilst the other (initiated if the direct match fails) attempts to match by analogy, by finding a pattern in the existing facts about other verbs. Discovering that many verbs append an -ed suffix, new rules will be composed for the verbs with previously unknown past tenses[1]. Irregular verbs will have to be learned by example as this analogy method does not work for them.

The production compilation may appear to be a little over-zealous for this sort of example, in that all sorts of incorrect past-tense rules and chunks may be constructed (e.g. swim → swimmed). However, the use of utility and activation in pattern matching (see above) should mean that incorrect examples will gradually be ignored more and more often as they fail to achieve the goal (i.e. a correct sentence) and thus decrease in utility, whilst the incorrect chunks (i.e. those facts which describe the past tense of a verb incorrectly) will be retrieved less and less frequently as their activation will decrease. However, Taatgen and Anderson (2002, p. 138) do admit that “there is no mechanism to really safeguard this”.

[1] Examples of verbs with this pattern will be returned more frequently than those such "hit" which do not change in the past. The reason for this is that -ed verbs are more common and so more examples of them will be available in declarative memory.

3.5. Summary

This section has described Soar, EPIC and ACT-R. The following table illustrates some of the key points of comparison.

	BOD	Soar	EPIC	ACT-R
Decomposition by activity rather than function?	✔	✗	✗	✗
Hierarchical?	✔	✔	✗	✗
Associations / Production-Rules?	✔	✔	✔	✔
How are multiple possible actions handled?	priorities	decision-phase, i.e. further associations	cognitive parallelism	utility
Parallel Actions?	✔	✗	✔	✗
Learning?	Updating of state, (dynamic plan-writing)	chunking	✗	probability & cost, production compilation
Partial Matching & Noise?	✗	✗	✗	✔

Table 3.1 Summary of Architectures

4. The Bot in Action

4.1. Introduction

This chapter presents a number of scenarios demonstrating the actions of the bot that I created (the "bodbot") and relates these back to the plan files created. The chapter's purpose is threefold:

- To demonstrate the development of the plan files.
- To illustrate how the actions of the bot are guided by the plan file it uses.
- To give examples of the bodbot's actions, and thus provide a more concrete starting-point for the discussion of the development process which follows in chapter 5.

The plan files control the execution of sensory and action primitives; these primitives are discussed as part of the explanation of the development process in chapter 5.

The actions of the bot are illustrated by a series of commentary-style descriptions (recognizable both by their indentation and by the style of writing they use) which are interleaved with brief analysis and samples of plan code. The complete texts of the plan files demonstrated in this chapter are given in Appendix A.

To improve the examples, some plans are illustrated by the actions of a bot on the red team and others by the actions of a bot on the blue team. This provides more of a difference as might be imagined as, for reasons which are not clear, the blue bot always began the game facing a certain direction regardless of map settings. This means that many of the early plans do not work very effectively for a blue bot as the bot never sees any navigation points at all (they're all behind her).

Which bot is used in the scenarios below can be determined from whether male or female pronouns are used: the red player is represented in Gamebots by a male wizard, whilst the blue player is represented by a female character.

4.2. The Scenarios

4.2.1. Walking To Navigation Points

The initial plan was based upon `poshbotfollow.lap` the plan created by Kwong's (2003) for his "poshbot". This original plan had the bot wandering around and following any players he saw.

The first plan I created removed the player-following element, replacing it with one which attempted to follow navpoints (navigation-points, aka pathnodes):

> Yes, the bodbot has just this moment spawned into the play-area. He's wasting no time running off that ledge and towards the tunnel, seems to be having a bit of trouble on the corners, though: he's paying more attention to that wall than it really deserves ... no, here he goes off again. Looks like he's missed that vital turning though, seems more interested in the walls of the tunnel again, no wait, he's coming back, takes the turning, now he's looking around again, trying to decide where to go. He's finally decided and now he emerges from the tunnel.

The important part of this plan is the competence below. The top-level Drive Collection only contains two drives and thus almost always fires this competence as the other is only triggered when the bot walks into something:

```
(C get-to-enemy-base (minutes 10) (goal ((at-enemy-base)))
  (elements
    (
      (find-base (trigger((reachable-nav-point)))
          walk-to-nav-point)
      (wander-base (trigger((succeed))) wander-around)
    )
  )
)
```

When the bot starts up, he can see a navigation point specified as reachable and so he runs off the ledge (only a short drop) to get to it. This is because the `reachable-nav-point` trigger returns true. This sense (described in more detail in section 5.3.3) also sets a variable to contain the location of the navpoint which the `walk-to-nav-point` action then uses when communicating with Gamebots.

On the occasion of his trouble in the tunnel, the problem is that because of the curve of the tunnel he can no longer see any navpoints. For this reason the `wander-base` element takes over (a trigger of `succeed` means that it always fires if no higher-priority element can). `wander-base` triggers the following competence:

```
(C wander-around (minutes 10) (goal((see-player)))
  (elements
    ((stuck (trigger ((is-stuck))) avoid))
    ((pickup (trigger ((see-item))) pickup-item))
    ((walk-around (trigger ((is-rotating False))) walk))
  )
)
```

This version of the competence is taken directly from Kwong's `poshbotfollow.lap`. The bot walks towards the wall (`walk-around`, the lowest-priority element, fires) and, when he hits it, the `stuck` element is triggered. `avoid` is the following Action Pattern:

```
(AP avoid (minutes 10) (stop-bot rotate then-walk))
```

This causes the bot to rotate and attempt to walk again. Eventually he sees another navigation point and continues as described. The reason he doubles back after missing the turning is again that he hits a wall, and his rotation is such as to make him face back the way he came.

`get-to-enemy-base` and `wander-around` provide an example of POSH's Slip-Stack (see section 2.4.5). `get-to-enemy-base` is fired by a Drive Collection element, `to-enemy-base`. Since `get-to-enemy-base` then triggers a further competence (`wander-around`), `to-enemy-base`'s active element α is set to point directly to the root of `wander-around` rather than to the root of `get-to-enemy-base`, thus missing out a link in the hierarchy. If `wander-around` triggers the `avoid` action pattern, then α will point to that instead, missing out another link in the hierarchy.

> Out of the tunnel, the bot's now running towards the blue flag ... yes, he's got it! What a performance! Seems in no hurry to be going anywhere now though. In fact, he's wandering around. I've never seen anything like it... Is he going to just throw that early lead away?

Finally, the bot grabs the blue flag. This is at this stage a mere side-effect of his navpoint-following behaviour (the points lead directly there). Once he is there he can see no further navigation points and so resumes the `wander-around` behaviour.

A point worth noting is that this plan is not often as effective as in the example above – the bot usually spends large amounts of time wandering around the same few navpoints, or getting stuck in rooms. The example above was chosen as it is more interesting.

4.2.2. A Greater Awareness of Flags

The plan used for this scenario is a quite major extension of the previous one, as the following commentary shows. For brevity, I have not included a commentary on the entire run, just the points of interest. Note that the red player mentioned is the one controlled by me.

> And here comes the blue bodbot now. She's looking around, wondering where to go next. And now she's off, running towards the tunnel...

The "looking around" at the beginning comes from a modification to the `get-to-enemy-base` competence, whose elements are now the following (the second is new):

```
(find-base (trigger((reachable-nav-point)))
    walk-to-nav-point)
(find-nav-point (trigger((succeed))) rotate 10)
(wander-base (trigger((succeed))) wander-around)
```

Although two of the elements have triggers of `succeed` (the function which always returns true), the retries limit (`10`) on `find-nav-point` means that the lowest-priority element does sometimes get a chance to fire. In the example given above, however, the rotating leads to a position where the bot can see a reachable navpoint, and thus the first element fires. A trigger of `succeed` is used rather than just `True` as this was the style used in `poshbotfollow.lap`.

The running through the tunnels, omitted from the commentary, is very similar to the behaviour described in section 4.2.1.

> The bodbot emerges from the tunnel, she's almost at the enemy base now, the prize in her sights. Yes, I think she's going to make it! She makes a clear run for the red flag and grabs it! Nice work there, but can she capitalise on this early success? Remember, she's still got to take it home.

To understand the bot's next actions (running to the enemy flag), we need to consider the top-level Drive Collection:

```
(RDC life (goal ((fail)))
  (drives
    ((pickup-our-flag-from-ground
        (trigger ((our-flag-on-ground))) go-to-own-flag))
    ((pickup-enemy-flag-from-ground
        (trigger ((enemy-flag-on-ground)))
            go-to-enemy-flag))
    ((attack-enemy-with-our-flag
        (trigger ((see-enemy-with-our-flag)))
            attack-enemy-carrying-our-flag))
    ((take-enemy-flag-from-base
        (trigger((enemy-flag-reachable)
```

```
            (have-enemy-flag False)))
               go-to-enemy-flag))
        ((hit (trigger((hit-object)(is-rotating False)))
            avoid))
        ((go-home (trigger((have-enemy-flag))) go-to-own-base))
        ((to-enemy-base (trigger((succeed)))
            get-to-enemy-base))
      )
    )
```

Until now, the main element which has been firing is `to-enemy-base`, which has resulted in the competence discussed above being used. Once the bot approaches the enemy flag, however, the trigger `enemy-flag-reachable` returns true and `go-to-enemy-flag` is fired instead (as it has a higher priority, and `have-enemy-flag` already returns `False` as required.). This is a single-element Action Pattern which fires the `to-enemy-flag` primitive. This primitive simply sends a `RUNTO` message to Gamebots, giving it the ID of the flag and causing the bot to run straight there.

> Wait a minute, John, there seems to be some sort of upset at the other end of the arena! Yes, the bodbot's quest for glory has left her own flag dangerously unguarded and the red player has stolen it! He looks pretty pleased about that one, and is heading for home. He passes the bodbot in the tunnel but she ignores him! What *is* she thinking?

To demonstrate a situation more similar to genuine Capture the Flag games, I intervened at this point and, playing as the red player, stole the blue flag. The reasons for the bodbot's ignoring of the red player are unclear. I believe that it is a limitation of the bot's sensory abilities as the relevant plan elements do fire at a later occasion (see below). It could be connected to the UT skill-level of the bot (set to "novice" during this scenario), as these do have an effect on all bots, not just those that come pre-written with UT. Brief preliminary tests suggest that a higher skill level does improve perception by a small amount, but not by as much as would be expected. I have not had time for detailed tests, however, and it remains an area for future work (section 9.2).

It is also worth noting that, unlike the previous scenario, the bodbot is able to return home once she has captured the flag. The reason for this is the addition of the `go-home` element, whose `have-enemy-flag` trigger now returns true, meaning that `go-home` now executes instead of `to-enemy-base` as before. As the bot recorded the location of her own base when she saw her own flag at the start of the game (see the discussion of the `PositionsInfo` class in section 5.3.2), she was able to send this information to the UT server and retrieve a list of navigation points to use to get home (the `GETPATH` command). Section 5.3.2 discusses some of the functions used as part of this.

> Well Clive, the red player seems to have a pretty unorthodox style himself – he's running back to the blue team's base. Is he just wanting to taunt his opponent with his advantage? Has he forgotten that the blue player has his own flag herself? Well, he won't forget it much longer as she's emerging from the tunnel now, this confrontation could spell trouble!
>
> Too right, John, the bodbot rounds on the red player, running towards him and shooting and ... it's a success! He's been tagged, and he drops the blue flag to the ground where the bodbot grabs it, restoring it to its rightful place! Yes, nothing

> can stop her now! She's running back to her own flag, she's made it now, the blue team scores!!

In an attempt to get the blue player to notice me (and since I cannot win whilst the other team has my own flag), I returned to the blue player's base. On this occasion the bodbot did notice that I had her team's flag. Doing so meant that her current action of going home was interrupted as the `attack-enemy-with-our-flag` Drive Collection element fired instead (it has a higher priority) and the bot began to attack me.

Upon being tagged (killed), the red player drops the blue flag he has been carrying and the bodbot's current undertaking is again interrupted, as the `pickup-our-flag-from-ground` element now fires (it has an even higher priority). Picking up one's own flag returns it instantly to the base, and the bodbot scores when returning to her own flag while carrying the red one. The bot only moves towards her own flag as the list of navpoints leads there: at this stage there is no specific drive to run directly there once it is reachable.

> Well, that certainly was impressive. The bodbot seems to have had enough though, she's not going anywhere! This is remarkable, she's just standing there! What it she thinking?!

This final segment illustrates a problem discussed in section 5.3.4: the expiry of out-dated

Figure 4.1: the blue bot runs towards the goowand on the floor.

state the bot holds. In this case, the instance of the `PositionsInfo` class held out-dated information about the enemy flag, claiming that it was reachable from the bot's current location (as that had the been the case until the bot scored and the red flag was returned to

the red base). This out-dated information resulted in both of the following element's triggers succeeding

```
((take-enemy-flag-from-base
    (trigger((enemy-flag-reachable)
     (have-enemy-flag False)))
        go-to-enemy-flag))
```

The bot therefore attempted to send a command to Gametbots to make it run directly to the enemy flag. This was not possible from its current location, and so nothing happened. Making the agent more robust to handle this sort of failure is discussed briefly in section 6.8.2.

4.2.3. Responding to Attack

This final scenario introduces a number of new elements, the most important being the bot's ability to respond when it is attacked. As before, only part of the run is described.

> For those of you who've just joined us, we're seeing a fine run by the blue bodbot. She's part way through the first stage of the tunnels but, uh oh, it looks like she's going to miss that turning. No, it's okay, she's spotted it, and is running to pick up that goowand. Her supporters will be breathing a sigh of relief that she's no longer unarmed.

The bot begins, as in the previous scenarios, by following navigation points as part of the `get-yourself-to-enemy-base` Drive Collection element. This is interrupted by the following higher-priority element:

```
((pickup-weapon-as-unarmed
    (trigger  ((see-reachable-weapon) (are-armed False)))
        get-weapon))
```

As described, this results in the bot running to pick up the goowand (one of the weapons the Gamebots interface adds to the game). This is an interesting example of a case where an outside observer may attribute different intentions to the actions of an agent than those which are actually underlying the agent's actions (Sengers, 1998): The only reason that the agent made the turning in this case was that she ran to the goowand which then resulted in her seeing new navpoints. Had the wand not been there, it might have taken her longer to find the turning. Figure 4.1 shows the blue bot running towards the goowand.

> Continuing through the tunnels she makes a dash for the red flag and takes it! Where are the defence? Well, someone's trying to shoot her but not doing a very good job of it, that shot landed just in front of her. Fortunately for the red team, that goo will stay there for a while before it explodes. Looks like the bodbot's a little confused though – looking around for where to go next... yes, she's off now, and ouch! That goo-explosion's got to hurt.

The assailant was a bot controlled by me. The goowand fires blobs of goo which stick to walls and floors and remain there for a few seconds before exploding. The bodbot's confusion described here was due to the bot receiving a corrupt Gamebots message of the form described in section 5.3.4.

> Not one to let that sort of behaviour go unnoticed, she's looking around for the assailant, she's spotted him now and begins to shoot ... ooh, right in the stomach! Keen not to throw that lead away though, she's now heading back to her own base. Obviously doesn't want another surprise attack, she's keeping firmly focussed on

that attacker as she runs back through the tunnels, doesn't seem to be shooting him, though!

The response to attack comes as a result of the following Drive Collection element:

```
((respond-to-attack-health-not-low
    (trigger ((taken-damage) (armed-and-ammo)
     (is-responding-to-attack False)))
        respond-to-attack))
```

This element has a higher priority than `go-home`, the drive element previously being attended to, and so the following competence is triggered:

```
(C respond-to-attack (seconds 10) (goal ((fail)))
  (elements
    (
      (attack-visible-attacker
      (trigger ((taken-damage-from-specific-player)))
          respond-to-visible-attacker)
      (find-attacker (trigger ((succeed)))
          try-to-find-attacker)
    )
  )
)
```

In some cases, the bot will receive details of the assailant when receiving a message from Gamebots about damage inflicted. For example, if the bot actually sees the shot being fired. This was not the case in this example, however, and so `find-attacker` is triggered. This in turn triggers the following competence:

```
(C try-to-find-attacker (seconds 3) (goal ((fail)))
  (elements
    (
      (found-attacker (trigger ((see-enemy)))
          respond-to-visible-attacker)
      (spin (trigger ((succeed))) big-rotate 1)
    )
  )
)
```

This competence is the reason the bot looks around for the attacker: the `spin` element causes the bot to perform the `big-rotate` action. Note the limit on retries here: the bot shouldn't keep on turning around as it may never be able to see the attacker. In this case the search was successful, leading to the `see-enemy` sense returning true and the `found-attacker` element running. It is this element which makes the bot shoot the attacker.

Furthermore, finding an attacker results in variables being set telling the bot to keep looking at the attacker whilst performing other actions (`KeepFocusOnID` in `CombatInfoClass`, see section 5.3.2). In practice, this means that when running, the bot instead sends a command to *strafe*. Strafing is running in one direction while facing another.

Such strafing makes it possible for the bot to continue shooting the assailant. This happens successfully much of the time but was not the case in the example above. The reason for this is that the behaviour relies on the fact that bots continue shooting until either explicitly told to stop or until their target is no longer visible. In this case the bot briefly lost sight of her assailant and so stopped shooting prematurely. Overcoming this sort of problem is an area for future work (see section 9.2).

> Into the home strait now, she turns around for the final sprint, she's nearly there, yes ... she scores! Now she's going back to try another capture, it could be a high-scoring game, folks!

The Drive Collection used for this scenario contains some unexciting but nevertheless very important elements: those which expire state:

```
((expire-our-damage-info (trigger ((succeed)))
    expire-the-damage-info (seconds 10)))
((expire-our-reachable-info (trigger ((succeed)))
    expire-the-reachable-info (seconds 20)))
((expire-our-focus-info (trigger ((succeed)))
    expire-the-focus-info (seconds 30)))
```

The need to expire state is discussed in section 5.3.4, and the previous scenarios give some examples of the problems brought about by not doing so. In this scenario, the reason the bot stopped facing her assailant was because the `expire-our-focus-info` element fired.

The three elements given here are the highest priority in their Drive Collection. However, their limits on frequency mean that other elements get plenty of chance to run.

The bot's attempt to capture the flag again will be aided by the fact that she now knows the location of the enemy base (this information is stored in an object of the `PositionsInfo` class). This means that she can obtain a list of navpoints from the server which give her the path she needs to follow. Note another advantage of the expiry of data: unlike in the previous scenario, the bot no longer believes the enemy flag to be reachable once it has been returned to the enemy base (`expire-our-reachable-info` handles this).

4.3. Summary

This chapter has demonstrated a number of plan files of increasing complexity, highlighting both the connection between plan files and actions, and various problems encountered during development. How the actions and senses triggered by the plan file actually work is explored as part of the next chapter.

5. The Development Process

5.1. Introduction

This chapter has a dual purpose: First, it provides a demonstration of some of the development work I undertook as part of this project (sections 5.3.2 and 5.3.3, primarily). Secondly, it evaluates BOD's contribution to the process and the effectiveness of those principles and practises BOD introduces.

There are two main sections to this chapter: section 5.2 describes the Initial Behaviour Decomposition (i.e. preparation for development), whilst section 5.3 describes the main iterative cycle of development itself.

5.2. Initial Behaviour Decomposition

5.2.1. The Process

As with other aspects of the design process, BOD provides guidelines to follow for initial behaviour decomposition (Bryson, 2001; p.133):

1. Specify at a high level what the agent is intended to do.
2. Describe likely activities in terms of sequences of actions. These sequences are the basis of the initial reactive plans.
3. Identify an initial list of sensory and action primitives from the previous list of actions.
4. Identify the state necessary to enable the described primitives and drives. Cluster related state elements and their primitives into specifications for behaviors [sic]. This is the basis of the behavior library.
5. Identify and prioritize goals or drives that the agent may need to attend to. This describes the initial roots for the POSH action selection hierarchy [explained below].
6. Select a first behavior to implement.

An important point is that BOD's emphasis on revision of specifications in the development cycle (see below) means that it is not assumed that the initial decomposition is perfect. The developer is free to modify the decomposition if issues arise, and the lack of coupling between components means that this can be easily achieved.

5.2.2. My Experience of the Process

Revision of the Specification

The full initial decomposition I produced is given in Appendix B. In my experience, I found that revision of this specification document only happened three times during development despite the fact that the project did depart from the specification laid down by the initial decomposition. The *concept* of revision of specification was one I found very useful and made use of and the initial decomposition provided a very useful starting point. However, the specific action of rewriting the initial document was not one I found helpful.

The main reason for this is that it felt like unnecessary redundancy to have, for example, information about sensory and action primitives both in terms of methods in the behaviour

modules and as items in a list in the specification document. This point is particularly relevant in light of BOD's encouragement of self-documenting code.

Furthermore, moving from coding to something at a higher level of abstraction (specification writing) interrupted the natural flow of development.

Other Points

The list of goals and drives proved to be very helpful as a high-level guide to what needed to be developed. I concentrated on one drive per iteration of the development cycle (see below).

5.3. Evolutionary Design and Development

Creating the agent was a major undertaking, involving learning the Python language, the syntax for POSH plans and how best to use the Gamebots API. This section explains and evaluates BOD's development process, and explores the actual development work I performed.

5.3.1. The Process

The core of the BOD process is the following iterative sequence (Bryson, 2001, p. 120):

1. Select a part of the specification to implement next.
2. Extend the agent with that implementation:
   - code behaviors [sic] and reactive plans, and
   - test and debug that code.
3. Revise the current specification.

The iterative nature of the process allows for *rapid prototyping*: different approaches can be tried, rolling back to the previous iteration if some development is found to be ineffective.

5.3.2. Behaviour Modules, State and Utility Functions

This section explains some of the specific development work that my project involved, outlining behaviour modules, classes used for agent-state and utility functions. Action and sensory primitives are discussed in the next section.

A discussion of the development of the plan file is given as part of chapter 4.

Behaviour Modules

A discussed in the Initial Decomposition (Appendix B), the bot's behaviour is split into three modules:

- Movement – containing state to do with positions of objects, bases and the bot himself.
- Status – containing state regarding health level, weapons held and so on.
- Combat – state about who is attacking the bot, what enemies are around and what teammates are around.

Each of these is stored as a separate Python class, and contains methods for action and sensory primitives. These primitives are registered with the agent (class `Agent` in

posh_agent.py[2]) via the init_acts and init_senses functions, which allows the plan interpreter to call them. Allowing primitives to reside in multiple files is one change I made to the original PyPOSH (see section 8.1 for a discussion).

I made much use of code from the "poshbot", an Unreal Tournament agent designed by Kwong (2003) as part of the development of PyPOSH. This included classes to store agent state and classes for communicating with Gamebots. The AndyBehaviour behaviour module holds those primitives developed for the poshbot. Although this meant that the behaviour decomposition was not as logical as it could be (many of these primitives would be logically suited to the movement module instead), I felt that such a distinction between the simple behaviour of the original bot and the more advanced behaviour of the bot I developed was useful.

Classes for Agent State

Apart from the general agent information provided specifically by Gamebots, and stored in Kwong's Bot_Agent class, I developed two further classes specifically to hold agent state:

- CombatInfoClass – this holds state relating to combat (for example, details of the player holding the bot's flag), and is used by both the movement and combat behaviours.
- PositionsInfo – this class holds state relating to the position of the bot and position of the game objects (e.g. flags and navigation-points), and is used by all three behaviour modules I developed.

The theoretical considerations concerning agent state are discussed in section 5.3.5 below.

Functions for Updating State

The Bot_Agent class (in bodbot.py) coordinates communication with Gamebots and handles some of the agent state. As part of this, it includes a number of functions to pass updating information to the behaviours (e.g. pass_flag_details). When the relevant information arrives from Gamebots, these functions are triggered and call the relevant "receiving" functions in the behaviour modules (e.g. receive_dam_details in CombatBehaviour). These receiving functions update state as required to reflect the new information.

See section 5.3.5 for a discussion of problems with the current method of handling these updates.

Utility Functions

The file utilityfns.py is not a class, simply a collection of functions. These help reduce redundancy by providing useful functionality used in a number of places.

Four examples worth particular note are is_previous_message, send_if_not_prev, nav_point_dict_to_ordered_list and compare_number_strings, as they provide an insight into some of the issues encountered when developing a UT agent, especially one which uses Kwong's (2003) classes to interact with Gamebots.

---

[2] This file is part of Kwong's (2003) PyPOSH implementation, as are posh_core.py and pyposh.py (referred to elsewhere in this document).

is_previous_message and send_if_not_prev

Kwong's `Agent` class, mentioned above, keeps a log of all messages it sends to Gamebots. There are times when this log needs checking to ensure that the same message is not sent twice in a row. An example of when this might occur is when the bot is running from navpoint (navigation-point, aka pathnode) to navpoint in an attempt to find the enemy base. Consider the following competence elements (from `bodbotattack.lap`):

```
(run-to-base (trigger((know-enemy-base-pos)))
    to-enemy-base)

(find-base (trigger((reachable-nav-point)))
    walk-to-nav-point)

(find-nav-point (trigger((succeed))) rotate 10)

(wander-base (trigger((succeed))) wander-around)
```

If the bot can see a reachable navigation-point which s/he is not already very close to (`reachable-nav-point` sense), s/he will attempt to run there (`walk-to-nav-point`). This involves sending a message to Gamebots instructing the bot to run to the given point. Obviously this movement takes some time, and during that time it is likely that the competence will run again. In that event, we do not need to send another message instructing the bot to run to the navpoint (in fact, the Gamebots API advises against sending the same message repeatedly), we simply need wait for the bot to arrive there, at which time the distance-tolerance check will fail (i.e. the bot will be very close to the navpoint) and the bot will attempt to find another point.

`is_previous_message` takes a message and a bot as its arguments, and returns true if the message was sent to Gamebots by the bot as the most recent command. `send_if_not_prev` uses this function but also sends the message to Gamebots if `is_previous_message` returns false. This removes the need for a large number of `if not is_previous_message(...` tests in the primitives themselves.

It is worth noting that since only the most recent message is checked, the instruction to run to the navpoint will be re-sent if the bot is interrupted whilst running there.

nav_point_dict_to_ordered_list and compare_number_strings

One of the advantages of using Kwong's classes to communicate with Gamebots rather than doing so directly is the classes parse attribute strings so that attributes can be sent and received as dictionaries (Python's `dict` type, essentially a hash-table). However, there is one situation where this is less than ideal: lists of navpoints generated by the UT server giving the path to a specified location. These are lists of triples containing a number, an ID and a location. Python dictionaries are unordered, and so a dictionary of these triples must be converted into lists in the correct order. This is accomplished by the function `nav_point_dict_to_ordered_list`. This sorts the list of dictionary keys (i.e. "1", "2" etc) using the `compare_number_strings` function; normal sorting does not work as the keys are strings so "10" would be treated as less than "2" and so on.

5.3.3. The Primitives

This section illustrates part of the development process by presenting some examples of action and sensory primitives. In total, I coded 20 actions and 23 senses, and re-used the 5 actions and 9 senses of the poshbot (Kwong, 2003). The actions and senses shown in this section have been chosen to demonstrate primitives of differing complexity: the first two are

relatively simple, whilst the third is quite advanced. They were also chosen with a view to demonstrating interesting features of the bot, such as its use of state, the trade-offs between plans and behaviours and so on.

`are-armed` sense (from the `status` module):

```
def are_armed(self):
    if self.bot.botinfo == {}:
        return False
    else:
        if self.bot.botinfo["Weapon"] == "None":
            return False
        else:
            return True
```

Note that names with underscores as separators are function names, whilst those separated by hyphens are the names used in the plans. (I.e. `are_armed` is registered with the agent as `are-armed`).

This function defines a sense. The only parameter is `self`, a required parameter for any class method in Python (it is automatically set to refer to the relevant class instance). The function tests against an item in `self.bot.botinfo`. `botinfo` is a dictionary of state about the bot, filled automatically by Kwong's class which communicates with Gamebots. The fact that we are testing against a string (`"None"`) reflects the fact that this dictionary was originally derived from a string-based list of attributes. The return value is either `True` or `False` (Python is case-sensitive). Senses need not be limited to returning true/false values (`ammo_amount` returns an integer, for example), but many of them do.

The following is the definition of the `shoot-enemy-carrying-our-flag` action primitive (from the `combat` module):

```
def shoot_enemy_carrying_our_flag(self):

    if self.CombatInfo.HoldingOurFlag != None and
    self.CombatInfo.HoldingOurFlagPlayerInfo != None:

        Target = self.CombatInfo.HoldingOurFlag

        Location = self.CombatInfo.
          HoldingOurFlagPlayerInfo["Location"]

        self.bot.send_message("SHOOT",
          {"Target" : Target, "Location" :Location})

    return True
```

This primitive uses details from the `CombatInfo` class (discussed in section 5.3.2): `HoldingOurFlag` contains the ID of the flag-carrier (every object in UT has a unique ID), whilst `HoldingOurFlagPlayerInfo` is a dictionary of information about the flag-carrier, including, as shown here, his/her location.

The astute reader may wonder why the checks against `HoldingOurFlag` and `HoldingOurFlagPlayerInfo` occur here rather than in a sensory primitive included in the plan. In fact, the plan does include such a sense (`see-enemy-with-our-flag`). However, including these checks in the action increases the robustness of the code.

As the penultimate line shows, messages are sent to Gamebots using Kwong's `send_message` function. (This is found in the `Bot_Agent` class). As discussed above, this accepts attributes in the form of dictionaries.

Actions must return true if they complete successfully. This is a general requirement of POSH. For example, Action Patterns terminate as soon as one of their elements fails (i.e. returns false). In fact, many of my actions return true even if they do not complete successfully, the reason being that their failure is not significant enough to warrant the termination of an entire sequence.

A Longer Example

The code for the `reachable-nav-point` sense follows. This is intended to provide an example of a more complicated primitive and hopefully to illustrate further features and functions used. For ease of explanation, I have broken it up into sections:

```
# returns True if there's a reachable nav point in
# the bot's list which we're not already at

def reachable_nav_point(self):
```

This is just a comment and a standard class-function definition.

```
    # setup location tuple

    if not self.bot.botinfo.has_key("Location"):

        # if we don't know where we are, treat it as
        # (0,0,0) as that will just mean we go to the
        # nav point even if we're close by

        (SX, SY, SZ) = (0, 0, 0)

    else:

        (SX, SY, SZ) =
utilityfns.location_string_to_tuple(self.bot.botinfo[
"Location"])
```

As part of this sense, we must already determine whether we are already close to the navpoint we are aiming for. Our location is stored in the `botinfo` dictionary (discussed above). However, this is stored as a string and thus must be converted into a tuple (in this case, a triple) for comparison, hence the call to `utilityfns.location_string_to_tuple`. This line also provides an example of Python's ability to perform multiple-assignment.

If the location is not available, we can treat the bot as being at `(0,0,0)`. This might mean that we are actually close to a navpoint but do not realise it, but it is worth taking this minor risk rather than doing nothing.

```
    # is there already a navpoint we're aiming for?

    DistanceTolerance = 30 # how near we must be to
                      be thought of as at the nav point

    if self.PosInfo.ChosenNavPoint != None:

        (NX, NY, NZ) = self.PosInfo.ChosenNavPoint

        if utilityfns.find_distance((NX, NY),
          (SX, SY)) > DistanceTolerance:
```

```
                return True
            else:
                self.PosInfo.VisitedNavPoints.append((NX,
    NY, NZ)) # set this NP as visited
                self.PosInfo.ChosenNavPoint = None
```

It may be that the bot has already chosen a navigation point to aim for (`self.PosInfo.ChosenNavPoint`, see above for a discussion of the `PosInfo` class) and is currently heading there. In this case, we test whether the bot has already got there. This uses another utility function, `find_distance` (inherited from Kwong's poshbot). If the bot is not already there, then we need do nothing more – the bot has a location to head for so we can simply return.

However, if the bot is there then we add the point to our list of visited navpoints and clear the variable stating where we are heading for. We do not return from the function but rather continue execution to find a new navpoint.

This extract of code is an interesting one as it is an example of something which could be accomplished either by a behaviour (as here) or by making the plan file more complicated. I.e. Adding a sense to check whether we are at the place we're heading and an action to clear it if we are. There is no overwhelming advantage to either method, it is more a matter of personal preference. The trade-off this demonstrates (between complexity of plans and complexity of behaviours) is an important one, however.

```
        # now look at the list of
        # navpoints the bot can see
        if self.bot.nav_points == None or
          len(self.bot.nav_points) == 0:
            return False
```

If the bot cannot see any navpoints then the sense obviously fails.

```
        else:
            # nav_points is a list of tuples.  Each tuple
            # contains an ID and a dictionary of
            # attributes as defined in the API
            # Search for reachable nav points
            PossibleNPs = self.get_reachable_nav_points(
              self.bot.nav_points.items(),
              DistanceTolerance, (SX, SY, SZ))
```

The `get_reachable_nav_points` function takes a list of navpoints and returns a list of all those which are specified as "reachable" and which the bot is more than `DistanceTolerance` units away from[3].

```
            # now work through this list of NavPoints
            # until we find one that we haven't been to
            # or the one we've been to least often
```

[3] "Units" refers to Unreal Tournament distance units, discussed in the Gamebots API: *http://www.andrew.cmu.edu/user/roman/15396/game_bots_api.html*

```
if len(PossibleNPs) == 0:
    return False # nothing found
else:
    self.PosInfo.ChosenNavPoint = self.
      get_least_often_visited_navpoint(
        PossibleNPs)
    return True
```

The function now searches this returned list (unless it is empty) and finds the one visited least often. This is accomplished by the `get_least_often_visited_navpoint` function which searches the list in `self.PosInfo.VisitedNavPoints` (see above).

`self.PosInfo.ChosenNavPoint` is set to this least-visited navpoint. This variable then used by the `walk-to-nav-point` action primitive to actually make the agent run to this navpoint.

5.3.4. Problems Encountered

This section briefly outlines some of the problems I encountered during development. It is intended to highlight difficulties with developing for Unreal and Gamebots rather than problems with the BOD methodology (see elsewhere in this chapter for evaluation of the methodology itself). Errors with PyPOSH are discussed in their own right in chapter 8.

Navigation is a major difficulty in Unreal Tournament as it is in robotics and other areas. There are various reasons for this, including the following:

- At the start of the game, the bot does not know locations of any other parts of the level, and thus cannot use UT's built-in path-finding functions.
- There are no sensory functions available which can tell the bot that it sees a wall, a slope, a doorway etc. This means that the only way it can discover these is by walking into them.
- Even when the bot does know the location of a point of interest there can still be problems with the built-in path-finding functions. For example, if the bot is very close to the desired location but facing the wrong way, an empty list of navpoints will be returned (as there are no points between the bot and its desired location) but the bot will not be aware that it is so close to its destination. (This problem was combatted with the `too-close-for-path` sense).
- It is very difficult to place navigation points in such a way that the bot will be able to follow them easily and not double back on itself or lose the trail. (This does not apply when the path-finding functions are used as these provide a pre-defined list of points for the bot so it does not matter whether the bot actually notices the points or not). This may also be improved by setting the bot's skill level as something higher than "novice". Preliminary tests suggest that the bot's perception is improved by a higher skill level, but that this improvement in fact results in it noticing the points it has already been to before seeing new ones and so still doubling back on itself frequently. This suggests that to make the most of the improved perception, the behaviour would also need modification. However, this has not been tested in detail and remains an area for future work (section 9.2).

- Unreal Tournament's levels are completely 3D, as opposed to simple grid-based layouts.

Expiry of data is a key issue affecting various aspects of the bot's behaviour. For example, if the bot died at a location where the enemy flag was reachable it still believed it to be reachable when it respawned, despite then being in a completely different location.[4] Similar problems occurred because the bot only receives information about objects when it sees them. This means that it could receive information that a flag was reachable and then continue to believe this even if it moved a long way away: if it did not see the flag again, it would not receive a message informing it that the flag was no longer reachable. This is an example of the Frame Problem (McCarthy & Hayes, 1969).

Such expiry problems were combatted in three ways. The main way was the creation of a number of expiry actions which were called periodically to clear particular items of state (e.g. `expire-reachable-info`). As well as this, various items of state were cleared when the bot died. Finally, Gamebot's `CHECKREACH` function was used. This function allows the bot to check whether a given location or object is reachable from the current location.

Some **other problems** encountered are worth mentioning but do not merit detailed discussion. These are listed below:

- Forgetting to rebuild a UT map before saving it means that changes are not reflected when playing the map. This is because rebuilding calculates how the changes will actually affect what the player sees when the map is played. For example, it performs rendering and raytracing.
- As suggested in section 5.3.3, tests against attributes from Gamebots should test for `"True"` and `"False"` rather than the standard boolean `True` and `False`. This is because information from Gamebots is sent as strings.
- Sending an extra space at the front of a location string results in the location being ignored and the bot just standing around.

Gamebots-specific Problems

One problem with Gamebots is that some commands and events do not work:

- `PRJ` is supposed to provide details of incoming projectiles. However, these messages are never generated.
- `INIT` is the message sent to generate a bot. The attributes sent with this message (name and team) are ignored.

These problems (and others) are fixed in the new version of Gamebots, available from *http://www.cs.rit.edu/~jdb/gameAI/gamebots/codechange.html*. However, I was not aware of this version at the time of development.

[4] "Respawning" is UT-jargon for what happens when a player (including computer-controlled bots) dies: the player is returned to his/her own base (or another "respawn point") and given full health. However, s/he loses all ammunition, weapons and armour. The concept is similar to "losing a life" in other computer games, except that there is usually no limit to the number of times it may occur.

When keeping attention on an assailant, the bot strafes (i.e. runs in one direction while facing another) rather than just running. However, the INCH command (moving a small amount forwards) has no strafe equivalent.

Occasionally, the list of navpoints sent from the UT server seems to be corrupted. For example, consider the following printout of these attributes (the problematic data is in **bold**):

{'11': 'CTF-Simple2dt.PathNode20 306.283630,-943.969238,
-205.899719', '10': 'CTF-Simple2dt.PathNode19 179.591705,
-1001.335571,-205.900146', '13': 'CTF-Simple2dt.PathNode22
409.230133,-551.878540,-205.899963', '12':
'CTF-Simple2dt.PathNode21 397.411743,-817.479858,-205.900085',
'15': 'CTF-Simple2dt.FlagBase0
**0.99\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00RCH** {ID
ReachPathThere', '14': 'CTF-Simple2dt.PathNode37 2.160962,
347.320404,-137.227463', '1': 'CTF-Simple2dt.PathNode14 389.730133,
-1862.520874,-205.900085', '0': 'CTF-Simple2dt.PathNode15
383.894440,-2076.516846,-205.900116', '3':
'CTF-Simple2dt.PathNode12 242.659698,-1564.088623,-205.899902',
'2': 'CTF-Simple2dt.PathNode13 355.602875,-1704.071777,
-205.900024', '5': 'CTF-Simple2dt.PathNode17 6.000000,-1346.876953,
-206.516571', '4': 'CTF-Simple2dt.PathNode27 8.000000,-1458.000000,
-204.000000', '7': 'CTF-Simple2dt.PathNode4-6.200005,-1248.000000,
-203.000000', '6': 'CTF-Simple2dt.InventorySpot21 -0.891642,
-1266.071045,-203.000000', '9': 'CTF-Simple2dt.InventorySpot16
2.643697,-1023.192627,-203.000000', '8': 'CTF-Simple2dt.PathNode0
51.432945,-1158.075439,-207.000000', 'Reachable': 'True', 'From':
'90.789177,-2312.877930,-212.100006'}

In this example it appears that two messages – a list of nodes and a RCH message (i.e. a response to the CHECKREACH command) – have been mixed together. This problem and possible solutions are discussed briefly in the future work section of this document.

5.3.5. Evaluation of the BOD Process

Types of Behaviour

One conceptual problem I encountered was that I found myself subconsciously treating the bot's "physical" actions (e.g. running, shooting) and "mental" actions (e.g. remembering information it was provided with from the server, "forgetting" out-of-date knowledge) as fundamentally separate types of entity. Therefore, it was easy to forget the plan file when coding this "mental" functionality. This lead to problems as whilst action primitives run atomically, other aspects of the agent may run in parallel with them. Therefore, coding a mental function as part of the main agent control loop rather than as an action meant that the mental function could alter a variable used by an action whilst that action was running and thus lead to exceptions being raised. For example, clearing a list of navigation-points whilst an action was searching the list.

Problems with functions overwriting data whilst another function was reading it occurred several times before the simple solution became apparent: the action plans should contain both mental and physical actions. An example of this can be seen in actions such as expire-reachable-info in the movement behaviour library. Due to constraints on time, however, I was not able to correct this flaw for all cases. This is discussed in the Future Work section (9.2).

It could be argued that this leads to untidy plans, as they mix both tangible actions of the bot with general housekeeping functionality. However, on reflection I believe that it leads in fact to plans which relate more closely to the action selection of humans: humans must

choose (even if only subconsciously) to devote some of their time to purely mental activities such as remembering information they are presented with, as well as the more obvious physical tasks.

Roll-backs

Whilst I found iteration and prototyping very useful (in creating the functionality to take the bot to the enemy base, for example), I found that rolling back to a previous iteration was not a technique I ever made use of. The primary reason for this was that rolling back would have removed other useful code I had written during the iteration as well as removing the code which proved to be ineffective. It was also the case that removing the code "by hand" rather than by rolling-back was simple and error-free, meaning that "automatic" removal by rolling back would not have been an advantage.

This problem of removing useful code when rolling back suggests that my iterations were too large. However, the usefulness of rolling back actually *decreases* with smaller iterations as it then become easier simply to remove the code by hand. It could be argued that the work done during an iteration was too wide-ranging for rolling-back to be effective but, even if this were true, I certainly did not notice it as a problem during development.

Specification Revision

Although, as discussed above, I did not perform many alterations to the initial decomposition produced, revision of specifications was still something I found useful. For example, there were a number of occasions when I modified action primitives to facilitate re-use of functionality.

One example of maximisation of re-use was separating the functionality which recorded who recently attacked the bot from that which made the bot turn to face him/her. This in turn allowed the plan file to be simplified by combining two Action Patterns into one. In fact, the combining could have been performed without separating the code, but the plan with the separated code was clearer and thus made this possibility of reuse more obvious to me.

Another form of refactoring specifically suggested by Bryson (2001) is to use the simplest type of POSH component possible (i.e. a primitive rather than a sequence rather than a competence). The recording / facing situation mentioned above also provided an opportunity for this: by increasing the complexity of the `set-attacker` primitive, I was able to use it (and thus the entire Action Pattern) in both the case when an attacker was spotted actually attacking the bot, and when the bot had to look around before finding an enemy it believed was the assailant.

There are a number of cases, however, where I deliberately did not combine sequence elements into new primitives. For example, the elements of the following sequence are used nowhere else:

```
(AP respond-to-visible-attacker (seconds 4)
    (set-attacker face-attacker shoot-attacker))
```

However, there are two reasons for keeping these separate. The first is that these some of these actions are likely to be required in their own right in future plans. The second is that this separation makes the plan file far more readable.

Agent State

As discussed in section 5.3.2, and highlighted in the code examples in that section, I made substantial use of state to store details about the Unreal world. Like the primitive-

complexity vs plan-complexity tradeoff (see section 5.3.3), there is also a trade-off between plan-complexity and the amount of state required. For example, Bryson (2001; section 6.5) gives the example of an insect which could either have two plan elements for hitting something on its left side or its right, or have some state indicating which side it hit something on, and a single plan element whose primitive uses this state to decide whether to move left or right.

I believe that the complexity of the information my bot requires means that the need for extra state usually prevails. For example, the state holds the attacker's ID, something dynamically generated by UT and thus not usable in a pre-written plan. Furthermore, even if it were, the need for *persistence* again suggests that state is required: the bot needs to "remember" that a certain agent attacked it when that agent is no longer shooting at it (and thus there are no external cues).

The need for complex state is a point made by Laird (2005; the full e-mail is given in Appendix D):

> I might be wrong, but I think this is where the behavior-based [sic] approaches really make it tough on themselves by not making it easy to encode complex state information, derived from many sources and both about the past and the current situation.

5.3.6. POSH's Contribution to Development

The POSH plan produced was the most complex ever published, and I found the POSH Action Selection mechanism a great help in the development of the agent. This section explores three of the key advantages: more focussed development, multi-threading, and frequency & retries. It begins, however, with a justification of my claim of the complexity of the plan:

The most complex POSH plan ever published

This section compares the final plan given in Appendix A (a version of `bodbotattack.lap`) with the most complex plans published to date: `stay-groom.lap` (see Bryson, 2003) and `educate-me+monk.lap` (see chapter 9 of Bryson, 2001). Both these plans are available from *http://www.cs.bath.ac.uk/~jjb/web/posh.html*.

The following table provides various relevant statistics:

	`bodbotattack`	`stay-groom`	`educate-me+monk`
Drive Collections	1	1	1
Competences	6	2	4
Action Patterns	11 (3 of which are multi-element)	0[5]	1
Drives	14	4	5
Deepest Nesting	4	0	2

Table 5.1 Statistical comparison of POSH plans.

[5] This plan file includes an Action Pattern as part of its documentation, but no others.

The bracketed figure given for `bodbotattack` is the number of Action Patterns with more than one element. Single-element patterns are required as PyPOSH does not allow Drive Collection elements to trigger action primitives directly (see section 8.3.3).

"Deepest Nesting" refers to the depth of nested competences and action patterns. The deepest potential nesting in `bodbotattack` is 4: The top-level Drive Collection triggers the `get-to-enemy-base` competence which can in turn trigger `wander-around`. This can then trigger the `avoid` pattern which triggers the `then-walk` competence. By contrast, `stay-groom`'s competences only trigger primitives whilst `educate-me+monk`'s deepest nesting is 2.

A final point worth noting is that whilst `educate-me+monk` comes closest to `bodbotattack` in terms of statistics, `educate-me+monk` is not a real-time plan. Working in real-time throws up many additional challenges for development and control of agents. Similarly, whilst robots developed using BOD (e.g. Bryson, 2001; Chapter 7) may have very complex *behaviours*, their plans are still simpler than that which I produced.

More Focussed Development

The greatest advantage of having an Action Selection mechanism was simply the fact that I needed not code it for myself! However, the advantage is greater than mere convenience: Not needing to think about *how* action selection was going to work left me free to think about *what* I should make the bot do, thus making development more focussed as less of my attention was on secondary concerns and more on the actual behaviour of the bot.

Weiser (1994; abstract) summarises this advantage neatly:

> For thirty years most interface design, and most computer design, has been headed down the path of the "dramatic" machine. Its highest ideal is to make a computer so exciting, so wonderful, so interesting, that we never want to be without it. A less-traveled [sic] path I call the "invisible"; its highest ideal is to make a computer so imbedded, so fitting, so natural, that we use it without even thinking about it.

Multi-threading

"Multi-threading" here refers to both the use of threads within Kwong's (2003) poshbot, and to the parallel nature of POSH. In practice, this was particularly useful in two areas: synchronous interaction with Gamebots and the bot, and the ability for higher-priority drives to interrupt lower-priority ones.

Interaction with Gamebots and the Plan Driver

UT is a dynamic environment and as such messages about the world and the player may be received and must be dealt with at any time. I was fortunate in being able to build on *poshbot* (designed by Kwong (2003) as a means of demonstrating PyPOSH), which already contained the necessary multi-threading functionality to handle synchronous and asynchronous messages from Gamebots whilst plans are being executed.

Interruptions

The ability for an agent to temporarily stop doing something to deal with a more pressing need is a fundamental part of action selection and I found this to be well-supported in POSH Action Selection. The existence of a schedule, allowing interrupted tasks to be continued, was a major boon: coding such functionality by hand would have been very difficult and would probably not have resulted in such a general-purpose solution.

Frequency and Retries

Many of the "mental" actions (see section 5.3.5) required the *frequency* attribute of POSH's drive collection elements. For example, every 30 seconds the bot "forgets" the details of whether flags are still reachable (otherwise it would continue to think they were reachable even when it had moved far away from them, as this is only set when it sees the flag). This proved a very useful feature and, in my opinion, makes the construction of realistic action selection far easier.

The *retries limit* for competence elements was also useful to help represent the "try x and then try y if x doesn't work" structure. Consider this extract from one of the competences:

```
(find-base (trigger((reachable-nav-point)))
    walk-to-nav-point)

(find-nav-point (trigger((succeed))) rotate 10)

(wander-base (trigger((succeed))) wander-around)
```

The use of retry limits allows both the final two elements to have a trigger of succeed (i.e. the function which always returns true) yet still both to have a chance to run: if rotating does not result in a navigation point becoming visible (and thus `find-base` being triggered), the bot will attempt to wander around. The limit is very useful as it means that the bot does not continue rotating forever if no navigation point is visible, but equally that is does try this action before it wanders around.

Limitations of these attributes

Whilst, as mentioned above, being able to specify the frequency and retries limit for elements was very useful, there were occasions when I felt constrained by this. For example, the bot's "forgetting" of flag details was arguably sub-optimal: a better solution would have the bot forgetting 30 seconds after receiving the information. By contrast, a frequency of 30 seconds means that the time between the bot seeing the flag and forgetting about it is anything up to 30 seconds, but potentially much less, as it is 30 seconds after the last "forgetting". However, it could also be argued that such an element of randomness makes the bot more human-like. This frequency problem could of course be solved by having the behaviour set a timer, another example of the trade-off between complexity of plans and complexity of behaviours.

5.4. Summary

This chapter has introduced BOD's development process and my experiences of using it. Specific information about development has been provided, including details of behaviour modules, and classes and functions relating to agent state. The development of sensory and action primitives has been illustrated by discussion of a series of examples. Problems encountered have been discussed, including some specific to Gamebots. Finally, the process and POSH's contribution to it have been evaluated, and the complexity of my POSH plan demonstrated. A summary of my evaluation of BOD can be found in Chapter 9. The next chapter, however, continues evaluation by way of a comparison with other architectures.

6. Comparison with Other Architectures

6.1. Introduction

A large part of my analysis of BOD is based on comparisons with other architectures and methodologies. One problem I encountered was that there are a surprisingly small number of existing methodologies for complex agents. This point has been noted elsewhere (e.g. Kinny et al., 1996).

There are a number of methodologies for *Multi-Agent* Systems [MAS], and a review of several of these can be found in Inglesias et al. (1999). However, although the majority of these MAS methodologies do refer to the structure of individual agents, most do not go into enough detail to be interesting (although see the discussion of Kinny et al's methodology in section 6.3). One possibility would be to examine whether these multi-agent methodologies could be adapted for complex agents. However, initial examination suggests that this would not be very enlightening: common concepts from these models (such as roles and organisations) seem too broad for meaningful adaptation.

This chapter makes much reference to ACT-R, Soar and EPIC. These architectures are introduced in Chapter 3.

6.2. The Rational Rose Approach

One approach studied in this chapter (hereafter referred to as the "Rational Rose approach") but which has not been previously introduced is that of specifying agents via UML [Unified Modelling Language, e.g. Odell (1998)] diagrams and automatically generating code from this specification using Rational Rose software[6].

Although there does not appear to be a huge amount of information available about this method of agent creation, and I do not have practical experience with it, it has been included as it is considerably different to the other approaches. Furthermore, it is currently being used to create Unreal Tournament agents as part of assignments at Carnegie Mellon University (e.g. Douglas, accessed 2005; Carnegie Mellon University, accessed 2005). The approach is touched on briefly in various sections of this chapter, and analysed in its own right in section 6.7. The Rational Rose approach can make use of subsumption, discussed in section 6.8.

6.3. Getting Started with Development

This section compares BOD's Initial Behaviour Decomposition to the initial processes of other architectures. BOD's approach is evaluated in its own right in section 5.2.2.

The fact that BOD supplies relatively detailed instructions as to how development should begin is an important point which can easily be taken for granted. It could be argued that doing so is going beyond what is required of an architecture. However, even the best-designed architecture is little practical use if no-one can develop with it. Bryson (2000b) even goes as far as stating that the best way to consider agent architectures is as design methodologies.

[6] See *http://www.rational.com*.

An important distinction between the initial development processes of BOD and of ACT-R or Soar is that using BOD one begins by considering what agents should *do* whereas in ACT-R and Soar one must consider what an agent needs to *know*. (This could be a reflection of the fact that ACT-R and Soar were designed primarily for the modelling of human cognition.) I believe that BOD's approach is a better fit to the problem, since to discover what an agent needs to know, you in any case must first consider what the agent should do. Furthermore, putting the components of a a Soar model together is a precise process: the division of knowledge into problem-spaces must be done with consideration of the relationships between impasses to ensure that learning is useful. BOD's iterative process which allows for easy alterations appears far simpler.

Arguably, one could begin development in ACT-R or Soar by writing production rules ("associations" in Soar terminology). However, as well as going against the recommended process (e.g. Lehman et al. 1996 p. 13) this would be rather difficult since writing production-rules relies on a substantial amount of information about what knowledge is held in the model and how that knowledge is structured.

Nevertheless, there are situations where the approach of ACT-R or Soar would be a better fit to the problem. For example, the modelling of human cognition and recall. However, I believe that this side of Artificial Intelligence is only a small part of the agent domain, which encompasses far more than AI alone.

EPIC's approach is a little more similar to that of BOD. Kieras and Meyer (1997; p. 409) state that the builder of an EPIC model must supply the following information:

1. A production-rule representation of the task procedures
2. Task-specific sensory availabilities and perceptual-processor encodings and timings.
3. Any movement styles not determined by the task requirements.

This process begins with the writing of production-rules, meaning that, like BOD, there is at this stage more emphasis on what is done than on what is known.

The Rational Rose approach is similar to BOD in that it deals with tasks ("Use Cases") rather than knowledge. Again, however, the initial stages of development with this technique are not discussed in detail. Rather, the majority of UML techniques relate to interactions between objects or to decomposition of object data (e.g. Entity-Relationship modelling), areas which are less relevant here.

There are ways in which BOD's Initial Decomposition could be improved, however. This becomes apparent when comparing it with Kinny et al's (1996; p. 61, my emphasis) methodology for BDI [Beliefs-Desires-Intentions] agents:

> Our methodology for the development of these models [for the internal structure of agents] begins from the services provided by the agent and the associated events and interactions. These define its purpose, and determine the top-level goals that the agent must be able to achieve. *Analysis of the goals and their further breakdowns into subgoals leads naturally to the identification of different means, i.e., plans, by which a goal can be achieved.*

There are many parallels between this process and BOD's Initial Decomposition. However, the key difference is that the top-level goals are used to establish the means by which they

may be achieved. In BOD, the list of top-level goals is created *after* sequences of actions are created.

I believe that, for an agent such as mine, BOD's ordering is slightly confusing: it is easier to determine likely sequences of actions once top-level goals are identified (as per Kinny et al's method). However, there are also areas where BOD's method would be preferred. For example, if one aims to replicate some simple real-world "agent" but to a great degree of detail, it would make sense to identify what it can do (e.g. sensory and action primitives) before identifying top-level goals. This is because the primitives are a relatively fixed-size set, whereas the set of goals is likely to grow as the agent is developed to match the range of abilities of the real-world version ever more closely.

Overall, I believe that for creating a complex agent such as mine, BOD's Initial Decomposition process is superior to the corresponding procedures of EPIC, ACT-R or Soar. There are three reasons for this:

1. The process is specified in more detail
2. The process is a closer map to the actual tasks carried out when designing an agent
3. The process requires less detail at earlier stages, allowing instead for highly iterative development as requirements and constraints become more apparent.

However, as comparison with the methodology of Kinny et al. (1996) suggests, in some cases a slight re-ordering of the stages of the process could make it more effective.

6.4. Evolutionary Design and Development

Soar, ACT-R and EPIC do not explicitly state the processes which should be followed during development. As with Initial Decomposition the fact that BOD does could again be considered an advantage. It could also be argued that specifying this is overly prescriptive. However, it makes sense to develop in a way which suits the architecture one is developing in, and by providing detail on the design process BOD makes this easier.

Attempting BOD's iterative, evolutionary development process in Soar could lead to complications. The primary reason is the complex inter-dependencies of Soar's production rules. Consider the following extract from the example associations given in section 3.2.3:

(a1) If I perceive that I have finished reading a paper

then suggest a goal to summarise that paper.

(a2) If there is a goal in WM to summarise a paper

then...

(a2) clearly depends to some extent on (a1), as (a1) suggests the goal which triggers (a2). However, such dependencies are not recognised by Soar, the dependency exists only in the mind of the developer. I claim, therefore, that it would be very difficult to see what rules would be affected when a rule is modified, deleted or added. Furthermore, consider the difficulties arising from moving something to a different problem-space: This would result in an entirely new set of impasses being created when the model runs, potentially drastically transforming the relationships between elements. By contrast, BOD is specifically designed so that changes can easily be made at any stage of the development process.

The fact that in Soar and ACT-R knowledge is specified before an agent's actions would also make iterative development more difficult: changes to what the agent does would be

likely to also require changes to be made to the knowledge an agent holds, thus leading to further alterations and risking affecting other parts of the system.

The Rational Rose approach seems to suggest incremental development, with an agent gradually being extended with further Use Cases. However, it appears to suffer from a similar problem to Soar: that of dependencies between components. Carnegie Mellon University (accessed 2005) gives an example of avoiding potentially problematic interactions between actions by manual insertion of delay states, a rather inelegant solution.

Revision of specifications and the supporting of maintenance via structured commenting (Bryson, 2001; section 8.4.1) are concepts which could, to some extent, be applied to ACT-R and EPIC. Again, however, the complex dependencies in Soar would make such revision more difficult. Rational Rose allows documentation to be associated with any diagram element which should ease the process of creating structured documentation. However, I expect that this would also produce disjoint, fragmented documentation, as it would be spread across all the diagram and sub-diagram elements.

The opportunities for incremental development given by the Subsumption Architecture are discussed in section 6.8.1, where this architecture is also introduced.

6.5. Goal-Driven Development

The central thrust of the development process involved taking goals from the list identified as part of the Initial Decomposition, and coding the corresponding plan and behaviour elements to support them. I believe that such a focus on goals leads to more robust designs as a domain's high-level goals are less volatile than behaviours or lower-level plan elements. Therefore, any changes to behaviours or low-level plan elements can be easily coded as another (or a replacement) way of achieving the high-level goal. This point is also discussed in Kinny et al. (1996).

6.6. The Architecture

The very fact that BOD provides the POSH architecture is an advantage. For example, whilst the Rational Rose approach does enable a lot of code to be automatically generated, it requires much of the action selection mechanism to be manually coded. This can easily become a complicated and daunting task.

6.6.1. POSH's Hierarchical Structure

One of POSH's greatest strengths is its hierarchical structure. This greatly increases the clarity of the plan files. On a basic level, decomposing the plans into sub-components makes them easier to read. On a more technical level, the structure leads to simpler triggers. Consider rule (a3) from the example Soar associations in section 3.2.3:

If using the Summary Creation problem space and the **amount_summarised** is < *100*

Soar's associations are required to explicitly state the problem space. A more hierarchical approach such as BOD's means that triggers can be designed with more certainty about context (e.g. we are already in the context of the parent competence).

The use of UML means that the Rational Rose approach is very hierarchical. However, this hierarchical decomposition requires a large number of sub-diagrams. I believe that this could become confusing, especially as it could mean that the detail of one part of the system may only be examined in isolation from other parts. The reasons for this could include a

lack of screen space, or the use of modal windows for component properties (only one of which may be open at any given time).

6.6.2. Priorities and Emergent Behaviour

As well as the contextual information provided by the plan hierarchy (discussed above), BOD's priority system means that if a given component fires then we can be certain that nothing more important can fire, reducing the need for extra "negative triggers" testing against the preconditions of more important elements.

EPIC's approach to this issue is its cognitive parallelism whereby all rules which match are fired simultaneously. Whilst this has advantages (Kieras and Meyer (1997) suggest a closer match to the cognitive processing of humans), I suspect that this would cause problems with regard to interactions between the items which are running in parallel. For example, it could make consistent behaviour difficult to achieve, and make debugging very complicated as the cause of some event could be emergent from these interactions rather than something obviously pre-coded.

Although ACT-R's pattern matching / utility-based retrieval system is also arguably a close match to human cognition, the issue of emergent behaviour is again likely to make debugging and consistent or predictable behaviour difficult to achieve. Note particularly that a new chunk may be retrieved in place of some existing one in some cases but not others. To discover exactly when this would occur would involve studying much of the existing facts. By comparison, modifying BOD plans is far simpler as such dependency issues do not arise.

Further discussion of utility-based action selection

BOD supports what could be called "manual" utility allocation: a programmer is able to try a number of approaches for responding to a given situation, adopting the one he or she finds most useful. This method is arguably more accurate as the programmer can adopt a broader view of the situation than the system's simple "has this worked in the past?" For example, it allows side effects to be considered, and can consider more complex outcomes than the simple success and failure used in ACT-R.

However, the issue of emergent behaviour means that the precise effects of different approaches may not always be obvious to the programmer. Furthermore, it may require extensive testing to determine which method is best, something which is especially difficult if the programmer has only limited control over the simulation, as is the case in Unreal Tournament. One area where "automatic" utility allocation could certainly be useful is where an environment cannot be simulated accurately, for example sending a robot to a previously unexplored planet.

6.6.3. Frequency and Retries

As discussed in section 5.3.6, I found the ability to limit frequency and the number of retries to be a useful feature. Achieving such effects in ACT-R would require chunks to be extended with further slots, increasing the complexity of the system and potentially requiring many chunks to be modified beyond just the one which is to be limited.

6.6.4. Modularity and Re-use

The fact that POSH plans are stored as entirely separate files from the code for behaviour primitives and from the Action Selection mechanism is a major advantage. Specifically, this means that radically different agents can be created from a single set of primitives simply

by supplying a different plan. This appears to be a far faster and simpler approach than, for example, rewriting production rules for Soar, ACT-R or EPIC.

A Comparison with JACK

As well as BOD, another agent framework designed with modularity and re-use as primary objectives is JACK (Howden et al., 2001). JACK is an agent framework which extends the Java language with both new syntax and new classes. One of the key ways in which the concepts of modularity and re-use are supported is by Capabilities. Capabilities allow agents to share plans, event-handlers and state. They work in a similar way to macro expansion: code of the form `#has capability <Capability Name> <Local Name>` is replaced by the statements stored in the relevant capability file.

This sharing of plans can be likened to BOD's use of external plan files, whilst the sharing of state can be likened to several BOD agents making use of an external class designed to store agent state.

Whilst I believe that Capabilities seem to perform a useful role in JACK in drawing various aspects together, I do not believe that BOD is weaker for not have such an encompassing component. The reason for this is that BOD agents similar enough to share many components in common would probably use the same classes (i.e. behaviour modules) and only differ in their plan files. Thus only a single set of classes would be required, not one for each agent, and so the issue of sharing would not arise.

Furthermore, whilst it is true that Capabilities effectively modularise functionality, similar modularisation is already very easy in BOD via behaviour decomposition and the structuring of plan files.

However, there is one area where further sharing could be beneficial, the sharing of plan components across plan files. Consider this extract from the Initial Decomposition (see Appendix B):

> If the bot were a defender by default, the list [of initial drives and goals] would be as above, but with the following changes:
>
> The following drive would be inserted between 8 & 9:
>
> Run after enemy carrying our flag
>
> Item 11 would be removed and items from 14 onwards would be replaced by the following:
>
> 14. Run to own base
> 15. Find medical kit, weapon or ammunition (as required).
> 16. Look for vantage point
> 17. Wander around near our flag / vantage point
> 18. Pick up nearby medical kit (if health not in dangerous range)

The current implementations of POSH would require those elements used by both attackers and defenders (i.e. most of the plan) to be repeated in the LAP file for each. Redundancy could be reduced by allowing the sharing of plan elements across files.

One possible way to achieve this would be via pre-processing. For example, the plan files could contain code such as `#uses <element name> from <file name>`. During

pre-processing, this would be replaced with the relevant POSH element. For systems where a number of similar plans are used, a plan file could be treated as "abstract" (i.e. not designed to be run directly) and this file could contain all the elements shared between the plans.

Sharing *within* plan files is discussed in section 7.3.2.

6.6.5. Explicit Goals

Kinny et al. (1996, section 4.3) discuss how plans using their BDI methodology are both event-driven and goal-driven, i.e. actions can be triggered in response to some external event or as a means of achieving some internal goal. This is true to some extent of POSH: the execution of competences and drive collections is controlled by goals (i.e. execution terminates if the goal condition is met), whilst the elements of competences and drive collections are more like event-driven elements, firing as a response to triggers evaluating to true.

The firing of elements could be made more goal-driven by explicitly storing goals and providing sensory primitives to test against them. However, I believe that this would be an unnecessary extra level of abstraction.

Furthermore, I believe that such goal-directed thinking can cause problems for the programmer also, by forcing him or her into a simplified view of the world. Consider the following fictitious scenario: A bot is busy defending himself from attack when he sees another enemy player running away with his team's flag. However, he does not run after the flag-carrier as his own dangerously low health means that killing his attacker is more important.

The programmer needs to ensure that something triggers the bot to run after the flag-carrier after the current attacker has been killed. In a purely goal-directed world, this would be accomplished by adding a "chase flag-carrier" goal to the goal list. This goal would then not be found for execution until the "kill assailant" goal had been accomplished. However, this apparently simple solution would cause problems: there is no guarantee that the flag-carrier is still visible after the attacker has ben killed. I claim that a programmer thinking in terms of events rather than just goals would know that goals can be an unnecessary abstraction and so would instead be drawn to considering how the event could be replicated. In this case, storing details of the flag-carrier in some state which could then be checked by sensory triggers. Since more detail is stored than just the fact that the bot needs to chase the flag-carrier, there is a higher chance of the bot knowing enough to be able to catch him.[7]

It could be argued that a greater use of goals makes it easier for partial plan representation to be used. For example, Ingrand et al. (1996, section 3.1) discuss how PRS [Procedural Reasoning System] allows plans to add goals to the list of active goals and rely on some other plan which can satisfy them being chosen by the meta-interpreter. This is possible as plans declare explicitly which states they bring about, whilst goals describe desired states.

However, similar things are achievable in POSH: If some action causes an event, it can be guaranteed that if some component can deal with that event in the current context (i.e. all other required triggers return true), and its priority is high enough, then that will fire. All

[7] For example, if the bot had recorded the team the flag-carrier was on, he could use his knowledge of where that team's base was to guess where the flag-carrier was headed.

goals must be a result of some event or condition (internal or external) and POSH is simply concerned with the event rather than the associated goal.

6.7. Further Analysis of the Rational Rose Approach

This section outlines aspects of the Rational Rose approach which are noteworthy but do not relate specifically to the comparisons in earlier sections of this chapter. The Rational Rose approach itself is introduced in section 6.2 above.

The fact that the Rational Rose approach provides automated code generation could make it easier to use for those unfamiliar with programming. However, even the simplest agents still seem to require some programming (e.g. Carnegie Mellon University, accessed 2005). Furthermore, the fact that the code is spread between various diagram components could be confusing, as the connections between the different pieces of code are not immediately apparent.

Code generation can be performed with a number of target languages. However, the fact that some code must be written manually makes this approach less useful as this code must be rewritten. (The code must also be found, which could require quite a lot of searching through diagrams and properties-windows.)

One important point about the Rational Rose approach is its visual nature. This could be very useful for people who find diagrams easier to understand than text. Furthermore, the connection with UML modelling could make it easy to adapt existing specifications. UML also provides an existing, relatively well-known structure within which to work. However, the approach is arguably *too* abstract. Kinny et al's (1996, section 4.3) plan models are similar to some of the diagrams used by the Rational Rose approach but are in my opinion superior as they offer a better fit to the situation being modelled (i.e. they are less abstract). For example, states can cause the generation of subgoals and may contain programming constructs such as loops.

6.8. Comparisons with the Subsumption Architecture

The Subsumption Architecture was introduced in Brooks (1986). Its influence on BOD is apparent in BOD's use of behaviours, discussed in section 2.3. This section introduces the architecture, examines its support for incremental development compared to BOD, and explores how its structure leads to robustness.

6.8.1. Incremental Development and Behaviour-Interaction

The Subsumption Architecture is designed for incremental development whereby a simple system is expanded by adding further behaviours ("levels of competence", *ibid*, p. 10). These behaviours are entirely self-contained, in that once one is added it is not changed. Rather, new higher-level behaviours can interact with it via suppression and inhibition:

- A higher-level layer may *suppress* the inputs of lower-level layers. I.e. it may replace the existing input with one of its own.
- A higher-level layer may also *inhibit* the outputs of lower-level layers. I.e. destroy the signal before it is acted upon.

Brooks (1986) describes a navigation robot which wanders aimlessly, avoiding obstacles. The robot is then augmented with a new level of competence which allows it to move purposefully towards areas of interest. This new level must inhibit the output of the

wandering layer so that the robot only wanders when it is not required to move towards some particular location instead.

This incremental approach offers some obvious parallels with the BOD development process. However, the concepts of subsumption and inhibition mean that there are some important differences also. The predominant factor is that subsumption and inhibition greatly increase the interdependencies between elements. This means that while development may be incremental, adding new behaviours must be done with great consideration for potentially every other behaviour in the system. In BOD this difficulty is far reduced.

I claim that this constraint means that more consideration need be given to the order in which new behaviours are added in the Subsumption Architecture. For example, if a new behaviour is added which is at a higher level than any existing one then all that need be considered is which lower behaviours it must subsume or inhibit. However, if a behaviour is added "between" two existing ones (i.e. a higher priority than one and a lower priority than the other), then the programmer must consider which lower behaviours the new behaviour must subsume / inhibit *and* which higher behaviours must subsume / inhibit the new one. Even if the programmer decides to avoid this problem by only ever adding behaviours which are a higher priority than all existing ones (as the Subsumption Architecture in fact requires), there is then the constraint that these lower levels must be perfect the first time they are created, as they cannot be modified.

In fact, even assuming that lower-level layers are completely correct, things are not simple. Consider the following slightly simplified extract from the `bodbotattack.lap` plan:

```
((pickup-weapon-as-unarmed
    (trigger  ((see-reachable-weapon)
    (are-armed False))) get-weapon))

((respond-to-attack-since-health-low
    (trigger ((taken-damage) (own-health-level 30 <)
     (armed-and-ammo))) respond-to-attack))

((attack-enemy-with-our-flag
    (trigger ((see-enemy-with-our-flag)))
        attack-enemy-carrying-our-flag))

((take-enemy-flag-from-base
    (trigger((enemy-flag-reachable)
     (have-enemy-flag False))) go-to-enemy-flag))

((respond-to-attack-health-not-low
    (trigger ((taken-damage) (armed-and-ammo)
     (is-responding-to-attack False)))
        respond-to-attack))

((hit (trigger((hit-object)(is-rotating False)))
    avoid))

((go-home (trigger((have-enemy-flag)))
    go-to-own-base))

((get-yourself-to-enemy-base (trigger((succeed)))
    get-to-enemy-base))
```

To achieve this prioritisation of tasks in the Subsumption Architecture could, I claim, require almost every element to subsume many of the elements below it! For example, the drive to

respond to an attack when the bot's health is low (respond-to-attack-since-health-low) should initially prevent the bot from taking the enemy flag, from attempting to go to the enemy base and from attempting to go to the home base. Even the lower-priority drive to respond to an attack when health is not low would need a more complicated trigger to prevent it running in parallel and thus commands being sent twice. However, since the lower-level behaviours are not modified once they are added, this would need to be accomplished by subsumption also! For a system of any size, subsumption and inhibition would clearly lead to a large number of very complex interactions.

One possible way to simplify this tangle of interactions would be via the idea of action groupings. For example, outputs from go-home, get-yourself-to-enemy-base and take-enemy-flag-from-base could all pass through a single "forwarding" point. Thus only the outputs of this forwarding point would need suppression. However, this approach has two major disadvantages:

- It is very coarse-grained and inexact. There are likely to be occasions where only outputs from some items in the group need suppression, but dealing with these cases separately makes it less worthwhile having the group at all.
- When considering interactions with different high-level behaviours, the lower-level behaviours could need grouping differently.

Overall then, this problem of interaction is a major obstacle to the creation and development of systems. However, it is unfair to end on this note and not to consider the area of robustness, one where the Subsumption Architecture claims a major advantage:

6.8.2. Robustness

The key role played by subsumption means that the Subsumption Architecture is fairly robust. For example, if a higher-level behaviour fails to act in time to suppress a lower-level one (or fails to act at all), the robot being controlled will continue to act intelligently, albeit at a lower competence level.

The fact that the failure of a high-level behaviour results (even though only indirectly) in a lower-level behaviour running instead is one advantage the Subsumption Architecture has over POSH. In POSH, a "broken" or ineffective plan component will simply continue to be executed until its trigger is no longer true, or until some higher-level element takes priority.

One could possibly imagine increasing the robustness of a UT agent by keeping track of the number of messages sent to Gamebots. If no messages were sent for a substantial period of time then it could be assumed that the current high-level drive had failed and the plan interpreter could attempt to execute a drive element of a lower priority. However, this would be a very imprecise mechanism and would involve an inelegant overlap between the plan interpreter and the low-level agent actions (i.e. messages to Gamebots).

6.8.3. Overall Comparison

Overall I believe BOD offers significant advantages over the Subsumption Architecture. True evolutionary development is possible, without the constraints of messy component interaction or the requirement that development of lower-level layers be frozen once the layers have been included in the architecture. However, BOD's lack of robustness is unfortunate.

7. General BOD Evaluation

7.1. Introduction

Whilst much evaluation of BOD can be done in the form of comparisons with other architectures (as in chapter 6), the architecture also merits analysis in its own right. This chapter performs such analysis. This chapter also evaluates POSH with respect to criteria for action-selection mechanisms proposed in Tyrrell (1993), and summarises existing analyses of Extreme Programming, a technique with interesting similarities to BOD's design methodology.

7.2. The Methodology

7.2.1. Lessons from Extreme Programming

The central role of iteration in the BOD process invites parallels with the Extreme Programming [XP] methodology (Beck, 1999): XP involves a frequently repeated cycle of Analysis, Design, Implementation and Testing, whilst BOD's cycle involves implementing functionality from the pre-written specification, testing and debugging this new functionality and revision of specifications. Similarly, BOD's revision of specification is parallelled by XP's emphasis on refactoring.

The popularity of Extreme Programming has lead to a number of evaluations, both experimental and theoretical, from which further opinions on the value of iterative development and of refactoring can be gained. Obviously this overview is far from comprehensive, but it does provide some interesting opinions.

The Evaluations

I considered the following XP evaluations:

Rumpe and Schröder's (2002) *Quantitative Survey on Extreme Programming Projects* presents the results of a detailed questionnaire sent to 45 developers who are or have been using XP for a project.

Müller and Tichy (2001) discuss their observations of XP's use by students at the University of Karlsruhe in Germany.

Karlström (2002) reports on the use of XP for a small development project for the company Online Telemarketing in Sweden. XP was chosen as the customer had a poor idea of the system required at the start of the project.

Bossi (2003) discusses the use of XP in the development of a portfolio-watching application for Credit Suisse Italy.

Lappo (2002) describes how a group of Masters students at Brighton University fared using XP for a 12-week project for web-based resource management.

As might be expected from the title, Keefer's (2002) *Extreme Programming Considered Harmful for Reliable Software Development* paints a negative picture of XP for anything except small projects with talented engineers and shallow technical specifications. The report has a mainly theoretical basis but also draws on Keefer's experiences as part of an XP team.

Iterative Development

XP's "Small Releases" (i.e. iterative development) scored quite highly in Rumpe and Schröder's survey: The XP elements were rated from 0 (not used at all) to 9 (strongly used), with Small Releases scoring an average of 6.86. Furthermore, Small Releases were viewed as very helpful and caused difficulties in only 4.4% of cases.

Müller and Tichy's students, by contrast, nicknamed designing in small increments "designing with blinders" since they felt limited by not being able to look at the more general picture. However, the paper admits that this may be due to the students' lack of experience, and that incremental design did ensure rapid feedback about code. In fact, BOD would solve this "blinders" problem: whilst it does make much use of incremental development, performing initial behaviour decomposition (section 5.2) allows the developer to consider the bigger picture, something kept in mind throughout development.

Karlström and Lappo both found that the initial iteration took longer than the others due to a lack of initial familiarity with the technology. However, Lappo observed that frequent iteration forced simpler designs to be created.

Keefer claims that iterative development makes the calculation of total project cost or schedule very difficult. I believe, however, that this conclusion is not especially relevant: a key reason for the use of XP is when a project which may change frequently and rapidly. Any cost or time estimates for such a project are likely to be inaccurate, whatever the methodology followed.

Refactoring

Rumpe and Schröder also report positively on Refactoring, its average on the not-used (0) to strongly-used (9) scale being 7.27 and participants declaring it helpful. Rumpe and Schröder also hypothesise that refactoring was one of the reasons that respondents found the cost of late changes lower in projects using XP. However, the survey also reports that 20% of respondents had difficulties with refactoring.

Although Bossi found that more of the development time was spent on refactoring than coding (44% vs 37%), he reports favourably on it, noting that less code was wasted and more code was able to be re-used. These advantages are among those outlined by Bryson (2001) as the reasons for revision of specifications in BOD.

The general consensus from those using XP with students was that projects were either too small to warrant (major) refactoring or that students did not appreciate the importance of it.

Keefer argues that refactoring leads to the need for alterations to the test suites and risks introducing further errors, and that a better solution is simply to write good code from the start. I consider this a slightly naïve view. Furthermore, Keefer's argument ignores the fact that well-designed test suites should operate on a program's *interfaces*, rather than on the main program code. Refactoring this program code would then not affect the test suites as the interfaces would not change. However, if the test suites are to remain entirely unchanged throughout the development process then the interfaces must be correct right from the start.

Conclusions

The most detailed evaluation considered (Rumpe and Schröder's) gives a positive view of both iterative development and refactoring, suggesting that BOD is right to emphasis these aspects of development. Although Müller and Tichy's students were not positive about

iterative development, I believe that the difficulties they encountered with iterative development would be solved by BOD's Initial Decomposition. Similarly, whilst Keefer is doubtful about the effectiveness of either technique, I believe that there are significant flaws in his arguments, as explained above.

7.3. POSH Plans

This section explores some of the issues which became apparent during my use of POSH plans. It covers syntax issues, improvements to sharing to reduce redundancy, POSH's use of parallel plan elements, and possible improvements to documentation. The discussion includes recommendations for changes to POSH and the plan interpreter as well as suggestions for things plan writers themselves can do to simplify the development process.

7.3.1. Syntax

Whilst syntax issues will be largely avoided when an IDE [Integrated Development Environment] becomes available for POSH, they still present a problem to the hand-coder and are arguably reflective of deeper problems in some cases.

Element Names

Consider the following extract from the description of the LAP file format (Bryson, 2001; pp. 225 - 226):

```
drive-element :: (<name> (trigger <ap>) <name>
[<sol-time>])
```

This defines the structure of Drive Collection elements such as the following from my `bodbotattack.lap` file:

```
((attack-enemy-with-our-flag
    (trigger ((see-enemy-with-our-flag)))
        attack-enemy-carrying-our-flag))
```

The problem is that elements require two names: the first is the element's own name (not referred to elsewhere in the plan) and the second the name of the POSH element to be fired if the trigger evaluates to true. This double-naming can make plan files harder to read as they can contain many similar names, especially when the names of any nested elements are taken into consideration.

This syntax does have the advantage that it provides a unique name for each element. This is an advantage in itself, and also provides greater scope for extensions (see section 7.3.2, for example).

I suggest that, for simplification and clarity, the plan-writer adopts naming conventions. This is not something I have made much use of, but would do so were I to write any new plans. For example, the names of Drive Collection elements could be identical to the names of those components which they may trigger, but with a *dc_* prefix. The fact that under PyPOSH drive collection elements may not trigger action primitives directly (see section 8.3.3) meant that a large number of single-item action patterns were required. Conventions for naming these would also help. For my development, I often added the word "the" to the action pattern's name, so the pattern which executed `expire-damage-info` was called `expire-the-damage-info`, for example.

A further possible improvement regarding names would be checking for the existence of plan elements at load time rather than run-time. At present, references to non-existent

elements only become apparent when the plan interpreter attempts to call the element, and generates a "not callable" exception. Checking all elements at load-time could reduce development time, as such errors would be caught sooner. This is especially important in cases where setting up a situation to test the new element takes a substantial amount of time. For example, when I had to wait for the bot to find his way to the enemy base and pick up the flag before the new plan components for returning home could be tested. At present, such existence checks are done by the Lisp version of POSH, but not by PyPOSH.

Filenames

One of BOD's development recommendations is that old plan scripts should be kept. This allows rolling-back to previous versions, and provide a useful guide to the development history of a bot. I simply named my old files `bodbotattack[1].lap`, `bodbotattack[2].lap` and so on. One possible improvement for future development would be to use more descriptive names (e.g. `to base and take flag.lap`, `to base, take flag and return home.lap`, `flag capture and combat response.lap`). This would reduce the amount of time needed to find particular old plans. There would obviously be a trade-off between conciseness and clarity, but even slightly more descriptive names could be useful. Combining descriptive names with sequential numbering could be particularly beneficial, as this would also provide a clear sense of order and of the progress of development.

Magic Numbers

In the current structure of POSH plans the use of limits on frequency and retries, along with the timeouts for elements, has required much use of so-called "magic numbers", i.e. numbers hard-coded into the plans rather than specified as variables or constants.

The magic numbers problem is not only one of readability and maintainability but also constrains the plan writer's ability to express relationships between values. For example, the timeout of `expire-the-damage-info` is very low (8 seconds) since the Drive Collection element which calls it is written so as to occur very frequently (every 10 seconds). Allowing variables or constants along with simple operators would allow this relationship to be expressed: the frequency for the calling element could be set to some variable `damage-expire-frequency` and the timeout then to `damage-expire-frequency-2`, for example.

A relatively simple way to accomplish this would be with some form of pre-processing on the files similar to that which makes C's `#define` statements possible. This would have the added advantage that existing plan engines would not need altering. The pre-processed file would not need saving to disk (this would add clutter to plan directories) but could simply be stored in memory before being parsed as normal. A solution using variables would be more elegant, but would require existing plan engines to be modified.

7.3.2. Sharing and Redundancy

The structure of POSH plans could be modified to allow further sharing and to reduce redundancy. For example, consider the following competences, simplified versions of two from the `bodbotattack.lap` plan (points of particular interest are in bold):

```
(C get-to-enemy-base (minutes 10)
(goal ((at-enemy-base)))
  (elements
    (
```

```
        (check-immediate-vicinity
            (trigger((too-close-for-path)))
                big-rotate 2)
        (run-to-base (trigger((know-enemy-base-pos)))
            to-enemy-base)
        (find-base (trigger((reachable-nav-point)))
            walk-to-nav-point)
        (find-nav-point (trigger((succeed))) rotate 10)
        (wander-base (trigger((succeed)))
            wander-around)
      )
    )
  )

  (C go-to-own-base (minutes 10) (goal ((at-own-base)))
    (elements
      (
        (check-immediate-vicinity
            (trigger((too-close-for-path)))
                big-rotate 2)
        (run-to-own-base (trigger((know-own-base-pos)))
            to-own-base)
        (find-base (trigger((reachable-nav-point)))
            walk-to-nav-point)
        (find-nav-point (trigger((succeed))) rotate 10)
        (wander-base (trigger((succeed)))
            wander-around)
      )
    )
  )
```

These competences are almost identical, the only differences being the goal condition and the second element (i.e. which base it needs information about). Some method of sharing this structure would reduced redundancy.

One possibility would be allowing competence elements to be shared by using their name (something which is not used at all at present). This could result in the two competences above being rewritten to look something like the following:

```
(C get-to-enemy-base (minutes 10) (goal ((at-enemy-base)))
  (elements
    (
      (check-immediate-vicinity
          (trigger((too-close-for-path))) big-rotate 2)
      (run-to-base (trigger((know-enemy-base-pos)))
          to-enemy-base)
      (find-base (trigger((reachable-nav-point)))
          walk-to-nav-point)
      (find-nav-point (trigger((succeed))) rotate 10)
      (wander-base (trigger((succeed))) wander-around)
    )
  )
)

(C go-to-own-base (minutes 10) (goal ((at-own-base)))
```

```
  (elements
    (
      (check-immediate-vicinity)
      (run-to-own-base (trigger((know-own-base-pos)))
          to-own-base)
      (find-base)
      (find-nav-point)
      (wander-base)
    )
  )
)
```

Where `check-immediate-vicinity` etc refer to the competence elements defined in `get-to-enemy-base`.

This is in fact a further example of the plan vs behaviour trade-off (e.g. section 5.3.3). The problem above could also be solved by using a single competence and delegating to a behaviour the job of deciding which base the bot is currently interested in.

Priorities which vary with the value of some variable

A further area in which my plans contain redundancy is in responding to attack: The priority of responding to attack is higher when the bot's health is below a threshold value than when it is not. In the plan this is accomplished by having two similar drive elements, the higher-priority one including a check against the threshold. (In fact, the lower-priority one also checks that the bot is not focussed on attacking someone already, but this difference can be ignored for this part of the discussion).

Some way of removing this sort of redundancy could be useful, not least because it could make more precise differentiation easier (i.e. using a finer-grained check than just testing against a single threshold). One possibility would be to allow pre-processing on triggers of the following format:

```
([(variant-priority <threshold> <min-limit> <max-limit>)]
<sensor-name> <value> <predicate>)
```

Pre-processing on drive elements with a variant priority would result in extra drive elements being added. For example, consider a drive element Ψ with the following trigger:

```
((variant-priority 10 10 70) health-level 30 <)
```

Pre-processing of this would result in copies of that drive element being inserted into the plan, each with a slightly different trigger. For example, an element with the following trigger:

```
(health-level 20 <)
```

would be inserted two elements before Ψ (thus having a higher priority than it and than the element before it). An element whose trigger had a value of 10 would be inserted two elements before that. (The value decreases in steps of `<threshold>` down to a minimum of `<min-limit>`). Similarly, elements with lower priorities would be inserted which tested against a higher health level (up to `<max-limit>`).

However, I believe that while such preprocessing would be interesting, it would only be useful or applicable in a limited number of cases. It would also risk making the bot's behaviour hard to understand, as the version of the plan the bot actually used would have many more elements than that which the programmer could see! There are undoubtedly

alternative solutions to this problem which are more elegant, but I do not have time to consider these here. The solution proposed above may be considered a proof-of-concept.

Further possibilities could be developed to enable the sharing of more of the structure. This would risk decreasing the plan's clarity, however.

7.3.3. Parallel Plan Elements

Consider the following description of the structure of Drive Collection elements (from Bryson, 2001; p. 226):

```
drive-elements :: (drives (<drive-element>+)+)
```

The drive elements in parentheses are ordered in terms of priority. However, the inner + indicates that each set of parentheses may contain more than one drive element. In this case, these elements are taken to have equal priority and should therefore have mutually exclusive triggers (behaviour if they do not is undefined). Similarly, competence elements may also be in parallel.

The astute reader will notice that this does not in fact add any expressive power to the plans: if items have mutually exclusive triggers, then allowing (either) one to be one place above the other in the priority list would result in the same action selections as would occur if they were at the same priority level.

However, despite this, and despite the fact that I did not use this functionality, I believe it is an important part of POSH which should not be removed. The reason for this is the subtle difference in meaning which the alternative structure conveys, thus contributing to the self-documenting nature of the code:

- Two items having the same priority level suggests that neither is more "urgent" than the other. (I.e. neither requires the bot's attention before the other).
- Two items having different priorities suggests that one is more urgent than the other. The fact that the triggers are mutually exclusive is *irrelevant* at this level of meaning.

The only downside I can envisage from keeping this distinction (apart from the fact that the majority of Drive Collection elements will be surrounded by double parentheses) is a slight increase in maintenance: the programmer must ensure that the triggers remain mutually exclusive even when they are altered (either by adding or removing triggers, or by altering the underlying senses). However, I believe that this is not a sufficient reason to remove the possibility for parallel elements, especially as the fact that this functionality is used infrequently means that the extra maintenance would be minimal.

7.3.4. Improvements to Documentation

One useful improvement would be the documentation of possible trigger predicates. The PyPOSH documentation does state that "triggers can contain predicates and values in addition to sensor specified. This allows the output form the trigger function to be compared against a value with a predicate." (Kwong, 2003; p. 34) and mentions that the default predicate is the test for equality. However, the other possible predicates are not mentioned. The PyPOSH code, and my own experience, indicate that the following are usable:

Values	Meaning
`eq`, `==`	sense's value equals test value

`lt, <`	sense's value is less than test value
`gt, >`	sense's value is greater than test value
`not, !, !=`	sense's value is different to test value

Table 7.1 Possible Predicate Values for PyPOSH

Plans for the Lisp version of POSH may use any valid Lisp comparator. However, this is also not made clear by the documentation.

7.4. Testing POSH against Action-Selection Criteria

Tyrrell (1993) examines a large number of action selection mechanisms from both a practical perspective (controlling a creature in a simulated environment) and a theoretical one. As part of his report, he outlines fourteen requirements which he claims such a mechanism should meet. In this section, I introduce a small number of these criteria and briefly explore how well POSH action-selection meets them. (The list of criteria adapted from pp. 214-216 of *ibid*). For brevity, only those criteria which are most interesting with relation to this discussion are included, but I claim that those left out are met by POSH.

7.4.1. Dealing with all types of sub-problem

For brevity, I will not explore all Tyrrell's sub-problem types here (I believe that POSH can handle all these problem types, incidentally). Interested readers are directed to section 7.2 of *ibid*. However, a number of problem types are worth discussion:

Homeostatic sub-problems concern internal variables which should be kept at a certain value or within a certain range. This has similarities to goal-driven behaviour, discussed in section 6.6.5. A goal- rather than event-driven agent would arguably provide a closer fit to ths sort of problem. However, there is no reason why a BOD agent could not handle such problems. In fact, the bodbot does deal with such a problem, that of keeping its health sufficiently high. For example, the priority of responding to attack is different depending on the bot's health level. Similarly, the bot will only pick up a medical kit if his health is low, but getting it will become a high priority in this case.

Periodic problems are those whose importance rises and falls with a regular rhythm (e.g. sleeping). A naïve approach might be to attempt this sort of control using the frequency attribute of drive collection elements although, as section 5.3.6 suggests, this would not be very effective. However, alternative solutions would be possible, using comparison predicates in triggers, for example. (I.e. "if time of day greater than x and less than y ...".) Section 7.3.2 outlines a potentially more optimal approach to this sort of problem in its discussion of priorities which vary with the value of some variable.

Proscriptive problems require that certain actions *not* be carried out (e.g. that the agent avoids walking into a lake of lava). There is no overarching mechanism in BOD which prevents actions from being executed. However, such things could arguably be achieved using POSH's hierarchical structure. For example, consider the problem that an agent should not shoot when a wall is in front of him, even if he is "facing" the enemy (i.e. the enemy is on the other side of the wall). Initially the component triggered could just be an action primitive which communicates with Gamebots. A more complex version of the plan could easily replace this with a reference to a competence which first checks that the agent

is not facing a wall and then shoots if this condition is met[8]. In short, plans and triggers can be designed such that actions will not run in certain cases.

7.4.2. Contiguous action sequences

"Contiguous action sequences" refers to the idea that once an agent has started on one sequence of actions, the likelihood of changing to another sequence should be reduced. Tyrrell claims that this is because of the cost of changing (i.e. of moving to a situation where some other goal can be successfully achieved). For example, once an animal has started drinking, it should be less inclined to go after nearby food as it would be "wasting" time in getting to the food which could be used for the guaranteed reward of continued drinking. Rather, it should wait until the need for food is higher than that which might normally trigger a response, or until the need for water is particularly low.

There are few parallels to this in Unreal Tournament (medical kits are picked up as soon as they are touched and provide instant health improvement, for example). However, one situation which is relatively similar is that of a defender finding a well-secluded vantage point: once it is found, the bot should be reluctant to abandon it in case an enemy approaches while s/he is "in the open" and will have less of an advantage in terms of location.

POSH does not provide this emphasis on existing actions (not doing so arguably helps it be more reactive), although the slip-stack mechanism (see section 2.4.5) is similar in some respects. However, I believe that the number of occasions in which such emphasis is relevant is sufficiently small that such behaviour can be accomplished with triggers and Action Patterns. Furthermore, I believe that a deep-rooted emphasis on contiguous action sequences is too broad an approach: although achieving such effects with triggers is more complicated, it is also more precise.

7.4.3. Compromise candidates

Tyrrell (1993; p. 216) describes the idea of compromise candidates as "the need to be able to choose actions that, while not the best choice for any one sub-problem alone, are best when all sub-problems are considered simultaneously". An example from the UT domain could be an unarmed bot taking a less direct route to the enemy's base as it increases his chances of finding a weapon on the way, or running away from an assailant and towards a medical kit rather than just taking the route which will put the most distance between him and his attacker. This sort of thing is far more complicated in UT than in Tyrrell's simulation (for example, Tyrrell's simulation takes place in a discrete grid with discrete timing). The selection of compromise candidates is still possible in theory as any part of the plan may include triggers used in any other part of the plan, and primitives may examine any of the bot's state which they need. However, in practice the selection of compromise candidates would run into problems:

- Considering all, or even most, of the other candidates would result in long and unwieldy triggers.
- Providing plan elements capable of producing a number of possible compromise candidates would result in unreasonably long competences.

---

[8] In reality, this problem would not arise as the bot is able to check whether other players are visible. However, I chose it as it provides a useful, simple example.

Although extending the action selection mechanism with an ability to choose compromise candidates could possibly help with these problems, I believe that the primary difficulty is the complexity of the domain. This means that enabling *any* action selection mechanism to choose compromise actions would be immensely difficult (not least because it would require the mechanism to have a huge amount of knowledge about the domain, violating encapsulation rules).

These problems could be avoided by producing more complex behaviours which examined state concerning other actions as well as that concerning the action which they wished to perform. Such information could allow the final action produced to be chosen with other candidates in mind. This would be possible (and could be very effective in some cases) but is not a general-purpose solution.

Overall, I believe that it is unreasonable to expect a general-purpose compromise-finder for a domain as complicated as Unreal Tournament.

Transitions

A loosely-related area to that of interactions between behaviours to produce a compromised outcome is that of *transitions*, an idea introduced by Sengers (1998). Sengers argues that current agent architectures result in agents which are "schizophrenic" (p. 37). This term is used to describe problems which occur as a result of attempting to reintegrate atomized agent behaviours. For example:

- Agents attempting to undertake two incompatible behaviours simultaneously (e.g. fighting and sleeping).
- Agents switching between behaviours in a way which appears unnatural.
- Agents rapidly switching between behaviours rather than actually achieving anything, or refusing to change behaviour when circumstances required it.
- Agents rapidly switching back and forth between two behaviours.

(These examples adapted from *ibid*, pp. 38-39.) The primary way in which Sengers suggests such problems be combatted is via a change in thinking during design: considering how an agent *appears to an observer* should be central to agent design. This is an interesting and novel approach which time unfortunately does not permit discussion of here. (Interested readers are encouraged to read Sengers' thesis.) However, one of the concrete examples of this principle is the idea of transitions: actions which occur during the switch between two behaviours.

As with compromise candidates, the POSH action-selection mechanism does not contain sufficient meta-level controls to implement such transitions at a high-level. Rather, they would need to be coded either in terms of more complex triggers and longer competences, or at the level of primitives themselves. Allowing more control over the meta-level procedures of action selection could usefully make POSH more extensible in this area and others, but would risk encouraging dilution and abuse of the mechanism. Alternatively, transitions could potentially be accomplished by means similar to those used by Bryson and Thórisson (2000). They combined POSH action selection with the Ymir architecture (Thórisson, 1996) to link high-level action selection with lower-level control over actions such as gestures and body language. Even without such methods, however, I believe that POSH has other means of addressing all but the second of Sengers' concerns listed above.

7.4.4. Conclusion

In conclusion, I believe that all of Tyrrell's criteria are met to some extent by POSH. POSH could deal better with periodic goals but this is simply a matter of conciseness – triggers can be designed which are sufficiently powerful for this.

POSH is not able to choose compromise candidates. However, I believe that for a domain as complex as Unreal Tournament this is an unrealistic requirement and, furthermore, that sufficient "compromised" choices can be made by behaviours in the specific cases where they are applicable. Similarly, POSH does not have built-in support for contiguous action sequences (although keeping items on the scheduler is arguably similar) but I again believe that this is needed infrequently enough that accomplishing it with triggers is a reasonable response.

8. PyPOSH: Problems, Alterations, Corrections and Recommendations

During the implementation stage of my project, I made great use of Kwong's (2003) PyPOSH framework (see section 2.6). In doing so, I discovered and corrected a number of problems with it. I also made some changes to the framework to make it more powerful and better suited to my needs. These corrections and changes are discussed in some detail in this section to facilitate understanding of the code and to assist anyone who wishes to compare the original version with the modified one. Furthermore, a number of problems I encountered but was not able to correct are also discussed.

8.1. Multiple Behaviour Files

The original version of PyPOSH only allowed action and sensory primitives to be specified in a single file. This is inconsistent with POSH's principle of decomposition of functionality into behaviour modules (see section 2.3) and makes effective development more difficult.

I therefore modified PyPOSH so that behaviours could be specified as multiple files. This involved a number of changes:

- Dictionaries for actions and senses were added back into `posh_agent.py` (lines 89 & 90), having previously been commented out. This provided a central reference point for the primitives, despite the fact that they were now in multiple files.
- The functions `get_act` and `get_sense` (again in `posh_agent.py`) were also modified. The original versions called their equivalent function in the single behaviour object to retrieve the relevant primitive. The new versions looked in the new dictionaries instead.
- Similarly, the functions `add_act` and `add_sense` now add details of primitives to the central dictionaries rather than passing them on to the behaviour object.
- The `make_behavior` function of the main agent file (in my case, `bodbot.py`) was originally defined to return a single behaviour object. I modified it to return a list of behaviour objects. This meant that `self.behavior_instance` in `posh_agent.py` then needed initialising to be an empty list rather than `None` (Python's null type), and various other pieces of code which used it needed slight modification.

8.1.1. How to Write a Behaviour Module

As a means of further illustrating the multiple-behaviour-file modification to PyPOSH, this section explains the key steps involved in writing a behaviour module for an agent.

Creating the Class

1. Create a new class definition, extending the class `Base`.
2. The `__init__` function should call the ancestor's initialisation function (`Base.__init__(self, **kw)`) and should set up any state. It should also call the functions you have written for registering primitives (see below).

3. Include the bind_bot function:

```
def bind_bot(self, bot):
        self.bot = bot
```

4. Write functions for your action and sensory primitives. Note that actions must return True to indicate successful completion and False otherwise. (See the examples in section 5.3.3.)

5. Write functions to register your primitives. These should call self.agent.add_act for each action and self.agent.add_sense for each sense. These functions take two arguments: a string which will be used to refer to the primitive in the plan, and a reference to the relevant function.

Initialising the Class

An instance of the behaviour class will need to be created by the make_behavior function in your top-level agent file (in my case, bodbot.py). This function should create a class instance, passing it any relevant parameters. One required parameter is the agent. This should be passed as a keyword parameter (i.e. agent = agent). agent is one of make_behaviour's parameters, so there is already a reference to it.

Once this is done, the instance's bind_bot method should be called, passing it an instance of Bot_Agent which in turn should be created in as part of make_behaviour. All behaviour instances use the same bot. The connect function of this bot needs calling as part of make_behaviour, but again only needs doing once, regardless of how many behaviour classes are instantiated.

Finally, the new behaviour instance should be appended to the behaviour list which make_behaviour returns.

8.2. Problems Corrected

I discovered two quite important bugs in the PyPOSH code. One affected timeouts for Drive Collection elements and the other retry limits for competence elements.

8.2.1. Timeouts for Drive Collection Elements

The code to test the frequency (lines 922 and 923 in posh_agent.py) originally contained the following:

```
if (element.frequency <= 0) or \
(element.frequency > (time - element.last_fired)):
```

However, time is a Python module and thus cannot be subtracted from. I replaced time with timestamp, a variable which had previously ben instantiated (in the original version of the code) to hold the number of seconds since the UNIX epoch (line 912).

This modification meant that the code did not crash. However, the element with the frequency limit was then never called. The reason for this was the greater-than operator in the test (element.frequency > (timestamp - element.last_fired)), which should instead have been less-than. Fixing this solved the problem.

8.2.2. Retry Limits for Competence Elements

Attempting to use a retry limit for a competence element resulted in the following error:

```
  File "C:\programming\Python\programs\fyp\pyposh\
posh_core.py", line 701, in fire_cel

    competence_element.retries =
competence_element.retries - 1

TypeError: unsupported operand type(s) for -: 'str'
and 'int'
```

The reason for this problem could be found in `posh_agent.py` at line 624:

```
retries = elements[3]
```

The variable `elements` contains strings parsed from the plan file. Therefore, the line needed changing to the following:

```
retries = int(elements[3])
```

Doing this solved the problem.

8.3. Other Issues and Suggestions

8.3.1. Debugging

I was unable to use the Python debugger on PyPOSH. This may have been due to my inexperience with Python, but I cannot be sure. This section illustrates the problem encountered.

The debugger is run using the `run` function of the `pdb` object. Thus my call to the debugger was as follows:

```
pdb.run(pyposh)
```

`pyposh` is the file which is run to launch the PyPOSH environment and interpreter. Note that the file as a whole is run, rather than a specific function from it. I believe that this may be the reason for the error I encountered:

```
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in ?
    pdb.run(pyposh)
  File "C:\programming\Python\lib\pdb.py", line 979,
in run
    Pdb().run(statement, globals, locals)
  File "C:\programming\Python\lib\bdb.py", line 347,
in run
    cmd = cmd+'\n'
TypeError: unsupported operand type(s) for +:
'module' and 'str'
```

However, I cannot be certain of the reason for this error and so it is an area for future work (see section 9.2).

8.3.2. Profiling

I was also unable to use the profiler successfully on PyPOSH. I set it up to profile the calling of `app.MainLoop()` in `pyposh.py`, but this seemed only to record details of the

execution of functionality concerned with the PyPOSH GUI. I believe that this may be because code is loaded dynamically when agents are run, but again I cannot be sure.

In an attempt to combat this, I tried to profile the code which is executed when the agent is "brought to life", by using the following command:

```
profile.run('self.agent.execute_thread()',
'profile.out')
```

However, the profiler uses the `exec` statement to run code dynamically, and that cannot understand the reference to `self` as `self` is a local variable. Therefore, this was also unsuccessful (it did not even run).

8.3.3. Issues with Primitives

PyPOSH does not allow Drive Collection elements to call action primitives directly, but requires them to be contained within single-element Action Patterns instead. This is not a restriction derived from the original POSH specification but rather is an extra constraint. I have not corrected this myself, but would suggest that it is considered in any further development of PyPOSH.

8.4. Distribution

A number of other projects are making use of Unreal Tournament and POSH / PyPOSH and for this reason I was asked to create a distribution containing the current version of my bot and my modified PyPOSH files. This has been circulated among a number of researchers and academics in both Europe and North America. The readme file I produced for this release is given in Appendix D.

9. Conclusions

This chapter begins by summarising the achievements of this document. It then highlights the document's limitations, and gives suggestions for future work. Finally, a summary of my evaluation of BOD is provided.

9.1. Summary of Achievements

9.1.1. A Discussion of Architectures

This dissertation has summarised Bryson's (2001) Behaviour Oriented Design [BOD]. The key features, both methodological and architectural (the POSH architecture) have been introduced, explored and discussed. A number of other architectures for Artificial Intelligence and Agent Development have also been introduced, discussed and compared against. These architectures have included Soar (Lehman et al. 1996), EPIC (Kieras and Meyer, 1997), ACT-R (ACT-R Research Group, 2004a), the Subsumption Architecture (Brooks, 1986), JACK (Howden et al., 2001), agent modelling via UML and Rational Rose (section 6.2), and Kinny et al.'s (1996) Beliefs-Desires-Intentions model

9.1.2. The Bodbot Project

This dissertation has explored the bodbot project, which includes the most complex POSH plan file ever published (and equally, a large number of behaviour modules and primitives). The development of this project has been discussed, and BOD's part in this process analysed. The dissertation has demonstrated the resulting behaviour of the created bot, further illustrating the workings of POSH and the development undertaken for the bodbot project.

9.1.3. An Evaluation of BOD, and Improvements to PyPOSH

BOD has been evaluated both in its own right and comparatively in light of significant research into existing architectures and methodologies. Suggestions for improvements have ben identified, as have BOD's particular strengths. The PyPOSH Python implementation of POSH (Kwong, 2003) has been discussed and further developed, improved and corrected.

9.2. Future Work, and Limitations of this Dissertation

Whilst this dissertation has accomplished a lot, there is still much which could be improved and covered in more detail, and many possibilities for further work. This section introduces a number of the possibilities for future work.

Improvements to the bodbot fall into two categories: significant developments and tweaks. Both are important to create a better CTF player.

9.2.1. Significant Developments

Meeting the Specification

Whilst the developed bot provided many opportunities for analysis and evaluation, and proved an interesting (although not particularly challenging) opponent, it requires further development to meet the original specification as outlined in Appendix B.

The most significant lacking is that the plan (and associated primitives) for a defender have not been created. Creating a defender agent would be useful in a number of ways:

- It would highlight general bot-development challenges and problems not demonstrated by the attacker.
- It would make the development of teams and bot-vs-bot simulations (see below) more feasible.
- It would be evidence of the usefulness of the fact that two different bots (i.e. attacker and defender) can be created from one set of primitives.

There are also a number of other ways in which the bot needs to be developed to meet the provided specification. Specifically, the following high-level goals and drives require attention:

- Avoid incoming projectile
- Pick up nearby ammunition (if have none)
- Attack enemy who is near our flag
- Find medical kit, weapon or ammunition (as required).

The drive "Avoid incoming projectile" would be particularly interesting, as that would require the new version of the Gamebots interface to be used (see section 5.3.4).

There are also a number of other minor discrepancies between the created bot and the specification which, for brevity, I will not list here.

Teams and Bot-vs-Bot Challenges

BOD's suitability for Multi-Agent Systems [MAS] has not been tested in great detail (but see Bryson, 2003). Using the plan created (ideally in conjunction with a "defender" plan) to control teams of bots could be very enlightening, not least in the possibilities it would open up for communication between agents. Similarly, observing bodbots playing against each other would be very interesting and provide a forum for testing the bot in novel ways.

Internal Maps

I believe that the bodbot could be far more effective were it to create an internal map as part of its state. Among other things, this would address the problem that Gamebots provides no stimulus to represent "seeing" a wall – the bot only knows about walls when it walks or is pushed into them. An annotated map of the type described by Laird (2005; see Appendix D) would be especially useful in the creation of a tactical bot. Potential annotations could include:

- Paths from and to various useful points, even including routes where there are no navpoints.
- Key doorways and tunnels, to help defenders identify vantage points.
- Points where weapons, ammunition and health kits appear.
- Common routes taken by opponents, and common hiding places and vantage points used by the other team.
- Routes worth avoiding (e.g. narrow tunnels, exposed areas or places where the risk of long falls or falling into lava is particularly high).
- Vantage points used by the defenders on the bot's own team, to help it identify how likely an assailant is to escape and so the priority of chasing him/her.

The above are just a few examples, experience and testing would probably reveal many more.

9.2.2. Tweaks

Although listed as a "tweak" as it would be relatively quick to fix, the fact that some **updating functions still run in parallel with actions** (thus causing exceptions and errors, see section 5.3.5) is quite a major problem which should be corrected as a priority.

With some important exceptions (e.g. navpoints), the current bot does not make much use of the **"reachable" attribute** sent when it sees objects. The reason for this has been that it often seems to be set to false when it should be true. However, this should be tested more extensively, and the information made use of. For example, when the bot decides whether to run towards a flag on the ground. Like other perception problems, this "reachable problem" could be due to the skill setting of the bot. However, I have not had time even for preliminary tests of this hypothesis.

The bot does not seem very good at **noticing enemies** carrying its flag (see section 4.2.2). This problem should be examined in more detail. In particular, the theory that the UT skill level affects this (section 4.2.2 again) should be tested.

The theory that a higher skill level improves the bot's ability to **notice navigation points** should also be tested, and modifications made to the behaviour as required to maker the most of the improved perception. (See section 5.3.4).

The user should be able to **specify the team** that the bot is on. This would involve modification of the parameters to `make_behaviour` (in `bodbot.py`). As with responding to incoming projectiles (see above), this would involve use of the new version of Gamebots.

The **list of visited navigation-points** which the bot maintains should instead be a dictionary with navpoints' locations as its keys. This should make calculating the least-visited navpoint (see section 5.3.3) a far less expensive process.

As discussed in section 4.2.3, there are occasions when the bot keeps looking at an attacker whilst continuing to move around the level, but **fails to keep shooting** at him / her. This sort of problem should be fixed, perhaps by resending the `SHOOT` message if it has not been already sent more than once.

The problem with **corrupt Gamebots data** demonstrated in section 5.3.4 could be examined. First, the new version of Gamebots could be tested to see if the problem still occurs. If it does, the code could probably be improved to detect the corrupt data (as the corruption results in a string being present where an integer is expected) and the response ignored.

9.2.3. Further Evaluation and Development of BOD

The evaluation of BOD performed as part of this document is a good start, but there is still much more which could be done. For example, more comparisons with other architectures could be carried out. Comparisons informed by experimentation would be particularly valuable (i.e. attempting to develop the same bot in two different architectures) as that is something missing from this document.

I concentrated primarily on evaluating the methodological parts of BOD by experience and by comparison with formal methodologies. It could be informative to analyse the

development processes of some agent-development projects which did *not* use a formal methodology.

This document has made a number of suggestions for improvements to the sharing of plan elements both within and across plans (e.g. section 7.3.2). These improvements should be implemented and tested. Implementing these by pre-processing on plans would mean that the existing POSH interpreters would not need modification.

Chapter 8 details improvements to and problems with the PyPOSH version of POSH. Similar analysis of other implementations of POSH (e.g. the Lisp version) would be useful, not least in informing the development of new implementations[9].

9.2.4. Other Work

As discussed in sections 8.3.1 and 8.3.2, I was unable to use the Python debugger or profiler with PyPOSH. Further attempts to get these tools to work successfully with PyPOSH would be very useful. For instance, I was unable to determine precisely why the bodbot spent so long doing nothing at the start of some runs; this sort of problem would be easier to identify with a working profiler and debugger.

The BOD methodology recommends the use of profiling, debuggers and version control software. The fact that I did not use any of these is a weakness of my evaluation. (Version control software was not used as I determined the learning curve to be too steep for the perceived benefits. I reached this conclusion after considering the quality and quantity of documentation available, and the fact that I already had previously devised manual version control procedures.)

9.3. Summary of Evaluation

BOD has been evaluated in light of both research and experience, and a number of key points have been identified in terms of both methodology and architecture.

9.3.1. Methodology

The detail BOD provides about the development process is definitely useful, and an improvement on that offered by most other architectures studied.

In terms of the initial stages of development, BOD's emphasis on what is done rather than what is known is very useful. However, the process BOD identifies could be improved in many cases by a slight re-ordering of the steps of the analysis (section 6.3). With regard to the main development process, BOD greatly facilitates evolutionary, incremental development and enables existing elements to be changed easily. By comparison, architectures such as Soar, ACT-R and the Subsumption Architecture bring about many complex dependencies between elements, making evolutionary development more difficult. The lessons learned from projects using Extreme Programming (section 7.2.1) suggest that BOD is right to emphasis iterative development and revision of specifications.

9.3.2. POSH Action Selection

The fact that BOD provides an action selection mechanism at all is a positive step. A number of approaches require manual coding of action selection. POSH's hierarchical nature and the ability to set limits on frequency and retries is also useful, whilst the simpler

---

[9] For example, work is currently underway on a Java implementation of POSH.

relationships between elements (see previous section) make debugging easier. There are cases where the ability to consider utility (Taatgen et al., in press) would be useful when choosing between elements, however (section 6.6.2). While many approaches are much more goal-directed than POSH, I claim that this reliance on goals can in fact be an unnecessary and even confusing abstraction (section 6.6.5). Looking at Tyrrell's (1993) criteria for Action Selection Mechanisms, I claim that POSH meets all of them to some extent (section 7.4).

One area in which POSH could be improved is that of robustness, e.g. how the system copes with the failure of a plan element or primitive. This weakness becomes apparent when comparing with the Subsumption Architecture (section 6.8.2).

9.3.3. Plan Files

The ability use different plans to generate radically different agents from one set of behaviour modules gives BOD much power. However, the ability to share elements across plan files could be a useful simplification and is not currently supported. Further sharing *within* plan files could also be useful, although the trade-off between redundancy and readability is very important here (e.g. section 7.3.2). The development of POSH plans can be made easier by the adoption of naming conventions (section 7.3.1).

9.3.4. Overall Summary

Overall, I believe that BOD stands up well to scrutiny: whilst there are areas for improvement, these do not detract significantly from the fact that this is a very useful and powerful methodology, being both scalable and applicable to a wide range of situations.

Bibliography

ACT-R Research Group, 2004a. *ACT-R: Theory and Architecture of Cognition* [online]. Pittsburgh, USA: Department of Psychology, Carnegie Mellon University. Available from *http://act-r.psy.cmu.edu/* [Accessed 17 November 2004].

ACT-R Research Group, 2004b. *ACT-R Tutorial* [online]. Pittsburgh, USA: Department of Psychology, Carnegie Mellon University. Available from *http://act-r.psy.cmu.edu/tutorials/* [Accessed 17 November 2004].

Anderson, J.R., Matessa, M., Lebiere, C., 1997. ACT-R: A Theory of Higher Level Cognition and Its Relation to Visual Attention. *Human-Computer Interaction*, 12 (4), pp. 439 - 462.

Barber, K.S. and Martin, C.E., 1999. Agent Autonomy: Specification, Measurement, and Dynamic Adjustment. *Proceedings of the Autonomy Control Software Workshop at Autonomous Agents 1999 (Agents '99)*, 1 May 1999, Seattle, Washington. University of Texas at Austin, pp. 8 - 15.

Beck, K., 1999. Embracing Change with Extreme Programming. *IEEE Computer*, 32 (10), pp. 70 - 77.

Booch, G., 1990. *Object oriented design with applications*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc..

Bossi, P., 2003. eXtreme Programming applied: a case in the private banking domain. *Proceedings of the OOP2003 (Object-Oriented Programming) Conference*, 20-24 January 2003, Munich, Germany. *Publisher unknown*, *page numbers unknown*.

Bovair, S., Kieras, D.E., Polson, P.G., 1990. The Acquisition and Performance of Text-Editing Skill: A Cognitive Complexity Analysis. *Human-Computer Interaction*, 5 (1), pp. 1 - 48.

Brooks, R.A., 1991. Intelligence without Representation. *In* Brooks, R.A.. *Cambrian Intelligence: the early history of the new AI.* Massachusetts, USA: The MIT Press, pp. 79 - 101.

Brooks, R.A., 1986. A Robust Layered Control System for a Mobile Robot. *In* Brooks, R.A.. *Cambrian Intelligence: the early history of the new AI.* Massachusetts, USA: The MIT Press, pp. 3 - 26.

Bryson, J.J., 2001. *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. Thesis (PhD). Massachusetts Institute of Technology.

Bryson, J.J., 2000a. *The Study of Sequential and Hierarchical Organisation of Behaviour via Artificial Mechanisms of Action Selection*. Dissertation (MPhil). University of Edinburgh.

Bryson, J.J. and Thórisson K., R., 2000. Dragons, Bats & Evil Knights: A Three-Layer Design Approach to Character Based Creative Play. *Virtual Reality*, 5 (2), pp. 57 - 71.

Bryson, J.J., 2000b. Cross-paradigm analysis of autonomous agent architecture. *Journal of Experimental & Theoretical Artificial Intelligence*, 12 (2), pp. 165 - 190.

Bryson, J.J., 2000c. Hierarchy and Sequence vs. Full Parallelism in Reactive Action Selection Architectures. *Proceedings of the Sixth International Conference on the Simulation of Adaptive Behavior (SAB2000)*, 11 - 15 September 2000, Paris, France. MIT Press, pp. 147 - 156.

Bryson, J.J., 2003. Where Should Complexity Go? Cooperation in Complex Agents with Minimal Communication. *Proceedings of the First GSFC/JPL Workshop on Radical Agent Concepts (WRAC)*, 27-29 September 2005, NASA Goddard Space Flight Center Visitor's Center, Greenbelt, MD USA. Springer, pp. 298 - 313.

Carnegie Mellon University *15-396: Agent 2 - a step by step tutorial* [online]. Carnegie Mellon University, USA: Department of Psychology, Carnegie Mellon University. Available from *http://www.andrew.cmu.edu/user/roman/15396/agent2.html* [Accessed 22 March 2005].

Davids, A, 1997. *Python: Yet Another Object Oriented Interpretive Scripting Language* [online]. Australia. Available from *http://www.metva.com.au/av_paper_python.html* [Accessed 2 November 2004].

Douglas, S. *1* [online]. Carnegie Mellon University, USA: Department of Psychology, Carnegie Mellon University. Available from *http://act-r.psy.cmu.edu/~douglass/Douglass/Agents/1/Tutorial/Agent-01.htm* [Accessed 22 March 2005].

Epic Games, 2004. *Unreal Tournament* [online]. Available from *http://www.unrealtournament.com/utgoty/* [Accessed 2 November 2004].

Howden, N., Ronnquist, R., Hodgson, A., and Lucas, A, 2001. Jack Intelligent Agents™ – Summary of an Agent Infrastructure. *5th International Conference on Autonomous Agents*, May 28 - June 1, 2001, Montreal, Canada. *Publisher unknown*, *page numbers unknown*.

Inglesias, C.A., Garijo, M., González, J.C., 1999. A Survey of Agent-Oriented Methodologies. *Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages*, July, 1998, Paris, France. Springer-Verlag, pp. 317 - 330.

Ingrand, F.F., Chatila, R., Alami, R., Robert, F., 1996. PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. *Proceedings of the IEEE International Conference on Robotics and Automation*, 22-28 April 1996, St Paul, Minnesota, USA. *publisher unknown*, pp. 43 - 49.

Kaminka, G.A., Veloso, M.M., Schaffer, S., Sollitto, C., Adobbati, R., Marshall, A.N., Scholer, A., Tejada, S., 2002. GameBots: a flexible test bed for multiagent team research. *Communications of the ACM*, 45 (1), pp. 43 - 45.

Karlström, D., 2002. Introducing Extreme Programming – An Experience Report. *XP 2002*, . Available from *http://www.agilealliance.org/articles/articles/DanielKarlstrom--IntroducingExtremeProgramming.pdf* [Accessed 7 March 2005]

Keefer, G., 2002. Extreme Programming Considered Harmful for Reliable Software Development. *AVOCA [Advanced Visioning of Components and Architectures] Technical Report*, . Available from *http://www.agilealliance.org/articles/articles/XPConsideredHarmful-GeraldKeefer.pdf* [Accessed 7 March 2005]

Kieras, D.E. and Meyer, D.E., 1997. An Overview of the EPIC Architecture for Cognition and Performance with Application to Human-Computer Interaction. *Human-Computer Interaction*, 12 (4), pp. 391 - 438.

Kinny, D., Georgeff, M., Roa, A., 1996. A methodology and modelling technique for systems of BDI agents. *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world*, 22 - 25 January 1996, Institute for Perception Research, Eindhoven - The Netherlands. Springer-Verlag New York, Inc., pp. 56 - 71.

Kwong, A, 2003. *A Framework for Reactive Intelligence through Agile Component-Based Behaviors*. Dissertation (MSc). University of Bath.

Laird, J., (laird AT umich.edu), 9 March 2005. *RE: Soar Quakebot Development Process? (Research for Dissertation)*. E-mail to S.J. Partington (sam AT samsolutions.co.uk).

Laird, J.E. and Duchi, J.C., 2000. Creating Human-like Synthetic Characters with Multiple Skill Levels: A Case Study using the Soar Quakebot. *Papers from the AAAI Fall Symposium: Simulating Human Agents*, November 3 - 5 2000, North Falmouth, Massachusetts. The American Association for Artificial Intelligence, *page numbers unknown*.

Laird, J.E. and van Lent, M, 2000. Human-level AI's Killer Application: Interactive Computer Games. *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, July 30 - August 3, 2000, Austin, Texas. The American Association for Artificial Intelligence, pp. 1171 - 1178.

Lappo, P., 2002. No Pain, No XP: Observations on Teaching and Mentoring Extreme Programming to University Students. *XP 2002*, . Available from *http://www.agilealliance.org/articles/articles/PeterLappo--ObservationsonTeachingandMentoringXP.pdf* [Accessed 7 March 2005]

Lehman, J.F., Laird, J., Rosenbloom, P., 1996. *A Gentle Introduction to Soar, an Architecture for Human Cognition*. United States of America: University of Southern California. (Technical Report A096413).

Maes, P., 1991. A bottom-up mechanism for behaviour selection in an artificial creature. *In* Meyer, J.-A. and Wilson, S., (Eds.). *From Animals to Animats*. Cambridge, MA: MIT Press, pp. 238 - 246.

Matarić, M.J., 1997. Behaviour-based control: Examples from navigation, learning, and group behaviour. *Journal of Experimental and Theoretical Artificial Intelligence*, 9 (2/3), pp. 323 - 336.

McCarthy, J., and Hayes, P. J., 1969. Some philosophical problems from the standpoint of artificial intelligence. *In* B. Meltzer and D. Michie (Eds.). *Machine Intelligence 4*. Edinburgh: Edinburgh University Press, pp. 463 - 502.

McCarthy, J., 1998. *Partial Formalizations and the Lemmings Game* [online]. United States of America: Stanford University. Available from *http://www-formal.stanford.edu/jmc/lemmings/lemmings.html* [Accessed 2 November 2004].

Mitchell, J.C., 2003. *Concepts in Programming Languages*. Cambridge: Cambridge University Press.

Müller, M.M. and Tichy, W.F., 2001. Case study: extreme programming in a university environment. *In* IEEE Computer Society. *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*. Toronto, Ontario, Canada: IEEE Computer Society, pp. 537 - 544.

Nilsson, N.J., 1998. *Artificial Intelligence: A New Synthesis*. San Francisco: Morgan Kaufmann Publishers, Inc.

Odell, J.J., 1998. *Advanced Object-Oriented Analysis & Design Using UML*. Cambridge, UK: Press Syndicate of the University of Cambridge and SIGS Books.

Rumpe, B. and Schröder, A., 2002. Quantitative Survey on Extreme Programming Projects. *XP 2002*. Available from *http://www.agilealliance.org/articles/articles/QuantitativeSurvey.pdf* [Accessed 7 March 2005]

Sengers, P., 1998. *Anti-Boxology: Agent Design in Cultural Context*. Thesis (PhD). Carnegie Mellon University.

Stone, P. and McAllester D., 2001. An architecture for action selection in robotic soccer. *Proceedings of the fifth international conference on Autonomous agents*, 2001, Montreal, Quebec, Canada. ACM Press, pp. 316 - 323.

Taatgen, N.A., Lebiere, C. & Anderson, J.R., (in press, this draft 2004). Modeling paradigms in ACT-R. *In* Sun, R. (ed.). *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*. New York: Cambridge University Press, Chapter 1.

Taatgen, N.A. and Anderson, J.R., 2002. Why do children learn to say "broke"? A model of learning the past tense without feedback.. *Cognition*, 86 (2), pp. 123 - 155.

Thórisson, K., R., 1996. *Communicative Humanoids: A Computational Model of Psychosocial Dialogue Skills*. Thesis (PhD). MIT Media Laboratory.

Tyrrell, T., 1993. *Computational Mechanisms for Action Selection*. Thesis (PhD). University of Edinburgh.

Veksler, V.D., Gray, W.D., 2004. State Definition in the Tetris Task: Designing a Hybrid Model of Cognition. *Proceedings of the International Conference on Cognitive Modelling*, July 29 - August 1, 2004, Pittsburgh, PA, USA. Lawrence Earlbaum, pp. 394 - 395.

Weiser, M, 1994. Creating the invisible interface: (invited talk). *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, November 2 - 4, 1994, Marina del Ray, CA USA. ACM Press, p.1.

Wolpert, D.H. and Macready, W.G., 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1 (1), pp. 67 - 82.

Wooldridge, M.J., 2002. *An Introduction to MultiAgent Systems*. England: John Wiley & Sons Ltd.

Appendix A: Sample Plan Files

The plan files listed here are those demonstrated in Chapter 4. Note that the plans shown here are merely a representative sample of those created during the development process: many more plans were created than those shown here.

The format of plan files is given in Bryson (2001; pp. 225 - 226). Note also that semicolons (`;`) begin comments which last until the end of the line. Furthermore, since some of these plans were adapted during the writing of Chapter 4 to make the given scenarios more informative, I have not included header information (e.g. date of creation).

Walking To Navigation Points

```
(
  (C wander-around (minutes 10) (goal((see-player)))
    (elements
      ((stuck (trigger ((is-stuck))) avoid))
      ((pickup (trigger ((see-item))) pickup-item))
      ((walk-around (trigger ((is-rotating False))) walk))
    )
  )

  (AP avoid (minutes 10) (stop-bot rotate then-walk))

  (C then-walk (minutes 10) (goal((is-walking)))
    (elements
      ((try-walk (trigger ((is-rotating False))) walk))
    )
  )

  (C get-to-enemy-base (minutes 10) (goal ((at-enemy-base)))
    (elements
      (
        (find-base (trigger((reachable-nav-point)))
            walk-to-nav-point)
        (wander-base (trigger((succeed))) wander-around)
      )
    )
  )

  (RDC life (goal ((fail)))
      (drives
        ((hit (trigger((hit-object)(is-rotating False))) avoid))
        ((to-enemy-base (trigger((succeed))) get-to-enemy-base))
      )
  )
)
```

A Greater Awareness of Flags

```
(
  (C wander-around (minutes 10) (goal((see-player)))
    (elements
      ((stuck (trigger ((is-stuck))) avoid))
      ((pickup (trigger ((see-item))) pickup-item))
      ((walk-around (trigger ((is-rotating False))) walk))
```

```
    )
  )

  (AP avoid (minutes 5) (stop-bot rotate then-walk))

  (C then-walk (minutes 10) (goal((is-walking)))
    (elements
      ((try-walk (trigger ((is-rotating False))) walk))
    )
  )

  (C get-to-enemy-base (minutes 10) (goal ((at-enemy-base)))
    (elements
      (
        (find-base (trigger((reachable-nav-point)))
            walk-to-nav-point)
        (find-nav-point (trigger((succeed))) rotate 10)
        (wander-base (trigger((succeed))) wander-around)
      )
    )
  )

  (AP go-to-own-base (minutes 10) (to-own-base))
  (AP go-to-own-flag (minutes 10) (to-own-flag))
  (AP go-to-enemy-flag (minutes 10) (to-enemy-flag))

  (AP attack-enemy-carrying-our-flag (minutes 20)
      (shoot-enemy-carrying-our-flag
       run-to-enemy-carrying-our-flag))

  (RDC life (goal ((fail)))
      (drives
        ((pickup-our-flag-from-ground
            (trigger ((our-flag-on-ground))) go-to-own-flag))
        ((pickup-enemy-flag-from-ground
            (trigger ((enemy-flag-on-ground))) go-to-enemy-flag))
        ((attack-enemy-with-our-flag
            (trigger ((see-enemy-with-our-flag)))
                attack-enemy-carrying-our-flag))
        ((take-enemy-flag-from-base
            (trigger((enemy-flag-reachable)
            (have-enemy-flag False)))
                go-to-enemy-flag))
        ((hit (trigger((hit-object)(is-rotating False))) avoid))
        ((go-home (trigger((have-enemy-flag))) go-to-own-base))
        ((to-enemy-base (trigger((succeed))) get-to-enemy-base))
      )
  )
)
```

Responding to Attack

Note the commented lines for avoiding projectiles – see section 5.3.4.

```
(
  (C wander-around (minutes 10) (goal((reachable-nav-point)))
```

```
  (elements
    ((stuck (trigger ((is-stuck))) avoid))
    ((walk-around (trigger ((is-rotating False))) walk))
  )
)

(AP avoid (minutes 5) (stop-bot rotate then-walk))

(C then-walk (minutes 10) (goal((is-walking)))
  (elements
    ((try-walk (trigger ((is-rotating False))) walk))
  )
)

(C get-to-enemy-base (minutes 10) (goal ((at-enemy-base)))
  (elements
    (
      (check-immediate-vicinity
          (trigger((too-close-for-path))) big-rotate 2)
      (run-to-base (trigger((know-enemy-base-pos)))
          to-enemy-base)
      (find-base (trigger((reachable-nav-point)))
          walk-to-nav-point)
      (find-nav-point (trigger((succeed))) rotate 10)
      (wander-base (trigger((succeed))) wander-around)
    )
  )
)

(C go-to-own-base (minutes 10) (goal ((at-own-base)))
  (elements
    (
      (check-immediate-vicinity (trigger((too-close-for-path)))
           big-rotate 2)
      (run-to-own-flag (trigger((our-flag-reachable)))
          to-own-flag)
      (run-to-own-base (trigger((know-own-base-pos)))
          to-own-base)
      (find-base (trigger((reachable-nav-point)))
          walk-to-nav-point)
      (find-nav-point (trigger((succeed))) rotate 10)
      (wander-base (trigger((succeed))) wander-around)
    )
  )
)


(AP go-to-own-flag (minutes 10) (to-own-flag))
(AP go-to-enemy-flag (minutes 10) (to-enemy-flag))

; no point having these stay on the stack for
; long as they get called very often anyway
(AP expire-the-damage-info (seconds 8) (expire-damage-info))
(AP expire-the-focus-info (seconds 10) (expire-focus-info))
(AP expire-the-reachable-info (seconds 6)
```

```
    (expire-reachable-info))
(AP expire-the-projectile-info (seconds 2)
    (expire-projectile-info))

; may need a better goal, but timeout should do it for now
(C respond-to-attack (seconds 10) (goal ((fail)))
  (elements
    (
      (attack-visible-attacker
          (trigger ((taken-damage-from-specific-player)))
              respond-to-visible-attacker)
      (find-attacker (trigger ((succeed))) try-to-find-attacker)
    )
  )
)

(AP respond-to-visible-attacker (seconds 4)
    (set-attacker face-attacker shoot-attacker))

; may need a better goal, but timeout should do it for now
(C try-to-find-attacker (seconds 3) (goal ((fail)))
  (elements
    (
      (found-attacker (trigger ((see-enemy)))
          respond-to-visible-attacker)
      (spin (trigger ((succeed))) big-rotate 1)
    )
  )
)

(AP attack-enemy-carrying-our-flag (minutes 20)
    (shoot-enemy-carrying-our-flag
    run-to-enemy-carrying-our-flag))
(AP get-medkit (minutes 1) (runto-medical-kit))
(AP get-weapon (minutes 1) (runto-weapon))

(RDC life (goal ((fail)))
    (drives
      ;((expire-our-projectile-info (trigger ((succeed)))
          ;expire-the-projectile-info (seconds 3)))
      ((expire-our-damage-info (trigger ((succeed)))
          expire-the-damage-info (seconds 10)))
      ((expire-our-reachable-info (trigger ((succeed)))
          expire-the-reachable-info (seconds 20)))
      ((expire-our-focus-info (trigger ((succeed)))
          expire-the-focus-info (seconds 30)))

      ((pickup-our-flag-from-ground
          (trigger ((our-flag-on-ground))) go-to-own-flag))
      ;will be something particular if required in future
      ;((avoid-being-shot (trigger ((incoming-projectile)))
          ;wander-around))
      ((pickup-enemy-flag-from-ground
          (trigger ((enemy-flag-on-ground))) go-to-enemy-flag))
      ((pickup-medkit-as-health-low
```

```
            (trigger ((see-reachable-medical-kit)
                (own-health-level 30 <))) get-medkit))
        ((pickup-weapon-as-unarmed
            (trigger ((see-reachable-weapon) (are-armed False)))
                get-weapon))

        ((respond-to-attack-since-health-low
            (trigger ((taken-damage) (own-health-level 30 <)
                (armed-and-ammo))) respond-to-attack))
        ((attack-enemy-with-our-flag
            (trigger ((see-enemy-with-our-flag)))
                attack-enemy-carrying-our-flag))
        ((take-enemy-flag-from-base
            (trigger((enemy-flag-reachable)
                (have-enemy-flag False))) go-to-enemy-flag))
        ((respond-to-attack-health-not-low
            (trigger ((taken-damage) (armed-and-ammo)
            (is-responding-to-attack False)))
                    respond-to-attack))
        ((hit (trigger((hit-object)(is-rotating False))) avoid))
        ((go-home (trigger((have-enemy-flag))) go-to-own-base))
        ((get-yourself-to-enemy-base (trigger((succeed)))
            get-to-enemy-base))
      )
  )
)
```

Appendix B: Initial Behaviour Decomposition

High-Level Descriptions

For simplicity, descriptions (and sequences) are grouped under two roles – attacker and defender. In reality, as alluded to below, bots will take on different roles at different times as circumstances require. However, it is useful to have an explicit role at the start of the game as there are not yet any circumstances to react to and the bot needs to know what to do: we do not wish all bots to run to the enemy base and neither do we wish none of them to.

Furthermore, the plans below assume no communication between bots as, at this stage, it is not certain whether time will permit a team of agents to be created or simply a single agent. It is also assumed that agents will not have an internal map of the play-area. Again, this will be added (and thus plans updated) if time permits.

Attackers should make their way to the enemy base, capture their flag and return to their own base. They will need to able to discover where the enemy base is, and should be able to deal with (using either stealth / avoidance or firepower) any members of the opposing team whom they encounter. In some cases, this may involve a temporary change in role:

If an attacker encounters an enemy who is carrying a flag from the attacker's team, then he (the attacker) should switch to the defender role, and thus attempt to kill the enemy and retrieve the flag. If the attacker is carrying the enemy flag and his health is dangerously low, and there are other teammates around who could attack the enemy instead, then he should run away as were he to be killed, he would drop the flag and the enemy could possibly retrieve it.

Defenders should either wander around near their flag or, if a suitable nearby vantage point is found (for example, a doorway through which an attacker must enter), move to it. Upon encountering an enemy player, they should begin to attack him (chasing him if necessary) and prevent him from reaching the flag by moving between him and it as well as just shooting. If a number of enemies are encountered, the defender should prioritise attacking those who are attacking him most aggressively, and those at highest risk of reaching the flag. If a flag-carrying enemy begins to return to his base then he should be pursued and attacked as far as required – there is nothing left at the bot's own base to defend!

If a bot (in either the role of defender or attacker) sees the enemy's flag lying on the ground, he should pick it up and attempt to return to his base.

All bots need to consider their own health and resources, collecting weaponry, ammunition and medical supplies as required. Specifically, bots should gather weaponry and ammunition at the start of the game, to enable them to deal with enemies they meet whilst carrying out their roles.

Sequences of Actions

Capture Flag: collect weaponry and ammunition, move to the enemy base, capture their flag (i.e. move to it) and return to own team's base.

Defend Flag: (no enemies present) collect weaponry and ammunition, wander around near base or near vantage point.

Defend Flag (Enemy/ies present): determine highest priority enemy (see above), if chosen enemy too far away or too near flag, move nearer, shoot enemy repeatedly.

Encounter Enemy not carrying team's flag (attacker): If enemy has noticed us and there's only one enemy and health above a certain threshold, attack enemy. Otherwise, run towards target.

Respond to enemy attack: If don't know who attacker is, do nothing. Otherwise, face attacker (and keep facing him/her if move elsewhere) and shoot.

Attack enemy: Run towards the enemy, shooting him/her.

Collect inventory item (e.g. weaponry / ammunition / health): if we are not being attacked, wander around near walls, run to required item if seen. Otherwise, run away until no longer being attacked. If see required item while running, collect it.

Run away: run backwards in a slight zig-zag, continuing to shoot at any enemy which is shooting us. If hit wall, turn 90° clockwise and continue. If turning 90°clockwise means we are running in the direction of the enemy, turn anticlockwise instead.

Sensory Primitives

The bots will need to be able to see flags and identify their location. They will need to be able to detect walls when wandering around. Enemies will need to be seen and their position relative to other objects (e.g. the flag, the bot itself) identified. Furthermore, the bot will need to be able to identify whether the enemy has noticed him, whether a given enemy is carrying the bot's flag and whether a particular enemy is shooting at him. Determining the highest-priority enemy to shoot is a further required sense.

The bot will need to be able to see objects such as medical kits, ammunition, weapons and body-armour.

A sense of the bot's own health level, ammunition level and weapons held / currently in use is required. The bot is required to know whether or not he is currently being shot at.

Action Primitives

The bot needs to be able to run and walk. The bot needs to be able to shoot at a particular target. The bot needs to be able to collect items (although this is in fact simply a case of running or walking into them). The bot needs to be able to turn both clockwise and anticlockwise.

Required State

State is required to store the position of the bot's own base and the enemy's base. State should be provided to give the bot's own health level, ammunition level and weapons held / currently in use. State should also show where the bot currently is, both in terms of an absolute location (co-ordinates) and a conceptual location (e.g. "at enemy base").

The bot needs state to store details of the enemies currently in his locality, and their state (e.g. ignorant of his presence, shooting him, carrying the flag, etc). State should also indicate the enemy with the highest priority for shooting. Similarly, state is required to give details of the other teammates in his vicinity.

Some state should also give details of the enemy's flag and the bot's own flag (e.g. at base, captured, dropped).

Based on these details of state, we can imagine the following behaviours:

- Movement – containing state to do with positions of objects, bases and the bot

himself.

- Status – containing state regarding health level, weapons held and so on.
- Combat – state about who is attacking the bot, what enemies are around and what teammates are around.

Goals and Drives

The following is a list of goals or drives the bot should be expected to act on, in descending order of priority. This first list assumes that, in situations where either role can be taken, the bot is an attacker rather than a defender. Note that actions such as *Attack enemy who is carrying our flag* assume that the bot can see this enemy – he would not abandon an almost-complete attempt to pick up the enemy flag to run after the enemy who has taken our flag, for example.

1. Pick up our team's flag from where it has been dropped
2. Avoid incoming projectile
3. Pick up the opposing team's flag from where it has been dropped
4. Pick up nearby medical kit (if health in dangerous range)
5. Pick up nearby weapon (if unarmed)
6. Pick up nearby ammunition (if have none)
7. Respond to enemy attack (if health in dangerous range)
8. Attack enemy who is carrying our flag
9. Attack enemy who is near our flag
10. Pick up nearby ammunition (if already have some)
11. Pick up enemy flag (i.e. when it's at their base and so are we)
12. Respond to enemy attack
13. Run to our base (i.e. with the flag)
14. Pick up nearby weapon (if already armed)
15. Find medical kit, weapon or ammunition (as required).
16. Run to enemy base
17. Pick up nearby medical kit (if health not in dangerous range)

If the bot were a defender by default, the list would be as above, but with the following changes:

The following drive would be inserted between 8 & 9:

Run after enemy carrying our flag

Item 11 would be removed and items from 14 onwards would be replaced by the following:

14. Run to own base

15. Find medical kit, weapon or ammunition (as required).
16. Look for vantage point
17. Wander around near our flag / vantage point
18. Pick up nearby medical kit (if health not in dangerous range)

Appendix C: E-mail from John Laird

This e-mail is part of Laird's response to my questions about his development of the Soar Quakebot (Laird and Duchi, 2000).

Date: Wed, 9 Mar 2005 10:30:35 -0500
From: John Laird <laird AT umich.edu>[10]
To: 'Sam Partington' <sam AT samsolutions.co.uk>
Subject: RE: Soar Quakebot Development Process? (Research for Dissertation)

We maintain a combination of data from different "sources":
1. When the system gets a mission either at start up or from another entity (such as a commander), it creates a mission structure that describes its role, the role of other agents, its goals, etc.
2. For many of the systems, they "live" in buildings and they will create an internal map of the building either from recalling it from previous runs (we have to explicitly save it now, but that is a detail), or from the current run. The agent annotates the map with relevant information such as path data (what door should I go through in this room to go to some other room), observability data - what rooms can I see from this room, etc.
3. During a run the agent will further annotate the map and mission structure with its specific progress and relevant tactical information that it accumulates from its sensors. Where am I in the world, what parts of the mission have I completed, what is the most important threat, where is the threat - what door is the threat likely to come through, what door in the current room can I use to escape, where are attack points in the current room, ...

None of this is hard-coded. I might be wrong, but I think this is where the behavior-based approaches really make it tough on themselves by not making it easy to encode complex state information, derived from many sources and both about the past and the current situation.

John

[10] The e-mail addresses given in this document have been written with "AT" in place of the @ symbol. This is to prevent e-mail-harvesting robots from successfully extracting them from online versions of this document.

Appendix D: Readme from Distribution

The distribution is discussed briefly in section 8.4. The version of the readme given here is slightly updated from that which I originally distributed, but I hope to make this improved version available shortly.

```
############################################################
                          BODbot
############################################################

=======
Installing
=======

1) First of all, install pyposh (see
http://www.cs.bath.ac.uk/~jjb/web/pyposh.html)

2) Put the bodbot directory and its contents (including
subdirectories) as a subdirectory of "modules"

3) Replace the relevant files in the main pyposh directory
with those in "pyposh fixes"

4) Put CTF-Simple2dt.unr into your Unreal Tournament maps
folder.  This is the map I've tested the agents on most, as
it's pretty simple.  I suggest you use it too.  It's a
slight tweak of CTF-Simple2.unr, the map the Gamebots people
made.

5) Run PyPOSH as per Andy's instructions.  Although the plan
will work for agents on either team, I've tested it mainly
for the blue team's agent.

======
BODbot
======

Known Issues
---------------------

1) The biggest outstanding problem is that when the bodbot
receives messages from Gamebots which require it to update
its internal knowledge (e.g. the "pth" message about a
recommended path), those are updated immediately (see
functions such as pass_pth_details).

This can cause a problem if this updating interrupts a
running action as it can clear some variable which the
action is using.  The best example of this is when passing
```

details of whether a point is reachable (pass_rch_details etc). It only happens occasionally, but is a problem!

This should be modified so that updating this information is another item run as part of running the plan (like the "expire" stuff already there) as two actions cannot execute at the same time. I may do this myself if I have time.

2) One other thing is that the bot does not seem very good at noticing when the player has its flag. It does sometimes, but less often than I'd expect. Not sure about the cause of this.

3) Projectile information (i.e. the "prj" message from Gamebots) is *never* sent from Gamebots. This is a problem with Gamebots, not with my code (see http://www.andrew.cmu.edu/user/roman/15396/game_bots_api.html). Similarly, parameters to "init" are ignored, so you cannot specify the team or name of your bot (although you must provide a name for pyposh to be happy, even though Gamebots then ignores it). Both these problems are corrected in a new version of Gamebots, available from http://www.cs.rit.edu/~jdb/gameAI/gamebots/codechange.html, although I have not tested this new version myself.

4) Have not managed to get the debugger or profiler working with pyposh. I've managed to get the profiler to profile the GUI, but not the stuff which that then runs. These two problems probably stem from my python ignorance.

Debug output

I've left lots of this in, feel free to comment / remove as necessary.

(If you're interested in detail, things to look out for include "is stuck?" when the bot gives up trying to follow navpoints and wanders around as per Andy's poshbot. In general, output with question marks indicates a sense running, though not all senses give output.)

===================
Tweaks / Fixes to pyposh
===================

Multiple Behaviour files

The most interesting difference with the main bodbot agent file (bodbot.py) is the make_behaviour function. Unlike all of Andy Kwong's agents, this allows multiple behaviour

files. Note that it now returns a *list* of behaviours.

This change required quite a few changes to PyPOSH. For instance, posh_agent.py's self.behavior_instance variable is now a list rather than a reference to a single object (so it's differently initialised and tested than before).

Furthermore, posh_agent has been modified to include sense and action dictionaries (lines 89 & 90) and functions which use these (i.e. get_act and get_sense) have also been modified. Similarly, the add_act and add_sense functions add details to these central dictionaries, rather than passing them on to the behaviour file.

The behaviour functionality is fairly logically separated, except that andybehaviour.py contains a range of things which should probably be in movement.py. I gave them their own file to separate those primitives written by Andy from those written by me (although big_rotate is mine).

pyposh.py

None of the changes here are essential but were all included to speed up testing. You can find them in the code by searching for "###", as they're highlighted by comment blocks.

A couple of tweaks which selected the right options in the lists are commented as your setup may vary. The ones which populate the options table and position the windows have been left in.

posh_agent.py

The frequency attribute of POSH elements caused a crash, thanks to a problem on line 923 of this file. See the comment for details of what's changed.
Similarly, using limits on the number of retires caused a crash, as the relevant plan element was not being cast to an integer before being modified. This was also fixed (see line 624 and associated comment).
I've also modified it to allow for multiple behaviour files (see above).

=======
old plans
=======

These are provided in case you want to try a simpler agent. (Note that bodbotattack[3].lap doesn't execute correctly -- see comment)

========
Other files
========

The text files (*.txt) in the bodbot folder are for your
interest only and *can be ignored*. They show printouts of
various bits of information from Gamebots, as represented in
pyposh. (If I remember rightly, the "R>>" in sample
Gamebots info.txt is debug output and can be ignored)

(The weird line breaks in these are simply due to the width
of my command window).

=====================
Feedback / Document details
=====================

This version of the readme produced 26/3/2005
Comments / Questions to sam AT samsolutions.co.uk

Appendix E: Code Listings and CD Contents

The independently-numbered pages which follow give code listings for the following files:

- `bodbot.py` – this is the main agent file. It coordinates communication with Gamebots and manages the dictionaries of the actions and senses from the various behaviour modules.
- `combat.py` – this is one of the behaviour modules (`CombatBehaviour` class). It also contains the `CombatInfoClass` class (see section 5.3.2)
- `movement.py` – another behaviour module (`MovementBehaviour` class). This also contains the `PositionsInfo` class (again, section 5.3.2).
- `status.py` – the third behaviour module (`StatusBehaviour` class). The final behaviour module (`AndyBehavior` class in `andybehaviour.py`) is not listed as the majority of the code is taken directly from Kwong (2003). It is given on the enclosed CD, however.
- `utilityfns.py` – this file holds utility functions, described in section 5.3.2.

Plan code is not given, that can be found in Appendix A.

As well as these files, the enclosed CD contains the following:

- `andybehaviour.py` – the final behaviour module (see note above).
- Compiled versions of `bodbot.py`, `combat.py`, `movement.py`, `status.py`, `utilityfns.py` and `andybehaviour.py`.
- The `1st distribution` folder, containing those files distributed as described in section 8.4 (but with the updated readme file). Note that this is where the modified PyPOSH files (chapter 8) may be found.
- `to weapon brightened.jpg`, a colour version of figure 4.1.
- The plans given in Appendix A
- A PDF [Portable Document Format] version of this document.

All these files, with the exception of the PDF, are to be found in the zipped file `sampartington-full-2004-5.zip`. This zip file also contains a second (identical) copy of the PDF.

The behaviour modules and plans are organised in the `bodbot` directory and subdirectory in such a way that this directory can be copied into the `modules` directory of a PyPOSH distribution and used from there. For this reason, the directory also contains the initialisation file `__init__.py` (and a compiled version thereof). Further details about how the code can be run can be found in Appendix D.

```
1   #  BODbot created as a means of evaluating Behaviour Oriented Design [BOD]
2   #  Much code here re-used from Andy Kwong's poshbot
3
4   #  We need to start a comms thread in order to get updates
5   #   to the agent status from the server.
6
7   from socket import *
8   from posh_core import *
9   import re #re is for Regular Expressions
10  import thread
11  import posh_utils
12  import utilityfns
13
14  # import any behaviour files
15  import andybehaviour
16  import movement
17  import status
18  import combat
19
20  # Init world in this example connects to gamebots server
21  def init_world(*args, **kw):
22      pass
23
24
25  # Returns a list of behavior objects
26  def make_behavior(ip, port, botname, agent, *args, **kw):
27      bot = Bot_Agent(ip, port, botname) # Bot_Agent keeps a local copy of the
        bot state
28      BList = []
29
30      agent.bot = bot
31      bot.agent = agent # was done in bind_bot, now moved here as doing it in ab
        is too low-level
32
33      # Andy's primitives
34      ab = andybehaviour.AndyBehavior(agent = agent)
35      ab.bind_bot(bot) #sets ab's bot to the arg sent
36      #ab.bot.connect() now done below
37      BList.append(ab)
38
39      PosInfo = movement.PositionsInfo()
40      CombatInfo = combat.CombatInfoClass()
41
42      mb = movement.MovementBehaviour(PosInfo, CombatInfo, agent = agent)
43      mb.bind_bot(bot)
44      BList.append(mb)
45
46      sb = status.StatusBehaviour(PosInfo, agent = agent)
47      sb.bind_bot(bot)
48      BList.append(sb)
49
50      cb = combat.CombatBehaviour(PosInfo, CombatInfo, agent = agent)
51      cb.bind_bot(bot)
52      BList.append(cb)
53
```



```
54      bot.connect()
55
56      return BList
57
58  # Called when pyposh is shutting down
59  def destroy_world():
60      pass
61
62
63
64  # Keeps a local copy of the bot state. Gamebots does not support
65  # queries on the agent sense, it sends a copy of the environment
66  # to the agent periodically.
67  class Bot_Agent:
68      def __init__(self, ip, port, botname):
69          self.ip = ip
70          self.port = port
71          self.botname = botname
72          self.events = [] #things like hitting a wall
73          self.conninfo = {}
74          self.gameinfo = {}
75          self.view_players = {}
76          self.view_items = {}
77          self.nav_points = {}
78          self.botinfo = {}
79          self.s_gameinfo = {}
80          self.s_view_players = {}
81          self.s_view_items = {}
82          self.s_nav_points = {}
83          self.s_botinfo = {}
84          self.msg_log = [] # Temp Log for message received
85          self.msg_log_max = 4096 # Max Temp Log size
86          self.sent_msg_log = [] # Temp Log for messages sent
87          self.sent_msg_log_max = 6 # Max Temp Log size
88          self.hit_timestamp = 0 # Used to inhibit was_hit()
89          self.thread_active = False
90          self.kill_connection = False
91          self.rotation_hist = []
92          self.velocity_hist = []
93          self.thread_active = False
94          self.conn_ready = False
95          self.conn_thread_id = None
96
97
98      def debug(self, level, message):
99          try:
100             self.agent.debugboard.add(level, message)
101         except:
102             print "Critical Failure - BotAgent cannot write to debugboard"
103             print message
104             raise
105
106     def proc_item(self, string):
```

```
107         (cmd, varstring) = re.compile('\s+').split(string, 1) #\s is a special
            escape character, which matches any white-space, varstring will hold
            flags returned from regular expression creation (see sre.py)
108         vars = re.compile('\{(.*?)\}').findall(varstring)
109         var_dict = {}
110         for var in vars:
111             (attr, value) = re.compile('\s+').split(var, 1)
112             var_dict[attr] = value
113         return (cmd, var_dict)
114 
115     # Calls connect_thread in a new thread
116     def connect(self):
117         self.debug(5, "Connecting to Server")
118         if not self.conn_thread_id:
119             self.thread_active = True
120             self.conn_thread_id = thread.start_new(self.connect_thread, ())
121             return True
122         else:
123             self.debug(1, "Attempting to Connect() when thread already active")
124             return False
125 
126 
127     # This method runs inside a thread updating the agent state
128     # by reading from the network socket
129     def connect_thread(self):
130         self.kill_connection = False
131         try:
132             self.sockobj = socket(AF_INET, SOCK_STREAM)
133             self.sockobj.connect((self.ip, int(self.port)))
134             self.sockin = self.sockobj.makefile('r')
135             self.sockout = self.sockobj.makefile('w')
136         except:
137             self.debug(1, "Connection to server failed")
138             self.kill_connection = True # Skip the read loops
139         else:
140             self.debug(1, "Connected to server")
141             self.kill_connection = False
142 
143         # This loop waits for the first NFO message
144         while not self.kill_connection:
145             try:
146                 x = self.sockin.readline()
147             except:
148                 self.debug(1, "Connection Error on readline()")
149                 self.kill_connection = True
150                 break
151 
152             if not x:
153                 self.debug(1, "Connection Closed from Remote End")
154                 self.kill_connection = True
155                 break
156 
157             #print x
158             (cmd, dict) = self.proc_item(x)
159             if cmd == "NFO":
```



```
160                 # Send INIT message
161                 self.conninfo = dict
162                 self.send_message("INIT", {"Name" : self.botname})
163                 self.conn_ready = True # Ready to send messages
164                 break
165 
166         # Now the main loop
167         # Not everything is implemented. Just some basics
168         while not self.kill_connection:
169             try:
170                 x = self.sockin.readline()
171             except:
172                 self.debug(1, "Connection Error on readline()")
173                 break
174 
175             if not x:
176                 self.debug(1, "Connection Closed from Remote End")
177                 break
178 
179             #print "R>> " + x
180             (cmd, dict) = self.proc_item(x)
181             sync_states = ("SLF","GAM","PLR","NAV","MOV","DOM","FLG","INV")
182             events = ("WAL", "BMP")
183             self.msg_log.append((cmd, dict))
184             if cmd == "BEG":
185                 # When a sync batch is arriving, make sure the shadow
186                 # states are cleared
187                 self.s_gameinfo = {}
188                 self.s_view_players = {}
189                 self.s_view_items = {}
190                 self.s_nav_points = {}
191                 self.s_botinfo = {}
192             elif cmd in sync_states:
193                 # These are sync. messages, handle them with another method
194                 self.proc_sync(cmd, dict)
195             elif cmd == "END":
196                 # When a sync batch ends, we want to make the shadow
197                 # states that we were writing to to be the real one
198                 self.gameinfo = self.s_gameinfo
199                 self.view_players = self.s_view_players
200                 self.view_items = self.s_view_items
201                 self.nav_points = self.s_nav_points
202                 self.botinfo = self.s_botinfo
203                 # Also a good time to trim the events list
204                 # Only keep the last 50 events
205                 self.events = self.events[-50:]
206                 self.msg_log = self.msg_log[-1000:]
207             elif cmd in events:
208                 # The bot hit a wall or an actor, make a note
209                 # of it in the events list with timestamp
210                 self.events.append((posh_utils.current_time(), cmd, dict))
211             elif cmd == "SEE":
212                 # Update the player positions
213                 self.view_players[dict["Id"]] = dict
214             elif cmd == "PTH":
```

```
215                 # pass the details to the movement behaviour
216                 self.pass_pth_details(dict)
217             elif cmd == "RCH":
218                 self.pass_rch_details(dict)
219             elif cmd == "PRJ": # incoming projectile
220                 self.pass_prj_details(dict)
221             elif cmd == "DAM": # damage taken
222                 self.pass_dam_details(dict)
223             elif cmd == "KIL": # some other player died
224                 self.pass_kil_details(dict)
225             elif cmd == "DIE": # this player died
226                 self.pass_die_details(dict)
227             else:
228                 pass
229 
230         self.debug(5, "Closing Sockets and Cleaning Up...")
231         try:
232             self.sockout.flush()
233             self.sockout.close()
234             self.sockin.close()
235             self.sockobj.close()
236         except:
237             self.debug(1, "Error closing files and sockets")
238 
239         self.thread_active = False
240         self.conn_ready = False
241         self.conn_thread_id = None
242         self.debug(5, "Connection Thread Terminating...")
243 
244     def disconnect(self):
245         self.kill_connection = True
246 
247     def send_message(self, cmd, dict):
248         string = cmd
249 
250         self.sent_msg_log.append((cmd, dict))
251         # does the list need truncating?
252         if len(self.sent_msg_log) > self.sent_msg_log_max:
253             del self.sent_msg_log[0 : -self.sent_msg_log_max]
254 
255         for (attr, value) in dict.items():
256             string = string + " {" + attr + " " + value + "}"
257         #print "About to send " + string
258         string = string + "\r\n"
259         # print >> self.sockout, string
260         # print "S>> " + string
261         try:
262             self.sockout.write(string)
263             self.sockout.flush()
264         except:
265             self.debug(1, "Message : " + string + " unable to send")
266             return False
267         else:
268             return True
269 
```



```
270     #handles synchronisation messages
271     def proc_sync(self, command, values):
272         if command == "SLF": #info about bot's state
273             self.s_botinfo = values
274             # Keep track of orientation so we can tell when we are moving
275             # Yeah, we only need to know the Yaw
276             self.rotation_hist.append(int(
277                 re.search(',(.*?),', values['Rotation']).group(1)))
278             # Trim list to 3 entries
279             if len(self.rotation_hist) > 3:
280                 del(self.rotation_hist[0])
281             # Keep track of velocity so we know when we are stuck
282             self.velocity_hist.append(self.calculate_velocity( \
283                 values['Velocity']))
284             # Trim it to 20 entries
285             if len(self.velocity_hist) > 20:
286                 del(self.velocity_hist[0])
287 
288         elif command == "GAM": #info about the game
289             self.s_gameinfo = values
290         elif command == "PLR": #another character visible
291             # For some reason, this doesn't work in ut2003
292             self.s_view_players[values["Id"]] = values
293         elif command == "NAV": #a path marker
294             # Neither does this
295             #print "We have details about a nav point at " + values["Location"]
296             self.s_nav_points[values["Id"]] = values
297         elif command == "INV": #an object on the ground that can be picked up
298             #print values
299             self.s_view_items[values["Id"]] = values
300         elif command == "FLG": #info about a flag
301             #pass these details to the movement behaviour as that stores
                details of locations etc and may need them
302             self.pass_flag_details(values)
303             #print("We have details about a flag.  Its values is: " +
                values["State"]);
304         else:
305             pass
306 
307     def find_movement_behaviour(self):
308         #find the movement behaviour, if there is one
309         BList = self.agent.behavior_instance
310         for CurrentB in BList:
311             if isinstance(CurrentB, movement.MovementBehaviour):
312                 return CurrentB
313         return None
314 
315     def find_combat_behaviour(self):
316         #find the combat behaviour, if there is one
317         BList = self.agent.behavior_instance
318         for CurrentB in BList:
319             if isinstance(CurrentB, combat.CombatBehaviour):
320                 return CurrentB
321         return None
322 
```

```
323     def pass_flag_details(self, values):
324
325         MB = self.find_movement_behaviour()
326         if MB != None:
327             MB.receive_flag_details(values)
328
329         # inform the combat behaviour as well
330         CB = self.find_combat_behaviour()
331         if CB != None:
332             CB.receive_flag_details(values)
333
334     def pass_pth_details(self, valuesdict):
335         #print "in pass_pth_details"
336         #print valuesdict
337         MB = self.find_movement_behaviour()
338         if MB != None:
339             MB.receive_pth_details(valuesdict)
340
341     def pass_rch_details(self, valuesdict):
342         #print "in pass_rch_details"
343         #print valuesdict
344         MB = self.find_movement_behaviour()
345         if MB != None:
346             MB.receive_rch_details(valuesdict)
347
348     # tell the combat behaviour about incoming projectile
349     def pass_prj_details(self, valuesdict):
350         CB = self.find_combat_behaviour()
351         if CB != None:
352             CB.receive_prj_details(valuesdict)
353
354     # tell the combat behaviour about damage taken
355     def pass_dam_details(self, valuesdict):
356         CB = self.find_combat_behaviour()
357         if CB != None:
358             CB.receive_dam_details(valuesdict)
359
360     # tell the combat behaviour about the death of another player
361     def pass_kil_details(self, valuesdict):
362         CB = self.find_combat_behaviour()
363         if CB != None:
364             CB.receive_kil_details(valuesdict)
365
366     # tell the combat & movement behaviours about our death
367     def pass_die_details(self, valuesdict):
368         CB = self.find_combat_behaviour()
369         if CB != None:
370             CB.receive_die_details(valuesdict)
371
372         MB = self.find_movement_behaviour()
373         if MB != None:
374             MB.receive_die_details(valuesdict)
375
376     def turn(self, degrees):
377         utangle = int((degrees * 65535) / 360.0)
```



```
378         self.send_message("ROTATE", {"Amount" : str(utangle)})
379         # self.send_message("TURNTO", {"Pitch" : str(0)})
380
381     def get_yaw(self):
382         if self.botinfo.has_key("Rotation"):
383             return int(re.search(',(.*?),', self.botinfo["Rotation"]).group(1))
384         else:
385             return None
386
387     def get_pitch(self):
388         if self.botinfo.has_key("Rotation"):
389             return int(re.match('(.*?),', self.botinfo["Rotation"]).group(1))
390         else:
391             return None
392
393     def move(self):
394         self.send_message("INCH", {})
395         return True
396
397     # Was the bot hit in the last 2 seconds
398     def was_hit(self):
399         lsec = 2 # How many seconds to look back to
400         isec = 0 # Number of seconds to inhibit consecutive was_hits
401         now = posh_utils.current_time()
402         def timefilter(item):
403             (timestamp, command, value) = item
404             if timestamp > now - lsec:
405                 return True
406             else:
407                 return False
408
409         # Filter the events from the last lsec seconds
410         lastevents = filter(timefilter, self.events)
411
412         if self.hit_timestamp > now - isec:
413             # Update the last hit timestamp
414             return False
415         else:
416             # Update the last hit timestamp
417             self.hit_timestamp = now
418             if len(lastevents) > 0:
419                 return True
420             else:
421                 return False
422
423     def turning(self):
424         # compares the most recent to the leat recent rotation_hist
425         # entry. If there is a descrepancy beyond the error fudge,
426         # then we say we are rotating
427         fudge = 386 # in UT units, roughly 2 degrees
428         if len(self.rotation_hist) > 0:
429             c_rot = self.rotation_hist[0]
430             e_rot = self.rotation_hist[-1]
431             diff = abs(c_rot - e_rot)
432             if diff > fudge:
```

```
433                 return True
434 
435         return False
436 
437     def moving(self):
438         # If there is recent velocity, return true
439         if len(self.velocity_hist) > 0:
440             if self.velocity_hist[0] > 0:
441                 return True
442         return False
443 
444     def stuck(self):
445         # If there is a period of no movement, then return true
446         fudge = 0
447         for v in self.velocity_hist:
448             if v > fudge:
449                 return False
450         return True
451 
452     def calculate_velocity(self, v):
453         (vx, vy, vz) = re.split(',', v)
454         vx = float(vx)
455         vy = float(vy)
456         return utilityfns.find_distance((0,0), (vx, vy))
```

```
1   from posh_core import *
2   import utilityfns
3
4   class CombatBehaviour(Base):
5       def __init__(self, PosInfo, CombInfo, **kw):
6           Base.__init__(self, **kw) # Call the ancestor init
7
8           # set up variables
9           self.CombatInfo = CombInfo
10          self.PosInfo = PosInfo
11          self.bot = None
12
13          # register behaviours and senses
14          self.init_acts()
15          self.init_senses()
16
17      def init_acts(self):
18          self.agent.add_act("shoot-enemy-carrying-our-flag",
            self.shoot_enemy_carrying_our_flag)
19          self.agent.add_act("run-to-enemy-carrying-our-flag",
            self.run_to_enemy_carrying_our_flag)
20          self.agent.add_act("expire-damage-info", self.expire_damage_info)
21          self.agent.add_act("expire-focus-info", self.expire_focus_info)
22          self.agent.add_act("expire-projectile-info",
            self.expire_projectile_info)
23          self.agent.add_act("face-attacker", self.face_attacker)
24          self.agent.add_act("set-attacker", self.set_attacker)
25          self.agent.add_act("shoot-attacker", self.shoot_attacker)
26
27      def init_senses(self):
28          self.agent.add_sense("see-enemy-with-our-flag",
            self.see_enemy_with_our_flag)
29          self.agent.add_sense("our-flag-on-ground", self.our_flag_on_ground)
30          self.agent.add_sense("enemy-flag-on-ground", self.enemy_flag_on_ground)
31          self.agent.add_sense("incoming-projectile", self.incoming_projectile)
32          self.agent.add_sense("taken-damage-from-specific-player",
            self.taken_damage_from_specific_player)
33          self.agent.add_sense("taken-damage", self.taken_damage)
34          self.agent.add_sense("is-responding-to-attack",
            self.is_responding_to_attack)
35
36      def bind_bot(self, bot):
37          self.bot = bot
38
39      # === SENSES ===
40
41      def see_enemy_with_our_flag(self):
42          #print "in see_enemy_with_our_flag sense"
43          if len(self.bot.view_players) == 0:
44              #print "  no players visible"
45              return False
46
47          #else check through every player we can see to check whether they're
            the one holding our flag
48          players = self.bot.view_players.values()
```



```
49          for CurrentPlayer in players:
50              #print CurrentPlayer
51              if CurrentPlayer["Id"] == self.CombatInfo.HoldingOurFlag:
52                  print "  can see the player holding our flag"
53                  self.CombatInfo.HoldingOurFlagPlayerInfo = CurrentPlayer
54                  return True
55          #print "  cannot see the player holding our flag (",
56          #print self.CombatInfo.HoldingOurFlag,
57          #print ")"
58          return False
59
60      def our_flag_on_ground(self):
61          if self.PosInfo.OurFlagInfo == {}:
62              return False
63          else:
64              # in case the flag was returned but we didn't actually see it
                happen
65              if not self.bot.gameinfo.has_key("EnemyHasFlag"):
66                  self.PosInfo.OurFlagInfo["State"] = "home"
67
68              if self.PosInfo.OurFlagInfo["State"].lower() == "dropped":
69                  #print "our flag is dropped!"
70                  return True
71          return False
72
73      def enemy_flag_on_ground(self):
74          if self.PosInfo.EnemyFlagInfo == {}:
75              return False
76          elif self.PosInfo.EnemyFlagInfo["State"].lower() == "dropped":
77              return True
78          return False
79
80      def incoming_projectile(self):
81          if self.CombatInfo.ProjectileDetails != None:
82              print "incoming-projectile returning True"
83              return True
84          return False
85
86      def taken_damage_from_specific_player(self):
87          if self.CombatInfo.DamageDetails != None and
            self.CombatInfo.DamageDetails.has_key("Instigator"):
88              print "taken_damage_from_specific_player returning True"
89              return True
90          # alternatively, even if we don't know who shot us this time, we may
            know from another recent attack
91          elif self.CombatInfo.KeepFocusOnLocation != None:
92              return True
93          else:
94              return False
95
96      def taken_damage(self):
97          if self.CombatInfo.DamageDetails != None:
98              return True
99          return False
100
```

```
101     # returns true if we're already responding to the most recent attack
102     # At present just test against KeepFocusOnID.  However, that doesn't 100%
        guarantee that we've started shooting,
103     # just that we know who we ought to shoot.  For now, however, I will use
        this check.
104     def is_responding_to_attack(self):
105         if self.CombatInfo.KeepFocusOnID != None:
106             return True
107         else:
108             return False
109
110     # === ACTIONS ===
111
112     def shoot_enemy_carrying_our_flag(self):
113         if self.CombatInfo.HoldingOurFlag != None and
            self.CombatInfo.HoldingOurFlagPlayerInfo != None:
114             Target = self.CombatInfo.HoldingOurFlag
115             Location = self.CombatInfo.HoldingOurFlagPlayerInfo["Location"]
116             self.bot.send_message("SHOOT", {"Target" : Target, "Location" :
                Location})
117         return True
118
119     def run_to_enemy_carrying_our_flag(self):
120         print "in rtecof very start"
121         if self.CombatInfo.HoldingOurFlag != None and
            self.CombatInfo.HoldingOurFlagPlayerInfo != None:
122             print "in rtecof, past initial if"
123
124             #Target = self.CombatInfo.HoldingOurFlag
125             #if not utilityfns.is_previous_message(self.bot, ("RUNTO",
                {"Target" : Target})):
126             #    self.bot.send_message("RUNTO", {"Target" : Target})
127             #    print "running after enemy"
128
129             Location = self.CombatInfo.HoldingOurFlagPlayerInfo["Location"]
130             if not utilityfns.is_previous_message(self.bot, ("RUNTO",
                {"Location" : Location})):
131                 self.bot.send_message("RUNTO", {"Location" : Location})
132             print "running after enemy"
133         return True
134
135     # not the usual sort of action, but ensures that details about e.g. damage
        taken doesn't reside forever and inform decisions too far into the future
136     def expire_damage_info(self):
137         #print "                ex di"
138         self.CombatInfo.DamageDetails = None
139         #self.CombatInfo.KeepFocusOnID = None
140         #self.CombatInfo.KeepFocusOnLocation = None
141         #self.CombatInfo.TriedToFindAttacker = False
142         return True
143
144     def expire_focus_info(self):
145         self.CombatInfo.KeepFocusOnID = None
146         self.CombatInfo.KeepFocusOnLocation = None
147         self.bot.send_message("STOPSHOOT", {}) # no-one to focus on
```



```
148
149     def expire_projectile_info(self):
150         self.CombatInfo.ProjectileDetails = None
151         return True
152
153     #def set_tried_to_find_attacker(self):
154     #    self.CombatInfo.TriedToFindAttacker = True
155
156
157     # if we can see the player currently, store his ID so e.g. runtos will be
        replaced by strafes to keep him in focus
158     # and issue a turnto command
159     def face_attacker(self):
160         print "in face_attacker"
161         if self.CombatInfo.KeepFocusOnLocation == None and
            self.CombatInfo.KeepFocusOnID == None:
162             return True
163
164         if self.CombatInfo.KeepFocusOnID == None: #just provide location
165             Location = self.CombatInfo.KeepFocusOnLocation
166             Msg = ("TURNTO", {"Location" : Location})
167             utilityfns.send_if_not_prev(self.bot, Msg)
168         else:
169             Target = self.CombatInfo.KeepFocusOnID
170             Location = self.CombatInfo.KeepFocusOnLocation
171             self.bot.send_message("TURNTO", {"Target" : Target})
172         return True
173
174     # sets the attacker (i.e. the keepfocuson one) to be the first enemy player
        we have seen
175     # or the instigator of the most recent damage, if we know who that is
176     def set_attacker(self):
177         print "in set_attacker"
178
179         def find_enemy_in_view():
180             # work through who we can see, looking for an enemy
181             OurTeam = self.bot.botinfo["Team"]
182             print "OurTeam:",
183             print OurTeam
184             Players = self.bot.view_players.values()
185             for CurrentPlayer in Players:
186                 if CurrentPlayer["Team"] != OurTeam:
187                     self.CombatInfo.KeepFocusOnID = CurrentPlayer["Id"]
188                     self.CombatInfo.KeepFocusOnLocation =
                        CurrentPlayer["Location"]
189                     return True
190
191         if len(self.bot.view_players) == 0 or self.bot.botinfo == {}: #if
            botinfo is {}, we can't yet set anything
192             return True
193         else:
194             if self.CombatInfo.DamageDetails != None and
                self.CombatInfo.DamageDetails.has_key("Instigator"):
195                 InstID = self.CombatInfo.DamageDetails["Instigator"]
196                 if self.bot.view_players.has_key(InstID):
```

```
197                     # set variables so that other commands will keep him in
                        view
198                     self.CombatInfo.KeepFocusOnID = InstID
199                     self.CombatInfo.KeepFocusOnLocation =
                        self.bot.view_players[InstID]["Location"]
200                 else:
201                     find_enemy_in_view()
202             else:
203                 find_enemy_in_view()
204             return True
205 
206     def shoot_attacker(self):
207         print "in shoot_attacker"
208         if self.CombatInfo.KeepFocusOnLocation == None:
209             return True
210 
211         if self.CombatInfo.KeepFocusOnID == None: #just provide location
212             Location = self.CombatInfo.KeepFocusOnLocation
213             if not utilityfns.is_previous_message(self.bot, ("SHOOT",
                {"Location" : Location})):
214                 self.bot.send_message("SHOOT", {"Location" : Location})
215         else:
216             Target = self.CombatInfo.KeepFocusOnID
217             Location = self.CombatInfo.KeepFocusOnLocation
218             if not utilityfns.is_previous_message(self.bot, ("SHOOT", {"Target"
                : Target, "Location" : Location})):
219                 self.bot.send_message("SHOOT", {"Target" : Target, "Location" :
                    Location})
220         return True
221 
222     # === OTHER FUNCTIONS ===
223 
224     def receive_flag_details(self, values):
225         # if its status is "held", update the CombatInfoClass to show who's
            holding it
226         # otherwise, set that to None as it means no-one is holding it
227 
228         #print "in rfd"
229         #print values
230 
231         if self.bot.botinfo == {}: #if botinfo is {}, we can't yet set anything
232             return
233 
234         OurTeam = self.bot.botinfo["Team"]
235         #print "OurTeam is of type",
236         #print type(OurTeam),
237         #print " and value is",
238         #print OurTeam
239         #print "values[\"Team\"] is ",
240         #print values["Team"]
241 
242         if values["Team"] == OurTeam:
243             if values["State"].lower() == "held":
244                 #print "setting holder"
245                 self.CombatInfo.HoldingOurFlag = values["Holder"]
```



```
246             else:
247                 #print "not being held"
248                 self.CombatInfo.HoldingOurFlag = None
249                 self.CombatInfo.HoldingOurFlagPlayerInfo = None
250 
251     def receive_prj_details(self, valuesdict):
252         print "received details of incoming projectile!"
253         print valuesdict
254         self.CombatInfo.ProjectileDetails = valuesdict
255 
256     def receive_dam_details(self, valuesdict):
257         self.CombatInfo.DamageDetails = valuesdict
258 
259     # handle details about a player (not itself) dying
260     # remove any info about that player from CombatInfo
261     def receive_kil_details(self, ValuesDict):
262         print "receive_kil_details",
263         print ValuesDict
264         print "-----"
265         print self.CombatInfo.HoldingOurFlag
266         if ValuesDict["Id"] == self.CombatInfo.HoldingOurFlag:
267             self.CombatInfo.HoldingOurFlag = None
268             self.CombatInfo.HoldingOurFlagPlayerInfo = None
269             self.bot.send_message("STOPSHOOT", {})
270 
271         if ValuesDict["Id"] == self.CombatInfo.KeepFocusOnID:
272             self.CombatInfo.KeepFocusOnID = None
273             self.CombatInfo.KeepFocusOnLocation = None
274             self.bot.send_message("STOPSHOOT", {})
275 
276 
277     # clean-up after dying
278     def receive_die_details(self, ValuesDict):
279         self.expire_damage_info()
280         self.expire_focus_info()
281 
282 
283 class CombatInfoClass:
284     def __init__(self):
285         self.HoldingOurFlag = None # the ID of the player holding our flag
286         self.HoldingOurFlagPlayerInfo = None # details about that player
287 
288         self.ProjectileDetails = None
289         self.DamageDetails = None
290         self.KeepFocusOnID = None
291         self.KeepFocusOnLocation = None
292 
293         #self.TriedToFindAttacker = False
```

```
1   from posh_core import *
2   import utilityfns
3
4   class MovementBehaviour(Base):
5       def __init__(self, PosInfo, CombatInfo, **kw):
6           Base.__init__(self, **kw) # Call the ancestor init
7
8           # set up variables
9           self.PosInfo = PosInfo
10          self.CombatInfo = CombatInfo
11          self.bot = None
12
13          # register behaviours and senses
14          self.init_acts()
15          self.init_senses()
16
17          # set up useful constants
18          self.PathHomeID = "PathHome"
19          self.ReachPathHomeID = "ReachPathHome"
20          self.PathToEnemyBaseID = "PathThere"
21          self.ReachPathToEnemyBaseID = "ReachPathThere"
22
23      def init_acts(self):
24          self.agent.add_act("walk-to-nav-point", self.walk_to_nav_point)
25          self.agent.add_act("to-enemy-flag", self.to_enemy_flag)
26          self.agent.add_act("to-own-base", self.to_own_base)
27          self.agent.add_act("to-own-flag", self.to_own_flag)
28          self.agent.add_act("to-enemy-base", self.to_enemy_base)
29          self.agent.add_act("inch", self.inch)
30          self.agent.add_act("runto-medical-kit", self.runto_medical_kit)
31          self.agent.add_act("runto-weapon", self.runto_weapon)
32          self.agent.add_act("expire-reachable-info", self.expire_reachable_info)
33
34      def init_senses(self):
35          self.agent.add_sense("at-enemy-base", self.at_enemy_base)
36          self.agent.add_sense("at-own-base", self.at_own_base)
37          self.agent.add_sense("know-enemy-base-pos", self.know_enemy_base_pos)
38          self.agent.add_sense("know-own-base-pos", self.know_own_base_pos)
39          self.agent.add_sense("reachable-nav-point", self.reachable_nav_point)
40          self.agent.add_sense("enemy-flag-reachable", self.enemy_flag_reachable)
41          self.agent.add_sense("our-flag-reachable", self.our_flag_reachable)
42          self.agent.add_sense("see-enemy", self.see_enemy)
43          self.agent.add_sense("see-reachable-medical-kit",
            self.see_reachable_medical_kit)
44          self.agent.add_sense("see-reachable-weapon", self.see_reachable_weapon)
45          self.agent.add_sense("too-close-for-path", self.too_close_for_path)
46
47      def bind_bot(self, bot):
48          self.bot = bot
49
50      # === SENSES ===
51
52      def at_enemy_base(self):
53          #print "in at_enemy_base sense"
54          if not self.bot.botinfo.has_key("Location"):
```



```
55              return False
56          LocTuple =
            utilityfns.location_string_to_tuple(self.bot.botinfo["Location"])
57
58          if self.PosInfo.EnemyBasePos == None:
59              return False
60          else:
61              (SX, SY, SZ) = LocTuple
62              (EX, EY, EZ) =
                utilityfns.location_string_to_tuple(self.PosInfo.EnemyBasePos)
63              if utilityfns.find_distance((SX, SY), (EX, EY)) < 100: # this
                distance may need adjusting in future (we may also wish to consider
                the Z axis)
64                  return True
65              else:
66                  return False
67
68      # returns true if we're near enough to our own base
69      def at_own_base(self):
70          if not self.bot.botinfo.has_key("Location"):
71              return False
72          LocTuple =
            utilityfns.location_string_to_tuple(self.bot.botinfo["Location"])
73
74          if self.PosInfo.OwnBasePos == None:
75              return False
76          else:
77              (SX, SY, SZ) = LocTuple
78              (HX, HY, HZ) =
                utilityfns.location_string_to_tuple(self.PosInfo.OwnBasePos)
79              if utilityfns.find_distance((HX, HY), (SX, SY)) < 10: # this
                distance may need adjusting in future (we may also wish to consider
                the Z axis)
80                  return True
81              else:
82                  return False
83
84      # returns True if we have a location for the enemy base
85      def know_enemy_base_pos(self):
86          #print "in know_enemy_base_pos sense"
87          if self.PosInfo.EnemyBasePos == None:
88              return False
89          else:
90              return True
91
92      # returns True if we have a location for our own base
93      def know_own_base_pos(self):
94          if self.PosInfo.OwnBasePos == None:
95              return False
96          else:
97              return True
98
99      # returns True if there's a reachable nav point in the bot's list which
        we're not already at
100     def reachable_nav_point(self):
```

```
101         # setup location tuple
102         if not self.bot.botinfo.has_key("Location"):
103             # if we don't know where we are, treat it as (0,0,0) as that will
                just mean we go to the nav point even if we're close by
104             (SX, SY, SZ) = (0, 0, 0)
105         else:
106             (SX, SY, SZ) =
                utilityfns.location_string_to_tuple(self.bot.botinfo["Location"])
107
108         # is there already a navpoint we're aiming for?
109         DistanceTolerance = 30 # how near we must be to be thought of as at the
            nav point
110         if self.PosInfo.ChosenNavPoint != None:
111             (NX, NY, NZ) = self.PosInfo.ChosenNavPoint
112             if utilityfns.find_distance((NX, NY), (SX, SY)) >
                DistanceTolerance:
113                 return True
114             else:
115                 self.PosInfo.VisitedNavPoints.append((NX, NY, NZ)) # set this
                    NP as visited
116                 self.PosInfo.ChosenNavPoint = None
117
118         # now look at the list of navpoints the bot can see
119         if self.bot.nav_points == None or len(self.bot.nav_points) == 0:
120             return False
121         else:
122             # nav_points is a list of tuples.  Each tuple contains an ID and a
                dictionary of attributes as defined in the API
123             # Search for reachable nav points
124             PossibleNPs =
                self.get_reachable_nav_points(self.bot.nav_points.items(),
                DistanceTolerance, (SX, SY, SZ))
125
126             # now work through this list of NavPoints until we find one that we
                haven't been to
127             # or the one we've been to least often
128             if len(PossibleNPs) == 0:
129                 return False # nothing found
130             else:
131                 self.PosInfo.ChosenNavPoint =
                    self.get_least_often_visited_navpoint(PossibleNPs)
132                 return True
133
134     def get_least_often_visited_navpoint(self, PossibleNPs):
135         CurrentMin = self.PosInfo.VisitedNavPoints.count(PossibleNPs[0])
136         CurrentMinNP = PossibleNPs[0]
137         for CurrentNPTuple in PossibleNPs:
138             CurrentCount = self.PosInfo.VisitedNavPoints.count(CurrentNPTuple)
139             if CurrentCount < CurrentMinNP:
140                 CurrentMin = CurrentCount
141                 CurrentMinNP = CurrentNPTuple
142         return CurrentMinNP
143
144     # also performs a distance tolerance check, (SX, SY, SZ) is position of
        player
```



```
145     def get_reachable_nav_points(self, NPList, DistanceTolerance, (SX, SY,
        SZ)):
146         PossibleNPs = []
147         for CurrentNP in NPList:
148             #print type(CurrentNP)
149             #print " is the type\n"
150             #print type(CurrentNP[1])
151             #print " is the type of its 2nd element\n"
152             (NX, NY, NZ) =
                utilityfns.location_string_to_tuple((CurrentNP[1])["Location"])
153             if CurrentNP[1]["Reachable"] == "True" and
                utilityfns.find_distance((NX, NY), (SX, SY)) > DistanceTolerance:
154                 PossibleNPs.append((NX, NY, NZ))
155         return PossibleNPs
156
157     # returns true if the enemy flag is specified as reachable
158     def enemy_flag_reachable(self):
159
160         #debug
161         #print "in enemy_flag_reachable"
162
163         #if self.PosInfo.EnemyFlagInfo != {}:
164         #    print self.PosInfo.EnemyFlagInfo
165
166         if self.PosInfo.EnemyFlagInfo == {}:
167             return False
168         elif self.PosInfo.EnemyFlagInfo["Reachable"] == "True":
169             return True
170         return False
171
172     def our_flag_reachable(self):
173         print "in our_flag_reachable"
174         if self.PosInfo.OurFlagInfo == {}:
175             return False
176         elif self.PosInfo.OurFlagInfo["Reachable"] == "True":
177             print "  is reachable!"
178             return True
179         return False
180
181     def see_enemy(self):
182         if len(self.bot.view_players) == 0 or self.bot.botinfo == {}: #if
            botinfo is {}, we can't yet set anything
183             return False
184         else:
185             # work through, looking for an enemy
186             OurTeam = self.bot.botinfo["Team"]
187             Players = self.bot.view_players.values()
188             for CurrentPlayer in Players:
189                 #print CurrentPlayer
190                 if CurrentPlayer["Team"] != OurTeam:
191                     print "we can see an enemy!"
192                     return True
193         return False
194
195     def see_reachable_medical_kit(self):
```

```
196         if len(self.bot.view_items) < 1:
197             return False
198         else:
199             # look through for a medical kit
200             ItemValues = self.bot.view_items.values()
201             for CurrentItem in ItemValues:
202                 if (CurrentItem["Class"].find("Health") != -1 or
                    CurrentItem["Class"].find("MedBox")) and
                    CurrentItem["Reachable"] == "True":
203                     return True
204             return False
205 
206     def see_reachable_weapon(self):
207         if len(self.bot.view_items) < 1:
208             return False
209         else:
210             # look through for a weapon
211             ItemValues = self.bot.view_items.values()
212             for CurrentItem in ItemValues:
213                 if utilityfns.is_known_weapon_class(CurrentItem["Class"]) and
                    CurrentItem["Reachable"] == "True":
214                     return True
215             return False
216 
217     # see PositionsInfo class for comments on TooCloseForPath
218     def too_close_for_path(self):
219         if self.PosInfo.TooCloseForPath:
220             print "we are too close for path"
221         return self.PosInfo.TooCloseForPath
222 
223     # === ACTIONS ===
224 
225     def runto_medical_kit(self):
226         if len(self.bot.view_items) < 1:
227             return True
228         else:
229             # look through for a medical kit
230             ItemValues = self.bot.view_items.values()
231             for CurrentItem in ItemValues:
232                 if (CurrentItem["Class"].find("Health") != -1 or
                    CurrentItem["Class"].find("MedBox")) and
                    CurrentItem["Reachable"] == "True":
233 
                        self.send_runto_or_strafe_to_location(CurrentItem["Location'
234             return True
235 
236     def runto_weapon(self):
237         if len(self.bot.view_items) < 1:
238             return True
239         else:
240             # look through for a weapon
241             ItemValues = self.bot.view_items.values()
242             for CurrentItem in ItemValues:
243                 if utilityfns.is_known_weapon_class(CurrentItem["Class"]) and
                    CurrentItem["Reachable"] == "True":
```


jEdit - C:\Wpwin8\doc\uni\Final Year Project\CD\for zip\bodbot\movement.py

```
244                     print "runto weapon",
245                     print CurrentItem["Class"]
246 
                        self.send_runto_or_strafe_to_location(CurrentItem["Location'
247                     return True
248             return True
249 
250     # Runs to the ChosenNavPoint
251     def walk_to_nav_point(self):
252         #print "walk_to_nav_point: " +
            utilityfns.location_tuple_to_string(self.PosInfo.ChosenNavPoint)
253 
254         # have we already sent it?
255         #if not utilityfns.is_previous_message(self.bot, ("RUNTO", {"Location"
            : utilityfns.location_tuple_to_string(self.PosInfo.ChosenNavPoint)})):
256         #    self.bot.send_message("RUNTO", {"Location" :
            utilityfns.location_tuple_to_string(self.PosInfo.ChosenNavPoint)})
257             #print "sending message"
258         #else:
259             #print "already sent"
260 
261         # new version just calls the utility function
262         #utilityfns.send_if_not_prev(self.bot, ("RUNTO", {"Location" :
            utilityfns.location_tuple_to_string(self.PosInfo.ChosenNavPoint)}))
263 
264         #even newer version has a strafe check
265 
            self.send_runto_or_strafe_to_location(utilityfns.location_tuple_to_strin
266 
267         return True
268 
269     # runs to the enemy flag
270     def to_enemy_flag(self):
271         print "!!in to_enemy_flag"
272         if self.PosInfo.EnemyFlagInfo != {}:
273             self.bot.send_message("RUNTO", {"Target" :
                self.PosInfo.EnemyFlagInfo["Id"]})
274         return True
275 
276     def to_own_flag(self):
277         if self.PosInfo.OurFlagInfo != {}:
278             # was self.bot.send_message("RUNTO", {"Location" :
                self.PosInfo.OurFlagInfo["Location"]})
279 
                self.send_runto_or_strafe_to_location(self.PosInfo.OurFlagInfo["Loca
280         return True
281 
282     # runs to the bot's own base by getting a list of navpoints showing the way
        there
283     def to_own_base(self):
284         print "to_own_base"
285         DistanceTolerance = 30
286         # If we don't know where our own base is, then do nothing
287         # However, this action should never fire unless we do know where our
            base is
```

```
288         if self.PosInfo.OwnBasePos == None:
289             print "Don't know where own base is!"
290             return True
291 
292         def send_getpath():
293             print "in send_getpath"
294             if not utilityfns.is_previous_message(self.bot, ("GETPATH",
                {"Location" : self.PosInfo.OwnBasePos, "Id" : self.PathHomeID})):
295                 self.bot.send_message("GETPATH", {"Location" :
                    self.PosInfo.OwnBasePos, "Id" : self.PathHomeID}) # the ID
                    allows us to match requests with answers
296                 print "sent GETPATH"
297             else:
298                 print "GETPATH already sent"
299 
300         # if we haven't already got a list of path nodes to follow then send
            the GETPATH message
301         # to try and mitigate the problem of pathhome being cleared part way
            through this, we assign
302         # the relevant value to a variable and then use that throughout, so the
            array is checked as infrequently as possible
303         # it's not an ideal fix though!
304         if self.PosInfo.PathHome == []:
305             send_getpath()
306         else:
307             if not self.to_known_location(self.PosInfo.PathHome,
                DistanceTolerance):
308                 print "DT check failed, tailing"
309                 self.PosInfo.PathHome = utilityfns.tail(self.PosInfo.PathHome)
310                 if self.PosInfo.PathHome != []:
311                     print "tail not empty"
312                     PathLoc = self.PosInfo.PathHome[0]
313                     self.send_runto_or_strafe_to_location(PathLoc)
314                 else:
315                     send_getpath()
316 
317         #before we return, send a checkreach command about the current
            navpoint.  That way the list can be recreated if it becomes incorrect
318         if self.PosInfo.PathHome != [] and self.PosInfo.PathHome != None:
319             self.bot.send_message("CHECKREACH", {"Location" :
                self.PosInfo.PathHome[0], "Id" : self.ReachPathHomeID, "From" :
                self.bot.botinfo["Location"]})
320         print "about to return from to_own_base"
321 
322     # runs to the enemy's base by getting a list of navpoints showing the way
        there
323     def to_enemy_base(self):
324         print "to_enemy_base"
325         DistanceTolerance = 30
326         # If we don't know where the base is, then do nothing
327         # However, this action should never fire unless we do know where it is
328         if self.PosInfo.EnemyBasePos == None:
329             print "Don't know where enemy base is!"
330             return True
331 
```



```
332         def send_getpath():
333             print "in send_getpath"
334             utilityfns.send_if_not_prev(self.bot, ("GETPATH", {"Location" :
                self.PosInfo.EnemyBasePos, "Id" : self.PathToEnemyBaseID}))
335 
336         # if we haven't already got a list of path nodes to follow then send
            the GETPATH message
337         # to try and mitigate the problem of pathhome being cleared part way
            through this, we assign
338         # the relevant value to a variable and then use that throughout, so the
            array is checked as infrequently as possible
339         # it's not an ideal fix though!
340         if self.PosInfo.PathToEnemyBase == []:
341             send_getpath()
342         else:
343             if not self.to_known_location(self.PosInfo.PathToEnemyBase,
                DistanceTolerance):
344                 print "DT check failed, tailing"
345                 self.PosInfo.PathToEnemyBase =
                    utilityfns.tail(self.PosInfo.PathToEnemyBase)
346                 if self.PosInfo.PathToEnemyBase != []:
347                     print "tail not empty"
348                     PathLoc = self.PosInfo.PathToEnemyBase[0]
349                     self.send_runto_or_strafe_to_location(PathLoc)
350                 else:
351                     send_getpath()
352 
353         #before we return, send a checkreach command about the current
            navpoint.  That way the list can be recreated if it becomes incorrect
354         if self.PosInfo.PathToEnemyBase != [] and self.PosInfo.PathToEnemyBase
            != None:
355             self.bot.send_message("CHECKREACH", {"Location" :
                self.PosInfo.PathToEnemyBase[0], "Id" :
                self.ReachPathToEnemyBaseID, "From" :
                self.bot.botinfo["Location"]})
356         print "about to return from to_enemy_base"
357 
358     # returns true and sends a runto message for the provided location if the
        DistanceTolerance check passes
359     # otherwise returns false
360     def to_known_location(self, Location, DistanceTolerance):
361         if len(Location) == 0:
362             return True # even though we failed, we return true so that it
                doesn't tail the list
363         # to first point in current list, if we're not already there
364         Location0 = Location[0]
365         (HX, HY, HZ) = utilityfns.location_string_to_tuple(Location0)
366         (SX, SY, SZ) =
            utilityfns.location_string_to_tuple(self.bot.botinfo["Location"])
367         if utilityfns.find_distance((HX, HY), (SX, SY)) > DistanceTolerance:
368             print "DistanceTolerance check passed"
369 
370             print "About to send RUNTO to",
371             print Location0
372             print "Current location",
```

```
373             print self.bot.botinfo["Location"]
374
375             self.send_runto_or_strafe_to_location(Location0, False)
376             # was
377             #if not utilityfns.is_previous_message(self.bot, ("RUNTO",
                {"Location" : PathLoc})):
378             #    self.bot.send_message("RUNTO", {"Location" : PathLoc})
379             #    print "Running to " + PathLoc
380             return True
381         else:
382             return False
383
384     #not used at present (18/2/2005)
385     def inch(self):
386         # just add a bit to the x value
387         print "in inch"
388         (SX, SY, SZ) =
            utilityfns.location_string_to_tuple(self.bot.botinfo["Location"])
389         NewLocTuple = (SX + 150, SY, SZ)
390
            self.send_runto_or_strafe_to_location(utilityfns.location_tuple_to_strin
391
392     # just because something was reachable the last time we knew about it
        doesn't mean it still is
393     def expire_reachable_info(self):
394         if self.PosInfo.OurFlagInfo != {} and
            self.PosInfo.OurFlagInfo.has_key("Reachable"):
395             self.PosInfo.OurFlagInfo["Reachable"] = "False"
396         if self.PosInfo.EnemyFlagInfo != {} and
            self.PosInfo.EnemyFlagInfo.has_key("Reachable"):
397             self.PosInfo.EnemyFlagInfo["Reachable"] = "False"
398
399         self.PosInfo.TooCloseForPath = False
400
401     # === OTHER FUNCTIONS ===
402
403     # checks the previous sent message against the provided one, returning True
        if they match
404     # now replaced by is_previous_message(bot, Msg) in utilityfns
405     #def is_previous_message(self, Msg):
406     #    if self.bot.sent_msg_log == None or \
407     #    len(self.bot.sent_msg_log) == 0 or \
408     #    self.bot.sent_msg_log[-1] != Msg:
409     #        return False
410     #    return True
411
412     # updates the flag positions in PositionsInfo
413     # also updates details of bases, if relevant info sent
414     # the position of a flag is how we determine where the bases are
415     def receive_flag_details(self, values):
416         #print "f",
417         #print values["Reachable"]
418         if self.bot.botinfo == {}: #if botinfo is {}, we can't yet set anything
419             return
420
```



```
421         #print "in receive_flag_details.  Values are:"
422         #print values
423
424         #set flag stuff
425         OurTeam = self.bot.botinfo["Team"]
426         if values["Team"] == OurTeam:
427             self.PosInfo.OurFlagInfo = values
428             #print "our flag"
429         else:
430             self.PosInfo.EnemyFlagInfo = values
431             #print "enemy flag"
432
433         # now set base stuff if appliable
434         if values["State"] == "home":
435             if values["Team"] == self.bot.botinfo["Team"]:
436                 self.PosInfo.OwnBasePos = values["Location"]
437             else:
438                 self.PosInfo.EnemyBasePos = values["Location"]
439                 #print "enemy base at",
440                 #print self.PosInfo.EnemyBasePos
441                 #print "self.PosInfo.EnemyBasePos has type",
442                 #print type(self.PosInfo.EnemyBasePos)
443
444     # if the 'ID' key is PathHome then it tells the bot how to get home.
445     # we need to turn the dictionary into a list, ordered by key ('0' ... 'n')
446     # at present other IDs are ignored
447     def receive_pth_details(self, ValuesDict):
448         if not ValuesDict.has_key("ID"):
449             return
450         elif ValuesDict["ID"] == self.PathHomeID:
451             self.PosInfo.PathHome =
                utilityfns.nav_point_dict_to_ordered_list(ValuesDict)
452         elif ValuesDict["ID"] == self.PathToEnemyBaseID:
453             print "Set PathToEnemyBase"
454             self.PosInfo.PathToEnemyBase =
                utilityfns.nav_point_dict_to_ordered_list(ValuesDict)
455
456         # if there's no 0 key we're being given an empty path, so set
            TooCloseForPath accordingly
457         if not ValuesDict.has_key("0"):
458             self.PosInfo.TooCloseForPath = True
459         else:
460             self.PosInfo.TooCloseForPath = False
461
462
463     # used in validating the bot's path home or to the enemy flag
464     # if the thing has the right ID, then clear the relevant path if it's not
        reachable
465     def receive_rch_details(self, ValuesDict):
466         print "in receive_rch_details"
467         if not ValuesDict.has_key("ID"):
468             return
469         elif ValuesDict["ID"] == self.ReachPathHomeID and
            ValuesDict["Reachable"] == "False":
470             self.PosInfo.PathHome = []
```

```
471             print "Cleared PathHome"
472         elif ValuesDict["ID"] == self.ReachPathToEnemyBaseID and
            ValuesDict["Reachable"] == "False":
473             self.PosInfo.PathToEnemyBase = []
474             print "Cleared PathToEnemyBase"
475 
476     # if the combatinfo class specifies that we need to remain focused on a
        player, send a relevant strafe command
477     # to move to the provided location.  Otherwise, a runto
478     def send_runto_or_strafe_to_location(self, Location, PerformPrevCheck =
        True):
479         if self.CombatInfo.KeepFocusOnID != None:
480             Message = ("STRAFE", {"Location" : Location, "Target":
                self.CombatInfo.KeepFocusOnID})
481             if PerformPrevCheck:
482                 utilityfns.send_if_not_prev(self.bot, Message)
483             else:
484                 self.bot.send_message(Message[0], Message[1])
485 
486         else:
487             Message = ("RUNTO", {"Location" : Location})
488             if PerformPrevCheck:
489                 utilityfns.send_if_not_prev(self.bot, Message)
490             else:
491                 self.bot.send_message(Message[0], Message[1])
492                 print "have just sent",
493                 print Message
494 
495     # clean-up after dying
496     def receive_die_details(self, ValuesDict):
497         self.PosInfo.PathHome = []
498         self.PosInfo.PathToEnemyBase = []
499         self.PosInfo.VisitedNavPoints = [] # this is new
500         self.PosInfo.OurFlagInfo = {}
501         self.PosInfo.EnemyFlagInfo = {}
502 
503 
504 # This class stores details about where things are
505 class PositionsInfo:
506     def __init__(self):
507         self.OwnBasePos = None
508         self.EnemyBasePos = None
509         self.VisitedNavPoints = []
510         self.ChosenNavPoint = None
511         self.OurFlagInfo = {}
512         self.EnemyFlagInfo = {}
513 
514         # a list of nav points showing the way to various places
515         self.PathHome = []
516         self.PathToEnemyBase = []
517         self.TooCloseForPath = False # set to true if we're sent a blank path.
            Blank paths indicate that we're right next to something but can't
            actually see it
518 
```

```
1  from posh_core import *
2  import utilityfns
3
4  # the status behaviour has primitives for stuff to do with finding out the bot's
   state (e.g. amount of health)
5
6  class StatusBehaviour(Base):
7      def __init__(self, PosInfo, **kw):
8          Base.__init__(self, **kw) # Call the ancestor init
9
10         # set up variables
11         self.bot = None
12         self.PosInfo = PosInfo
13
14         # register behaviours and senses
15         self.init_acts()
16         self.init_senses()
17
18     def init_acts(self):
19         pass
20
21     def init_senses(self):
22         self.agent.add_sense("have-enemy-flag", self.have_enemy_flag)
23         self.agent.add_sense("own-health-level", self.own_health_level)
24         self.agent.add_sense("are-armed", self.are_armed)
25         self.agent.add_sense("ammo-amount", self.ammo_amount)
26         self.agent.add_sense("armed-and-ammo", self.armed_and_ammo)
27         #self.agent.add_sense("holding-enemy-flag", self.holding_enemy_flag)
28
29     def bind_bot(self, bot):
30         self.bot = bot
31
32     # === SENSES ===
33
34     # returns true if we are carrying the enemy's flag
35     def have_enemy_flag(self):
36         #print "have_enemy_flag?"
37         if not self.bot.gameinfo.has_key("HaveFlag"):
38             return False
39         else:
40             #print "have enemy flag!"
41             return True
42
43     def own_health_level(self):
44         HealthLevel = int(self.bot.botinfo["Health"])
45         #print "Our bot has health ",
46         #print HealthLevel
47         return HealthLevel
48
49     def are_armed(self):
50         if self.bot.botinfo == {}:
51             return False
52         else:
53             if self.bot.botinfo["Weapon"] == "None":
54                 print "unarmed",
```



```
55                 print self.bot.botinfo["Weapon"]
56                 return False
57             else:
58                 print "armed",
59                 print self.bot.botinfo["Weapon"]
60                 return True
61
62     def ammo_amount(self):
63         if self.bot.botinfo == {}:
64             return 0
65         else:
66             return int(self.bot.botinfo["CurrentAmmo"])
67
68     def armed_and_ammo(self):
69         #return True
70         return (self.are_armed()) and (self.ammo_amount() > 0)
71
72     # use have_enemy_flag instead
73     #def holding_enemy_flag(self):
74     #    if self.PosInfo == None or self.PosInfo.EnemyFlagInfo == {}:
75     #        return False
76     #    elif self.PosInfo.EnemyFlagInfo["State"] == "held" and
       self.PosInfo.EnemyFlagInfo["Holder"] == self.bot.bot_info["Id"]:
77     #        return True
78     #    else:
79     #        return False
80
81     # === ACTIONS ===
82
83     # none at present
```

```
1  import string
2
3  # Some utility functions
4  def find_distance(one, two):
5      (x1, y1) = one
6      (x2, y2) = two
7      return ((((x1-x2)**2) + ((y1-y2)**2))**0.5)
8
9  # takes a string of the form 'x,y,z' and converts it to a tuple (x,y,z)
10 def location_string_to_tuple(LocationString):
11     LocList = string.split(LocationString, ",")
12     if len(LocList) != 3:
13         return (0,0,0)
14     LTuple = (float(LocList[0]), float(LocList[1]), float(LocList[2]))
15     return LTuple
16
17 def location_tuple_to_string(LocationTuple):
18     LString = str(LocationTuple[0]) + "," + str(LocationTuple[1]) + "," +
       str(LocationTuple[2])
19     return LString
20
21 # returns negative if the number a represents is < the number b represents.  0
   if equal, positive if >
22 def compare_number_strings(a, b):
23     anum = int(a)
24     bnum = int(b)
25     if anum < bnum:
26         return -1
27     elif anum == bnum:
28         return 0
29     else:
30         return 1
31
32 # lists of nav points arrive as dicts with an "ID" key and keys "0", "1", ....
   "n" these need converting to lists
33 def nav_point_dict_to_ordered_list(ValuesDict):
34     del ValuesDict["ID"] #remove the ID key to leave just numbers
35     #now get a list of just keys, and sort it to use in extracting the key:value
       pairs
36     KeyList = ValuesDict.keys()
37
38     # debug
39     if ValuesDict.has_key("Reachable"):
40         print ValuesDict
41         print "-------"
42
43     KeyList.sort(compare_number_strings) #need a home-grown sort function as
       although they're strings, we don't want "10" < "2"
44
45     #now use the keylist to create an ordered list of location strings
46     LocList = []
47     CurrentLoc = 0
48     while CurrentLoc < len(ValuesDict):
49         #need to strip out the ID by including only everything after the first
           space
```


jEdit - C:\Wpwin8\doc\uni\Final Year Project\CD\for zip\bodbot\utilityfns.py

```
50         LocString = ValuesDict[str(CurrentLoc)]
51         LocString = LocString[string.find(LocString, " ") : len(LocString)]
52         LocString = LocString.strip()
53         LocList.append(LocString)
54         CurrentLoc = CurrentLoc+1
55
56     return LocList
57
58 def tail(SentSequence):
59     if SentSequence == [] or len(SentSequence) == 1:
60         return []
61     else:
62         return SentSequence[1 : len(SentSequence)-1]
63
64 # checks the bot's previous sent message against the provided one, returning
   True if they match
65 def is_previous_message(bot, Msg):
66     if bot.sent_msg_log == None or \
67     len(bot.sent_msg_log) == 0 or \
68     bot.sent_msg_log[-1] != Msg:
69         return False
70         return True
71
72 def send_if_not_prev(bot, Msg):
73     if not is_previous_message(bot, Msg):
74         bot.send_message(Msg[0], Msg[1])
75
76 def is_known_weapon_class(SentClass):
77     if SentClass == None:
78         return False
79     else:
80         if SentClass.find("goowand") != -1:
81             return True
82     return False
```