An Integrated Development Environment for Behaviour Oriented Design

Philip J. Richards¹

May 13, 2004

¹This 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.

Signed:
Abstract

Bryson's POSH (Parallel, Ordered, Slip-stack Hierarchical) Artificial Intelligence system has been ported to the scripting language Python, but has been lacking a development environment. This project is an attempt to develop an integrated development environment for both POSH and the new PyPOSH systems, in order to encourage uptake and testing by new users.

It introduces POSH IDE, which provides a useful editor but falls short of integrating with the POSH systems.
An Integrated Development Environment for Behaviour Oriented Design

Submitted by Philip J. Richards

COPYRIGHT

Attention is drawn to the fact that 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 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 Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed.....................................................................
Acknowledgements

I would like to first and foremost thank my tutor and the person who made this possible — Dr. Joanna Bryson, the author of BOD. Thank you to Emily Korvin and Mark Wood who graciously donated their time to help me test POSH|IDE. Thanks to Mum and Dad and Chloë, and finally thank you to my girlfriend Claire for managing to be there without being here.
Contents

1 Introduction 2

2 Background 5
2.1 Introduction .................................................. 5
2.2 POSH AI ......................................................... 6
2.2.1 Overview .................................................. 6
2.2.2 The BOD Software ...................................... 6
2.2.3 The BOD Methodology .................................. 10
2.2.4 Summary ................................................... 12
2.3 Integrated Development Environments ..................... 12
2.3.1 Overview .................................................. 12
2.3.2 Existing IDEs ............................................. 13
2.3.3 Conclusions ................................................. 13
2.4 Supporting and Similar Software ......................... 14
2.4.1 PyPOSH and PoshEdit .................................... 14
2.4.2 Languages ................................................ 14
2.4.3 GUI Frameworks .......................................... 15
2.5 Summary ....................................................... 15

3 Requirements Analysis and Specification 17
3.1 Introduction .................................................. 17
3.2 POSH AI Components ....................................... 19
3.3 Integration with PyPOSH AI ................................. 21
3.4 Summary ....................................................... 21

4 Design Decisions 23
4.1 Introduction .................................................. 23
4.2 The User Interface .......................................... 23
4.2.1 Introduction ............................................. 23
4.2.2 The Design ............................................... 24
4.3 Internal Structure ........................................... 25
4.4 Summary ....................................................... 25
5 Implementation 27
5.1 Introduction ........................................... 27
5.2 Decisions .............................................. 27
  5.2.1 Development Methodologies .................. 27
  5.2.2 Programming Language ....................... 28
  5.2.3 Windowing Toolkit ............................. 28
  5.2.4 Rapid Application Development (RAD) .... 28
5.3 GUI Elements Used ................................. 29
  5.3.1 Summary ......................................... 30
5.4 Development ......................................... 31
  5.4.1 First Steps ...................................... 31
  5.4.2 Class Arrangement ............................ 32
5.5 Summary ............................................... 35

6 Testing 36
6.1 Introduction .......................................... 36
6.2 Code testing .......................................... 36
6.3 Requirements Testing ............................... 37
6.4 Usability Testing .................................... 37
  6.4.1 The First Tester: Dr. Bryson ................. 37
  6.4.2 The Second Tester: Mark Wood .............. 39
  6.4.3 The Third Tester: Emily Korvin .......... 41
  6.4.4 Observations .................................. 41
  6.4.5 Improvements Made ........................... 42
  6.4.6 Summary ....................................... 42

7 Discussion and Conclusions 43
7.1 Introduction .......................................... 43
7.2 Achievements ........................................ 44
7.3 Reflections .......................................... 44
7.4 Future Work ......................................... 45
  7.4.1 Introduction .................................. 45
  7.4.2 Problems, issues and bugs .................. 45
  7.4.3 New features ................................ 45
  7.4.4 Game Creation or Modification ............ 46
7.5 Summary ............................................. 46

A Code listing 50

B Requirements Specification 51
  B.1 Non-functional Requirements .................. 51
  B.2 Functional Requirements ....................... 51
    B.2.1 Necessary .................................... 51
    B.2.2 Desirable .................................... 52
B.3 User Requirements ........................................ 53
B.4 System Requirements ...................................... 53
B.5 Requirements Testing ................................. 53
Chapter 1

Introduction

Artificial Intelligence (AI) has been the subject of research since the early days of computing (Brooks, 1991; Humphrys), by numerous groups of people. There has been no single, unified effort to develop AI, as researchers have constantly disagreed over the method with which AI development should be done. What we have today is a plethora of AI systems each with a unique take on how AI should be done, for example ACT-R (Anderson, 1996), Soar (Laird et al., 1987), the Subsumption Architecture (Brooks, 1991) and POSH-BOD (Bryson, 2001, 2003b) to name but a few.

One of the major reasons for the different architectures is the fact that the AI developers have historically had different aims; some are working on expert systems, some are working on computer vision and so on. Although the aims of AI are now varied and fragmented, a common, early aim was to achieve intelligence on a par with an average adult human. The success of the work toward this aim has been limited — and now commonly accepted as too difficult a challenge to begin research on (Brooks, 1991; Humphrys). Some current aims of AI research are:

- Games — enemy and opponent AI, and creating believable non-player characters with emotions (Gratch and Marsella, 2001); creating believable behaviours to challenge and immerse the player.
- Speech recognition — to aid and increase the speed of data input, and reduce the need for keyboards to improve accessibility.
- Natural language — another traditional, long term goal of AI research is to make a computer understand natural language. This could work well when partnered with speech recognition — think ‘Data’ from Star Trek, or ‘Holly’ from Red Dwarf.
• Computer vision — image reconstruction, motion tracking, facial recognition etc.

• Expert systems — another early aim of AI research.

The long-standing question of which is the most effective approach to AI has not been conclusively answered (Brooks, 1991; Gordon and Logan, 2002; Bryson, 2001). Clearly there is no single approach that is ‘best’ for every application — a system that has been designed to recognise natural language will not perform well in a gaming environment. However, this is why a modular approach to AI is useful (Bryson, 2001; Brooks, 1991), in that it allows the same code base to be used for different applications. Bryson has developed a new approach which addresses some of the flaws present in current architectures, such as over extensive searching, and trivialised action primitives (see Bryson, 2001) — her approach is to use a design methodology (Behaviour-Oriented Design) together with an AI system of her own (the Parallel, Ordered, Slipstack Hierarchical (POSH) engine). Some interesting test results were recorded by Bryson, notably simulated rodent behaviour in Tyrell’s Simulated Environment (SE) (Tyrell, 1993), which indicated that the new BOD/POSH method performed better than Tyrell’s own agents (Bryson, 1999).

The POSH engine has since been ported, from Common Lisp, to the object oriented scripting language Python by Kwong (2003). This was done so that developers who are uncomfortable with using Lisp would still be able to make use of the POSH system (Kwong, 2003). Unfortunately the Python port (PyPOSH) does not have any kind of development environment (unlike the original system, which has one written by Bryson). The original development environment works only with the original POSH engine, so a version that would work with both versions would be preferable.

It is now accepted that Integrated Development Environments (IDEs) can increase productivity when developing software. They can give further insight than simple tools such as text editors into why something is or is not working, and they can give insight into the internal state of the software being developed. Because AI systems and agents are types of software it follows that tools such as a specialised IDE will aid development.

Although the BOD systems have shown promising results, uptake by new users has been slow, and the main reason for this is the lack of a development environment — this project is an attempt to remedy this by creating one. An early aim of the project was to test the IDE by either extending Kwong’s work (Kwong, 2003, p.49) and enhancing the Unreal
Tournament AI-controlled bot, or by creating a sparring game/simulator from scratch; however the lack of time meant that these rather ambitious goals have not been met. Instead the outcome is a usable and portable editor for POSH plan files, even though no integration is in place to allow debugging.

This project builds mainly on the work done by Bryson in creating BOD, and as such it is not an attempt to get embroiled in the debate over which approach is superior — instead it is an attempt to make it easier for others to do so by allowing development and testing of POSH/BOD agents to be done more easily and hence comparisons to be made with other methods. Therefore this project will not attempt to explain in detail the differences between the various AI architectures and approaches, this project is firmly focused on providing a development and testing environment for POSH AI only. POSH AI so far shown promising results, this should help others develop agents for it.
Chapter 2

Background

2.1 Introduction

One of the goals of AI research is to create human-level AI (John E. Laird, 2001), although it has been argued that true human level AI is not required (Brooks, 1991), as AI-based applications tend to have a very small domain in which intelligence is actually used and so full human-level AI would be effectively useless — either because human-level intelligence would be too much, or because the application required AI more powerful than a human. However, most agree that human-level intelligence is a goal worth working toward (Nau, 1999).

Indeed, games are increasingly used as test beds for AI research (Gordon and Logan, 2002). The rules of the real world are very difficult to work with in a simulated environment as they are so complex; games provide the developer with a fixed set of rules — which can still be non-trivial and so allow research to move forward whilst staying within the limits of the game engine (Schaeffer, 2001). Laird is currently working towards a game (‘Haunt 2’, a modification to Unreal Tournament) (Bachelor et al., 2003) where the player’s actions have a significant effect on the behaviour of the computer-controlled characters. The aim of Haunt 2 is to create complex AI characters that are critical in determining the outcome of the game (Laird et al., 2002; Laird, 2002).

Games are mentioned here as a motivation for AI research, and also because they have proven to be useful environment in which to test AI-controlled agents. What this review concentrates on, though, is the work done by Bryson in developing Behaviour Oriented Design.
2.2 POSH AI

2.2.1 Overview

The POSH-BOD (Parallel, Ordered, Slip-stack Hierarchical Behaviour Oriented Design) system this dissertation was made for was created by Dr. Joanna Bryson as a PhD thesis, see Bryson (2001). Bryson made two contributions with her PhD — the BOD software system used to control agents, and the BOD agent development process.

Note that in this dissertation the terms ‘BOD’ and ‘POSH’ are more or less interchangeable, although strictly speaking ‘BOD’ is the overall term encompassing both the software and the methodology, and ‘POSH’ is an adjective used to describe the type of reactive plans used by the software.

2.2.2 The BOD Software

Introduction

The BOD AI software contains the following key elements:

**POSH Reactive Plans** These control the action selection mechanism. They determine what the agent should do according to the current state of the environment and the agent’s senses.

**Behaviour modules** These are treated as ‘primitives’ in BOD AI, but unlike Brooks’ Subsumption Architecture (Brooks, 1991), Bryson’s behaviours are semi-autonomous, and can run in parallel to an extent.

**Specialised learning** Although BOD recommends careful planning over extensive learning, it does support a learning mechanism.

These are explained more fully in the following paragraphs.

**Basic Reactive Plans**

The POSH AI software builds heavily on the powerful concept of reactive plans. Reactive plans are a set of production rules ordered by priority, and they contain a notion of the state of the environment. Each production rule consists of a precondition and an action primitive (a behaviour module), eg. Bryson’s own example using a monkey agent (Bryson, 2001, p.32):
This shows that if the monkey has a banana and it is peeled, she (or he) will eat it. If she has a banana that is not peeled, she will peel it. If she does not have a banana, she will get one. If she is not hungry, she will not do any of the above. This compact notation provides a behaviour for a relatively large set of input states, and reacts very quickly to changes in the environment.

The lowest priority rule does not need a precondition, as it will always fire if none of the other, higher priority preconditions are met when the monkey is feeling hungry.

The left hand side shows the plan’s precondition, ‘have hunger’. So, while the monkey is hungry, the reactive plan will be triggered. The behaviours are on the right of the arrows in the brackets, and most reactive plans (as with this one) have the top rule as the special behaviour ‘goal’.

**POSH Plans**

The POSH plans are based on the above reactive plans, but they have some additional features. The first difference between basic reactive plans (BRPs) and POSH plans is that POSH plans are hierarchical, which means a plan can contain elements that are themselves plans. Bryson states (see Bryson, 2001, p.33) that the problem here is that if plans contained elements that were also plans, where would the agent start? And how would the agent continue to run (ie, be a ‘complete’ agent) once the first plan terminated? Her solution was to create a type of plan specifically for this purpose: the Drive Collection (DC).

Please note that I will use the term POSH ‘component’ to mean one of the following: drive collection, action pattern or competence (both introduced shortly); whereas Bryson uses the term ‘element’. I do this because drive collections and competences also have ‘elements’, so using this term for both can create confusion. To clarify — ‘component’ for drive collections / competences / action patterns; ‘element’ only for drive elements or competence elements.

The drive collection has the following properties additional to those of a BRP:
DC will not terminate  A DC will never terminate, unlike a BRP, allowing an agent to be ‘complete’ and run indefinitely. A complete agent is one that is not part of a multi-agent system.

Realtime  A DC has a ‘realtime’ property — timing can be measured in real time units (minutes, seconds etc.), else it is measured in cycles.

POSH Root  Each DC drive (or drive element) has a ‘root’, which is the name of the action primitive, competence or action pattern that must be triggered when the drive element fires.

The slip-stack  Whilst the system is running the DC will have a variable pointing to the currently active element (CAE). If the currently active element then triggers a child POSH component (action pattern or competence) the CAE variable will then point to the child. From Bryson (2001, p.65): “The slip stack hierarchy defeats the overhead of ‘hierarchy bottleneck’ that Maes (1990) warns of. For any cycle of the action selection, only the drive collection itself and at most one other compound POSH element will have their priorities examined.”

Drive element frequency  Each drive element can have maximum frequency specified, either by setting the amount of time (seconds, minutes etc.) between visits in a realtime DC or by the number of cycles of the action selector.

A POSH agent can support just one drive collection, and because the DCs do not terminate another form of reactive plan is required to interface with the behaviour modules. Bryson uses a ‘competence’, another modified BRP for this task.

The competence focuses on a certain set of actions suitable for a particular task, and as such differs from the DC which covers (albeit in a high-level manner) all actions of the agent. Little modification was done to the specification to that of a BRP to create the competence specification — the only thing different between them is the fact that a competence has an optional number retries before it is removed from the scheduler. This was added to remove the possibility of a failing competence resulting in an infinite loop.

Also worth noting is the fact that competences return a value: ‘true’ if the competence terminates due to the goal trigger firing, and ‘false’ if no step can fire.

The remaining POSH component is an action pattern (AP). This is the simplest of the components as it is a sequence of actions that can be
ordered (or that can run in parallel, however Bryson states that the usefulness of this feature has not yet been proven).

**Behaviour Modules**

Behaviour modules are coded as software objects and are responsible for the perception and action of a BOD agent, as such they are often referred to as ‘primitives’ in this document and by Bryson. Perception is the process of taking information from the environment and interpreting it so that the agent finds it useful. Action primitives are responsible for taking information from the agent and changing the environment in some way. Behaviours are also responsible for any learning (see below) that the agent must do.

Behaviour modules can have a state or be stateless; and can run in parallel to a certain extent — however this depends on the plan used. For example, a robot could have a ‘walk’ primitive, and a ‘check battery’ primitive that is called by the DC every two minutes — the DC is able to call the ‘check battery’ primitive even when the robot is walking because the ‘walk’ behaviour had not terminated when the robot checked its battery.

**Specialised Learning**

Bryson recommends using learning as little as possible (Bryson, 2001, p.95), and that it should only be used when ‘programming control is otherwise prohibitively complex’. This means that BOD agents will be as reactive as possible in dynamic environments. However learning can be used if, for example, the triggers of a set of competence elements become too convoluted, using learning it is possible to reduce the triggers using data about new perceptual state.

Bryson states that learning and memory states can be categorised in the following ways:

- Control state
- Deictic variables
- Perceptual and motor memory
- Meta-state

Control state is the currently active behaviour, or the paths that are currently open to the agent. BOD AI holds control state in the plan and the program holds a pointer to the next step.
Deictic variables are representative of things in the environment, and refer to something to which the agent is currently attending. They allow the system to generalise over cases where plans (held in control state) may operate; and can remove the need to keep a full representation of the environment in memory — eg. in Blocks World, rather than keeping track of 20+ blocks the agent could hold variables such last ‘currently-held-block’, or ‘last-block-seen’. Bryson states that deictic variables should be used in place of control structure if the control structures would otherwise become replicated, as this is unreliable and cumbersome and hence hard to engineer.

Perceptual and motor memory is a store where information can accumulate; information received through perception can be stored for any length of time between a split-second and the entire lifetime of the agent. Perceptual memory is needed because each cycle can only reveal limited information about the environment. Motor control may need memory if it cannot be fine-tuned — this information would be stored in behaviour modules if used.

BOD does not support meta-state learning, as Bryson believes that it is unnecessary when an agent is engineered instead of ‘growing’ as a biological organism would.

Summary

The POSH-BOD system builds heavily on reactive plans, which themselves are robust, easy to engineer and are reactive to dynamic environments. Bryson has added extra features to reactive plans, and although the changes themselves are fairly simple, the extra functionality gained is significant. Allowing the reactive plan to trigger other reactive plans or action primitives has allowed for nesting and hence non-trivial action-selection.

Usually, use of nesting (components) would entail slowdown (the hierarchy bottleneck), however POSH-BOD defeats this with the ‘slip stack’.

2.2.3 The BOD Methodology

Introduction

Together with a software system Bryson introduced a development methodology specifically for creating AI controlled agents. The BOD methodology borrows from software engineering methods and focuses on iteratively extending (increase the complexity, or make it more interesting)
and fixing problems (that may have been introduced by extending) with the POSH plan.

**Iterative Design**

Bryson (2001, p.34) states that the most important fact about iterative design (and BOD) is that specifications will be changed — ‘you never fully understand a problem until you try to solve it’. Trying to fit a program around the first set of specifications will likely end up with a program with too many fixes so that it is ungainly, and have some underdeveloped areas. Programs like this are hard to debug, and become harder to fix as more and more fixes are added. Iterative design combats this by allowing the specifications to be revised and changed as development progresses.

BOD favours simplicity over complexity, and it sets time aside for revising the specification to ensure that the agent is as simple as possible. BOD takes into account and even encourages the fact that change will continue to happen throughout the development process.

**Favouring Simplicity**

The main design principle of BOD is ‘when in doubt, favour simplicity’ (Bryson, 2001, p.34). Provided no functionality is lost, it is better to use an action pattern than a competence, a single action primitive than an action pattern, and control state rather than variable (deictic) state in a behaviour.

**How, When, What?**

BOD comprises three main stages:

- **Saying How**
- **Saying When**
- **Choosing What**

‘Saying how’ involves creating a behaviour library, which means the action and sense primitives. You need to have an idea of what the agent will be doing, and split this up into different activities. For example a robot could ‘walk’, ‘check batteries’, and ‘destroy humanity’; a behaviour would need to be created each of these.
'Saying when’ translates to ‘under what circumstances’. This is the problem of action selection, and is handled by the POSH components and the plan we give them.

‘Saying what’ is not so clear cut. It is the problem of deciding how much detail to specify when creating the behaviours, and deciding how lenient we can be about action selection. Using the robot example, we can tell it to ‘destroy humanity’ assuming that its corresponding action primitive has all the relevant information and commands to allow our robot to do so; or we could bring more detail into the plan. Instead of saying ‘destroy humanity’, we would issue instructions one by one — but again we have the exact same problem of deciding how much detail to give the robot in each of these instructions. The problem is deciding where to ‘put’ the complexity, and deciding how much to split actions into behaviours and instructions. Another problem we have here is interfacing between ‘how’ and ‘when’. BOD behaviours have different interfaces for each action they perform, so the ‘walk’ behaviour may have an interface for ‘walk quickly’, ‘saunter’ and ‘skip’ — ie., an interface for each action it can perform.

2.2.4 Summary

BOD is a methodology based on iterative design, but tailored for agent creation. It does not shirk away from common problems and it encourages regular revisions to the specifications of the system.

‘The trick of agent design is to choose a set of ‘whats’ that make the ‘whens’ and ‘hows’ as easy to build as possible’ — Bryson (2001, p.26).

2.3 Integrated Development Environments

2.3.1 Overview

It is generally accepted that a good development environment is preferable for project development and software engineering to a simple text editor and command prompt (Shneiderman, 1997). It follows, then, that a functional and easy to use development environment would be useful to support the BOD methodology for designing intelligent agents.

Integrated development environments (according to Shneiderman) support:

• Faster development of code
• Easier maintenance of code
• Usability and unit testing

Obviously, these apply to programming languages, not AI systems but the advantages are very similar: faster development and easier testing.

2.3.2 Existing IDEs

Current, existing development environments include:
• Anjuta
• KDevelop
• Eclipse
• MS Visual Studio
• Borland JBuilder
• Tcl/Tk Soar Interface (TSI)
• Bryson’s own frontend for POSH AI

Most of the above (with the exceptions of TSI and the POSH frontend) are development environments for programming languages. It would be possible to create a plugin for Eclipse for BOD — the Eclipse website states “Eclipse is a kind of universal tool platform - an open extensible IDE for anything and nothing in particular”. Because the Eclipse program is written entirely in Java, the BOD plugin for Eclipse would also have to be written in Java where Python is preferred, for consistency and ease of communication with the PyPOSH system.

The TSI includes some useful features for Soar such as stepping — ie, controlling its actions and the speed at which it executes, so that the human observer can read the states and outside stimuli before the agent acts and the information changes. Bryson’s frontend also supports stepping of the AI engine, however it only works for the Lisp version of BOD, when an IDE that works with the Python version is the aim of this project.

2.3.3 Conclusions

There are already several packages available that support software development, and at least two that support AI engines — including one
that already supports the Lisp version of BOD. Creating an IDE for Py-POSH AI should encourage people to create agents and test the BOD system, as many people are not comfortable working in Lisp.

2.4 Supporting and Similar Software

2.4.1 PyPOSH and PoshEdit

The Python port of POSH, PyPOSH, was completed by Kwong (2003) as an MSc project. A PyPOSH agent has been successfully implemented and tested with the game Unreal Tournament (UT), using the Gamebots UT modification. (Gamebots allows external programs to interface with UT using standard network sockets.)

PoshEdit was developed by D. Wright (reference unavailable) in 2002 as another MSc project and was intended for use as an editor for the '.lap' files that POSH uses. Unfortunately, this program has some problems with file parsing — sometimes even small files take a long time to parse, and at times it is unable to parse them at all even if the syntax is correct (found after testing by the author of this document).

The features of PoshEdit were implemented in this project, but the development was done from scratch.

2.4.2 Languages

These paragraphs reflect the decisions made when deciding with which language to develop POSH IDE.

The PyPOSH software was developed in the object-oriented scripting language Python, as was the POSHBot UT (Unreal Tournament) agent. One option was write the IDE in Java as a plugin for the Eclipse project, mentioned previously. However, this may have proven to be overly complex, so POSH IDE was written in Python (together with a GUI framework) for the sake of continuity with PyPOSH.

Python, Perl, Rexx and Tcl (all scripting languages) have been compared to C, C++ and Java to find how memory usage and speed time vary across the languages (Prechelt); Python emerged as the fastest scripting language in several tests and faster than Java in some tests, although C/C++ are faster and more efficient in terms of memory usage. Python was therefore a good choice of language to use for porting POSH; it will also be used to develop the PyPOSH IDE.
One other consideration taken into account was development time — as a loosely typed high-level language, Python requires far less time than strongly typed language to develop equivalent programs. The trade-off is speed and memory consumption — high-level languages are slower and use more memory than C, for example.

Although it was assumed even before researching languages that Python would be the best choice, the above findings only back up the decision.

### 2.4.3 GUI Frameworks

There are several GUI frameworks that work with Python such as:

- wxWindows (as wxPython)
- QT (as pyQT)
- gtk (or, the Gnome Toolkit; as pyGTK)
- win32
- Tk (as Tkinter)

Because both POSH and PyPOSH work on most platforms, it would make sense for the IDE also to be multiplatform. Therefore, gtk and win32 are not viable solutions as win32 is not multiplatform and gtk has not been cleanly implemented on Windows. This leaves QT, Tkinter and wxWindows as possible solutions — however, the QT license requires a fee to develop software for the Windows platform (see Trolltech), whereas wxWindows and Tk are cross-platform and completely free.

The PyPOSH GUI was written using wxWindows, therefore the framework of choice for this project will be wxWindows again for a sense of continuity.

### 2.5 Summary

The BOD methodology is based on the iterative design process, borrowed from software engineering disciplines and comprises a task decomposition followed by the iterative steps of developing the plan. The BOD software is based on reactive plans but with a hierarchical structure. This review has shown that Bryson’s BOD software system and development methodology have been proven to work well, however uptake has been low which has been attributed to the lack of a good de-
velopment environment. This project is an attempt to rectify this by creating one — POSH|IDE.

From this review and from the problem description (shown in the next chapter) the aims of this project could be produced:

- To create an integrated development environment for POSH AI supporting the BOD methodology — ‘POSH|IDE’.
- To make the above as usable and accessible as possible.
- To integrate the new system with PyPOSH AI and / or POSH AI to allow debugging and testing of agents.
- To test POSH|IDE by creating a software agent to run in a gaming environment.

These are ambitious aims, and not all could be fulfilled as the following chapters will show. The next chapter will begin the documentation of the development of POSH|IDE by introducing the requirements and how they emerged from the above aims.
Chapter 3

Requirements Analysis and Specification

3.1 Introduction

The requirements for the implementation were taken from various sources, including the previously shown project aims. Firstly, the formal project description was as follows:

Title:
A GUI Development Environment for Reactive Plans

Description:
Reactive plans are an excellent way to organise the behaviour of complex artificial agents, such as VR characters or simulated animals. A developer provides a concise, robust plan which the agent can then apply in a large number of diverse, dynamic situations. However, visualising, coding and debugging reactive plans can be tricky.

... The project will require creating an improved view of the complete plan, connecting the plan execution software, connecting the plan execution software to the planner so that execution can be observed in real time. Additional features should include some of the following: connecting to an editor (preferably emacs) for editing source code for plan primitives, building demonstration AI characters in a graphical environment, user testing, a well specified API, logging facilities so that behaviour can be re-run out of real-time for debugging and analysis, evaluations and possibly improve-
ments on the POSH plan representation. The final writeup may be directed toward either industrial or academic standards, depending on the student's career interests.

The POSH planning system, though developed by Dr. Bryson, is being used and evaluated at several other universities. It is expected that the outcome of this project will be made publicly available from the Bath AI web pages, and will be used by many students and researchers.

Bryson (2003a)

This description provided several desirable requirements — that is, features that would be nice to have, but not essential. It does not, however, make it clear which of these would be most desirable; so after further discussion with Dr. Bryson, it emerged that the most important requirements were:

- To create an editor for the plan files the POSH agents use
- To connect to a running POSH agent to receive and display feedback, ie. ‘integration’ with a running agent

with the possibility of adding the other features (logging, linking with emacs etc.) if enough time was available.

These were both very high-level requirements that needed to be more fully specified before programming commenced. In particular the first, ‘create an editor’, is extremely open and it was this that needed to be further investigated.

The majority of the requirements for the IDE were deduced from reading Bryson (2001), as it contains a detailed description of the features of the POSH (and hence PyPOSH) systems. The POSH systems have four components (see the literature survey) which, for an editor or IDE to be useful, would need to be fully supported. So the functional requirements of the editor can be briefly expressed as the following:

- The program needs to load and correctly parse all features of .lap behaviour files, and hence the features of POSH AI
- Should allow all POSH AI features to be edited, deleted and created

These are the requirements at the most basic level. (For the requirements specification see the Appendix.) To meet these requirements relatively little needed to be known about the internal workings of PyPOSH/POSH systems. That is, most of the information needed to build the editor was taken from the .lap file specification (see below) and Bryson’s thesis.
The file specification, taken from ‘posh-script.lisp’, a file available in Bryson’s BOD code (added verbatim):

```lisp
name :: [a token]
sol-time :: (<time-unit> #)
time-unit:: minutes, seconds, hz(1/sec), pm(1/min), none(not realtime)
should maybe add hours, days, microseconds?
goal :: (goal <ap>)
Competence :: (C <name> <sol-time> <goal> <other-elements>)
where time is for timeout, Ymir-like
Drive-collection :: (DC|RDC <name> <goal> <drive-elements>)
RDC for real-time systems
other-elements :: (elements (<comp-element>)+)
inner paren is priority level
drive-elements :: (drives (<drive-element>)+)
inner () is priority
comp-element :: (<name> (trigger <ap>) <name> [#])
2nd name is an action, followed by optional # of retries
drive-element :: (<name> (trigger <ap>) <name> [<sol-time>])
drive name, trigger, POSH root, [freq. for scheduling]
ap :: (<act|(sense [value [predicate]])>*
default value is t, default predicate is eq
"act" can be an action primitive or a composite
Action-pattern :: (AP <name> <sol-time> <ap>)
where time is for timeout, Ymir-like
```

Even though investigating Bryson’s papers gave a great deal of insight into the way the POSH architecture works and the properties of its components, it was still unclear how to transfer these properties into an editor. Because of this, rapid prototyping and iterative programming methodologies were used which allowed the functional requirements to emerge with each prototype made, together with overall understanding of the BOD system.

### 3.2 POSH AI Components

Bryson’s papers gave the following understanding of the PyPOSH/POSH AI plan elements and what is required of POSH|IDE. The following attributes of a BOD/POSH agent must be modifiable by POSH|IDE:

- **Drive Collection(s)**

  1. **Name** — each drive collection has an identifier. This is a string.

  2. **Goal** — the goal of a drive collection is a primitive (usually a sense) that the agent is trying to achieve. This is a list of primitives.

  3. **Realtime** — whether the timing of a drive collection is based on CPU cycles, or units of time. This is a boolean.
4. Drive Elements — each drive element has several properties:
   - One of which is created and updated at run time and so can be discounted
   - Priority (an integer — priorities must be consecutive, more than one drive element can have the same priority).
   - Name (a string).
   - Trigger (a list of primitives).
   - POSH root (which is the name of an action, a competence or an action pattern).

   • Action Patterns
     1. Name — as with drive collections, each action pattern has a string identifier. This is the POSH root that a drive element (see above) will refer to.
     2. Timeout — the length of time before being removed from the scheduler (an integer and a time unit (minutes, seconds, etc)).
     3. Elements — the list of primitives (normally actions) to perform, in order.

   • Competences
     1. Name — as with drive collections and action patterns, each competence has a string identifier. This is the POSH root that a drive element will refer to.
     2. Timeout — the length of time before being removed from the scheduler. Same as action pattern timeout.
     3. Goal — a list of primitives (usually senses) that must be true for the competence to finish.
     4. Elements — the competence elements are similar to drive elements in that they must have a priority, a name and an action; but differ in that they do not contain timing information, instead they have a number of retries.

The system should also have the ability to load an existing set of behaviours (a behaviour library), it should be able to load existing .lap plan files as well as create new ones and it should be able to save files to the .lap specification given previously.
3.3 Integration with PyPOSH AI

However, implementing the system to fulfil just these would mean that the program would be an editor of the behaviour files, and not a true IDE in the sense that there is no integration with the AI system. Integration should involve the following:

- Ability to spawn the PyPOSH engine.
- Ability to display (in realtime) information about a running agent; ie. which element is triggered / fired.
- Modify PyPOSH agent to provide layered disclosure (Riley et al., 2001) — similar to previous item, but information would be logged instead of displayed in realtime, and with multiple ‘levels’ of data which are user-selectable.
- Ensure the realtime feedback API is able to receive data from the Dr. Bryson’s lisp version, to allow compatibility.

Spawning the PyPOSH engine is a basic requirement that was trivial to fulfil on its own, but it serves merely as a convenience.

Its use would be complemented significantly if the functionality to display feedback in realtime was implemented, but this would require creating an API (of sorts) for the two programs to communicate and has not been completed due to time constraints. This is definitely a possibility for future work and would significantly improve the implementation; the development environment would be integrated with PyPOSH AI to a greater extent.

Displaying feedback in realtime would require some modifications to PyPOSH, namely informing the API of POSH|IDE of events such as a competence (or some other component) being fired. The API need not be very complex, but the message types would have to be considered if the original lisp version of POSH AI is to be supported.

To display the feedback, the tree structure that is central to POSH|IDE could be utilised to provide consistency with the rest of the system — the tree is central to the use of POSH|IDE.

3.4 Summary

This chapter has introduced the basic functionality in POSH|IDE — what is present, and what was left out. The requirements detailing the
editor portion are distinctly more detailed than the others as the iterative development process used allowed requirements to emerge with each iteration and prototype made. Because development did not reach integrating the PyPOSH software and POSH|IDE, the requirements did not become very detailed.

The next chapter outlines how the initial requirements led to some high-level design decisions about the user interface and the underlying structure of POSH|IDE.
Chapter 4

Design Decisions

4.1 Introduction

Although using iterative and rapid prototyping techniques meant that changes were frequently made to the system whilst programming, some high level design decisions needed to be made before starting to code. The first decision was to create an initial design of the interface, and base the rest of the system around it; designing the interface first meant that it was relatively easy to keep the interface synchronised with the underlying system. Because the structure of POSH AI is fundamentally a tree (comprising a single drive collection and zero or more competences and action patterns (see Bryson, 2001), it made sense to make the main part of the graphical user interface (GUI) reflect this, to give the user greater understanding of how their agent will decide on which behaviours to perform. On either side of the tree (which is expandable and collapsable) there is a panel containing further information.

4.2 The User Interface

4.2.1 Introduction

The first few prototypes of the POSH IDE interface were made using two different Rapid Application Development (RAD) tools; firstly ‘Boa Constructor’ (a play on the programming language name Python) and then ‘wxGlade’. Both these are available in the public domain, Boa was used first as it was the first I found. The first few prototypes were discarded as it became clear that Boa only supports (at time of use) absolute positioning of GUI elements — that is, positioning by pixel and
not by percentage of window size. This is a problem with cross-platform programming as there is no guarantee that the size of one platform’s GUI element will be the same as another’s, so absolute positioning will cause problems. As the project uses the wxWindows development library which utilises the native windowing toolkits of the platform the system is running on, relative positioning of GUI elements must be used instead. wxGlade was the only (free) RAD tool for wxWindows that supports relative positioning, so development restarted with that.

4.2.2 The Design

As prototypes were created and discarded, it became gradually clearer how the interface should look and how to interact with it. The requirements state that it must be possible to edit, view, create and delete the drive collection, competence(s) and action pattern(s) of the agent. These are integral parts of the POSH agent so the decision was made to include an editable, constantly viewable list of each type of component on the left of the tree displaying the agent’s behaviour layout. If the contents of any of the lists change, the internal representation of the system is changed to reflect this, and vice versa — that is, if for example a new competence is created in the list of available competences, a competence object is created in the internal structure.

The third (and final) main panel contains context-sensitive information about what is currently selected in the tree. If nothing is selected, nothing is displayed; if the current drive collection is selected on the tree, the panel updates to show information relevant to the drive collection. This allows information to be displayed to the user the instant a node is selected. This panel is also where most of the editing takes place, and will be referred to as the ‘dynamic panel’ or ‘edit panel’ for the remainder of this document. As well as showing the user information about the selected item, the edit panel allows the information to be changed (hence its name). The layout of each panel was not fully designed before implementation; it is possible this has affected usability (see the usability tests in 6.4 for more information) as it was felt that the area reserved for displaying available primitives was a little small.

One decision that was made before implementation began was to keep the edit panels free from clutter — mainly confirmation buttons such as ‘OK’, ‘Done’ or ‘Apply’ buttons. Instead, when the user changed some data on the panel such as changing the name of a POSH root of a drive element, it should immediately update the internal representation of the system and the external representation (ie, what the user can see). In the case of changing the POSH root field (which is a drop-down list
which can let users input their own name or choose from a selection),
whenever the text in the field changes, the drive element instance that
the panel relates to should be updated, and so should the tree node that
represents the element.

During development, as more and more features were added it became
difficult to maintain a consistent interface without creating clutter. The
usability testing (see next chapter) shows that some issues remain with
this, so perhaps in retrospect providing all editing functions for the cur-
rent selected item in one panel was suboptimal.

### 4.3 Internal Structure

Again because of the methodology used, only very high-level design of
the system was done before implementation started. I decided to make
classes to represent each of the POSH components, and to use Python
dictionaries (or hash maps) as a convenient method of storing (pointers
to) instantiations of these classes.

The component classes contain all the properties of the POSH compo-
nents. A class was created for each of the following:

- Drive collection (`DriveCollection`)
- Competence (`Competence`)
- Action pattern (`ActionPattern`)
- Sense primitive (`Sense`)

These were the classes that were designed. They are simple classes that
contain variables relating to the POSH component properties (see the
2.2.2). It was anticipated that some extra, smaller classes such as the
drive element (`DriveElement`) class would be needed, but they were
implemented in parallel with the file parser, as their composition was
not entirely clear until work had started.

The rest of the system architecture is covered in the next section.

### 4.4 Summary

This chapter has shown that comparatively few design decisions were
made before implementation began — however this in itself was a de-
cision because the iterative development process encourages updating
system requirements regularly (see Sommerville, 2001). The next chapter goes into more detail about the design of POSH IDE.
Chapter 5

Implementation

5.1 Introduction

This chapter will cover the key aspects of the implementation — some early decisions on which language and methodologies were used, some information on the wxWindows classes used, and information on what each of the main classes does.

5.2 Decisions

5.2.1 Development Methodologies

This system was created using the incremental development, iterative process of software development (see Sommerville, 2001, p.51). Sommerville describes this model as the following: ‘where the software specification, design and implementation is broken down into a series of increments which are developed in turn’. Sommerville then goes on to expand on this short description and states that ‘the essence of iterative processes is that the specification is developed in conjunction with the software’ — it is this specifically that made the process work well with this system as the requirements were not concrete at the beginning of the process.

This was complemented by using rapid, throw-away prototyping methods (see Sommerville, 2001, p.46).
5.2.2 Programming Language

One of the main purposes of this system is to present information in an appealing way and to allow for ease of modification, and another is for it to be as accessible as possible and hence to work across platforms. Python was chosen over other cross-platform languages such as Java (using the Swing GUI toolkit) because Python supports the iterative approach of development better than Java — Python is a loosely typed scripting language, which means that no compilation is needed before a program is run (or more accurately, compilation is transparent to the developer and is performed when the program is run), and there are no primitive types (such as an ‘int’ in Java). These features mean that development was able to progress at a higher rate than with strongly-typed languages.

Choosing Python also allowed for a sense of continuity with PyPOSH AI, which was also written in Python.

5.2.3 Windowing Toolkit

Having chosen Python, the next choice to make was which GUI framework/toolkit to use. This choice was relatively easy and has been covered previously (see 2.4.3) — there are two mature, widely used, cross platform frameworks available, namely ‘Tkinter’ and ‘wxPython’. Tkinter (Tk) uses its own set of widgets that are different to every platform’s native set and hence an application written in Tk will not look at home on any system; wxPython by contrast does use native widgets to draw windows. (wxPython uses the GTK2 libraries on Unix-based systems (see Roebling et al.).) I felt that using native GUI elements would make the user feel more at home when starting to use the system, and so wxPython was chosen.

5.2.4 Rapid Application Development (RAD)

As mentioned in the previous chapter, RAD tools were used to create early prototypes quickly. The current iteration of the system contains three main sections in the main frame:

- The left panel, which holds the list of currently available POSH components.
- The middle panel, holding the tree.
Figure 5.1: The POSH IDE main window

- The right panel (referred to as the ‘edit’ panel), which dynamically displays editable information about the currently selected component or element in the tree.

The RAD environments were used as little as possible, as exposure to the code generated was kept to a minimum by the RAD tools which meant that understanding the code structure would be difficult if the system grew too much. In fact, the prototypes generated by Boa/wxGlade did not have the edit panel present — this was added later by hand.

This does not mean that few prototypes were made, however — merely that those that were created did not have any underlying functionality (they were ‘lo-fidelity’ prototypes, see Rudd et al. (1996)).

5.3 GUI Elements Used

This text has already covered the fact that wxWindows was the windowing toolkit used, but a little more information on the elements used by POSH IDE is now required to understand the following paragraphs.

wxWindows’ class used to create a main application window is called ‘wxFrame’. wxFrame can be given a menu bar (along the top, think
‘File’, ‘Edit’ and ‘Help’), a tool bar (just below the menu bar and holds buttons) and status bar (along the bottom, used to display information about the state of the application running). The rest of the window is devoted to other elements such as panels, trees and buttons.

Window classes (classes that can have other elements added to them) all inherit from the wxWindow class. wxFrame and wxPanel both inherit from wxWindow, as do child window and dialogue box classes.

wxPanel is a useful class on which to place other elements — it does not have any GUI widgets itself, but it allows other elements (including other wxPanels) to be added to it and can be placed on any wxWindow. POSH|IDE has a wxPanel on both the left and right hand sides of the main window, but no visible elements (buttons etc.) are placed directly onto a wxPanel. Placing directly (using absolute positioning) onto a wxPanel is possible but not recommended due to the reasons explained in 4.2.1.

Instead, wxSizers are used to allow relative positioning meaning that the application should display correctly on different architectures. Using wxSizers involves creating an instance of, for example, wxBoxSizer (which inherits from wxSizer) and adding the GUI elements to it in order. wxBoxSizer is the simplest form of sizer, it is essentially a row or a column of elements. On instantiation, the wxBoxSizer is passed a constant that tells it whether to align elements horizontally (ie. as a row) or vertically (a column). Once the elements have been added to the sizer, the sizer is normally added to a wxPanel, and the wxPanel is informed that it now has a sizer via the SetSizer function (inherited from wxWindow).

Two more things need to be mentioned — ‘wxTreeCtrl’ and ‘wxListCtrl’. wxTreeCtrl is wxWindows’ tree class and allows nodes to be expanded or collapsed, coloured and have formatted text; wxListCtrl is a basic list class. Both are able to have a ‘data’ property for each node on the tree and item on the list, which is essentially a pointer to some other object. POSH|IDE makes use of this feature, setting pointers to the relevant POSH component or element in the tree and in the lists.

5.3.1 Summary

This section has introduced the most important wxWindows classes used, and the different panels and their uses in the user interface. The next section outlines how the implementation started once the first few prototypes had been created using the RAD tools, and how the system is structured.
5.4 Development

5.4.1 First Steps

The file parser is central to the program — it forms the foundation of the system. It is what is used to load existing .lap plan files, which includes files previously created by POSH|IDE and files created by hand. If the parser did not work well there would be little point in developing the rest of the program, so it was essential that this functioned well before starting on any other programming. I started development on the parser before any other part of the system — even before the first prototypes of the interface.

Kwong’s PyPOSH system naturally contains a working parser, but a comment at the top of the file that contained the parsing code recommended implementing a full parse tree. Attempting to use Python counterparts for Lex (a file tokeniser generator) and Yacc (a token parser generator) took a little over two weeks with no measurable success, so this was very disheartening. Because so little success was achieved and development was behind schedule already, I decided to take Kwong’s parser code and modify it for use with POSH|IDE.

Four of Kwong’s parser functions were used — ‘stripComments’, ‘readList’, ‘readObjects’ and ‘readFile’. ‘readFile’ is the function called by the main window class when a user chooses a file to open. It takes a file-name and path as its argument, calls ‘stripComments’, ‘readList’ and ‘readObjects’ (which in turn calls other functions). readFile does not return anything, instead the functions readObjects calls such as ‘addCompetence’ will create the relevant POSH components and store them in a Python dictionary, or hash map. More information on the Parser class is given in 5.4.2.

The functions that do most of the work when parsing are those that are called by readObjects. readObjects is passed a list as its argument, and each member of the list (assuming the file has valid syntax) is a list representation of a POSH component. readObjects determines the type of the POSH component and passes it to the relevant support function, which will in turn create an instantiation of the component, and add it to the hash map. The hash maps are then available to the main window class via get* (ie. getDriveCollections, getCompetences etc) methods.
5.4.2 Class Arrangement

This section will not be an exhaustive list of all classes created but description of a few key ones. These are ‘PoshFrame’, ‘Parser’ and ‘PanelMaker’.

PoshFrame

PoshFrame is the main class of the system. It extends wxFrame (covered above), it contains event handling methods and the dictionaries used to point to POSH component objects. PoshFrame also handles redrawing the tree with the information about the POSH components and elements and it updates the edit panels with relevant information about what is selected in the tree. In short, it does most of the work, and some of the key methods are explained below.

The right and left areas of the POSH IDE window are wxPanel instances, and contain elements that are resized automatically via the use of sizers. The tree in the middle of the window is an instance of wxTreeCtrl, and is also placed on a wxPanel instance. The window is split into the three main areas via two wxSplitterWindow instances, ‘leftMainSplitter’ and ‘rightMainSplitter’. leftMainSplitter splits the main window into two sections, rightMainSplitter splits the right hand section created by leftMainSplitter; see Figure 5.2.

The method ‘_do_layout’ was originally generated by wxGlade whilst creating early prototypes. It is a method called on instantiation of the class (by __init__, the constructor method), and sets the basic layout. Since starting to modify the code by hand (not using wxGlade) this method has been heavily modified, adding extra functionality as more features were added to POSH IDE.

A method that is frequently used is ‘redrawNode’, which is given a tree node as argument. redrawNode recursively redraws the node given as
an argument and its children. It does this by first determining the type of POSH component or element (or priority) pointed to by the node, determines type of component or element the node represents, and calls another function depending on its type. The functions (referred to as 'sub-functions' here) called by redrawNode are of the form 'redrawXXNode', where 'XX' represents the type of component the node is representing (e.g. 'DC' for a drive collection). The sub-functions set the colour, name and pointer of the node they are passed as an argument, create the child nodes and then call redrawNode on these child node.

Because redrawNode operates recursively it is used to draw the whole tree, once the root node has been created. The wrapper function 'createTree' handles this: first it creates a DC node (if a is DC present) and then it calls redrawNode to create all the child nodes.

redrawNode contains workaround for an ugly problem — competence elements and drive elements are both classes stored within nested lists (and the level of parentheses represent priority). Because redrawNode needs to find the type of the data pointed to by the tree, examining the drive priority and competence priority nodes both reveal the same type (i.e. a Python list data structure). The first way I used to distinguish between them was to examine the type of the first list member, however it quickly became clear that this would not work when the list was empty (when no drive or competence elements existed). I had to use the tree structure present in the GUI to find the parent node and deduce the type from that, which violates the Model View Controller (MVC) programming paradigm. A better way would be to create a class for both the drive and competence priorities, which would mean it would be possible to distinguish between them easily. This is one of many small improvements that could be made to improve the code and system structure, but unfortunately there is not enough time to implement them all. See 7.4 for more information on the possible improvements.

The final method to be mentioned here is ‘updateLists’ — this is used to keep the information held in the left hand side component lists current, and works first by clearing the lists and then retrieving the names of (and pointers to) the current components from their respective dictionaries, and then by adding the names and pointers to the list. It is called whenever component of any type is created or deleted.

It would make sense to only clear and update one list at a time; eg. update only the competence list whenever a competence is added or deleted. However, none of the operations use noticeable CPU time so the change would only result in negligible performance gain.
Parser

As said before, the Parser was the first part of the system to be developed. The parser uses a few methods from PyPOSH but was mostly rewritten.

The function ‘readFile’ is the function called by PoshFrame when a user wants to open a file, but this does not return a value. A useful modification to this function would be to return a boolean value, denoting whether parsing was successful, which would at least give a primitive indication of whether opening a file was completely successful.

When readFile is called (with a filename and path as an argument), the chain of tasks performed by the Parser is as follows:

1. The function ‘stripComments’ is called to remove all lines that begin with a semi-colon, using a regular expression. stripComments returns the contents of the file as a string.

2. The string returned by stripComments is parsed by ‘readList’, which splits the string into a nested list structure by recursively calling itself each time it encounters a closed bracket ‘)’ which is on the same level as the first opening bracket ‘(’ found. readList returns a list of lists and strings.

3. Next is ‘readObjects’. This function iterates through the list returned by readList, and determines the type of component is specified in each list member by examining its first item — ‘DC’ denotes a drive collection (or ‘RDC’ for a realtime DC), ‘C’ for a competence or ‘AP’ for an action pattern. readObjects will then call the list’s helper function, ‘addDriveCollection’, ‘addCompetence’ and ‘addActionPattern’ respectively.

4. The add* functions will create instances of POSH component classes based on the information (a list) passed to them by readObjects, and add them to the relevant dictionary / hash map.

Once readFile has finished, the PoshFrame class will use the getDriveCollections, getCompetences and getActionPatterns functions present in the Parser class to retrieve the dictionaries storing the POSH components.

PanelMaker

This class is a support class for PoshFrame. It contains functions that create the dynamic edit panels — these are called when the application
is loaded and each function returns a wxPanel. The wxPanel that is returned has one or more wxSizers and all necessary lists, buttons and other GUI elements.

The construction of each wxPanel is similar, but not so similar that one general function could generate each one, so some code duplication is apparent in this class.

5.5 Summary

This chapter has described the methodology used to create POSH|IDE, and it has given an overview of the structure of the underlying system and the user interface. The main classes and their main methods have been explained, and the next chapter will explain how testing was performed.
Chapter 6

Testing

6.1 Introduction

To test the system, three main methods were employed:

- Unit and module testing (see Sommerville, 2001, p.62), which was done whilst programming.
- Testing against functional requirements — that is, ensuring the requirements are fulfilled by using the system.
- Usability testing — three users were given a set of tasks and asked to complete them on POSH|IDE and also on PoshEdit for a comparison, and given minimal help.

6.2 Code testing

To test the code, print statements were used during all stages of development (Python print statements are particularly useful as they can print information about many types of objects, not just strings). The dynamic edit panels show information about the current state of the system, so these were also useful testing tools (once developed) and often gave enough detail to determine the cause of problems without having to resort to adding large numbers of print statements or use the Python debugger.
6.3 Requirements Testing

Testing against the requirements was performed by myself by using the system, and essentially walking through the list of requirements and ‘ticking off’ those which could be performed. See B.5 for more details.

Superficial testing was also done to see if POSH|IDE would run in a win32 environment (with Python 2.3 and wxPython installed), and no problems were found; but the amount of time spent using it was low compared to the amount of time in a Unix-based operating system.

6.4 Usability Testing

Usability testing was done to evaluate the interface of POSH|IDE, to see if it is an improvement over Wright’s PoshEdit — which this project aims to replace. Three users of Bryson’s POSH architecture were given a set of tasks that would constitute creating a simple POSH agent (see the appendix for details), and I watched whilst they carried them out. They also performed the same set of tasks using PoshEdit directly afterwards. The following is a description of each user’s thoughts on the interface — the usability testing is extremely important to this type of system, and so is quite in-depth.

It should be said that the system was not changed between tests with each user.

6.4.1 The First Tester: Dr. Bryson

Bryson is the author of the original POSH AI system, so it is likely that she will be one of the first (and main) users of POSH|IDE. As such her input into the usability testing is important. Her input is the following:

‘Add primitives’ button To add primitives, the ‘open’ button used instead of ‘Add primitives’ in the file menu. This could be remedied by adding an ‘Add primitives’ button, next to the ‘Open’ button.

Primitives files not could not be found The primitives files were not in an expected directory. This was not strictly a problem with the interface, more a general problem that was addressed before a final release. Dr. Bryson also suggested changing the file filter used in the ‘Add primitives’ dialogue box to a more restrictive *prims.lisp wildcard — a *.lisp wildcard was used in the test.
**Competence primitives list too small**  This issue is self explanatory. Resolving will require a modification to the relevant editing panel. Bryson suggested having some editing options available in child windows that opened when, for example, a button was clicked in the main panel, in a similar manner to PoshEdit. This idea was contested by Mark Wood, the second tester.

**‘Move left’ and ‘Move right’ labels unintuitive** The labels of the buttons used for modifying triggers, goals and action pattern elements have been changed to ‘Add’ and ‘Remove’ respectively. This was an issue that I could not decide upon when coding the interface; I chose ‘Move ...’ labels arbitrarily.

**Priorities inconsistent with POSH AI** It was not immediately obvious that a priority (for a competence or drive) should be added before a drive element could be added. This is an issue that each user found unintuitive. It does not mirror the POSH systems in that POSH priorities are given by the number of parentheses in the .lap plan file, Bryson suggested using a child window to show the priorities of elements.

**Double clicking** It was expected that double clicking on an item in most of the lists would allow a relevant action to be performed. eg. double clicking on a primitive in a competence trigger list should remove that item from the list.

**Bug — ‘Sub elements’ labelling** An error was found in labelling on the Drive Collection panel — ‘Sub elements’ ought to be changed to ‘Drive elements’ to preserve consistency. This has been fixed.

**Unexpected editing behaviour** It was unexpected that a competence or action pattern could only be edited when present in the tree. However this was a conscious design decision, as competences or action patterns not present in the tree will not be used by the agent anyway — only those present as a POSH root in a drive element are used, and only these are present on the tree. Bryson contested this by stating that humans do not always develop software (of any kind) in the most ‘logical’ way — eg. the user may wish to start development with just a competence, and not have to create a DC and drive element first.

**Bug in the competence editing panel** Each time the competence editing panel is displayed, the list showing the primitives present in the trigger is not cleared, meaning that the list will contain duplicate entries if the panel is updated once or more. This has been fixed.
Automatically show new entry When an item (such as a drive element or priority) is added to the tree, Bryson thought it should be automatically selected and the relevant editing panel shown.

Confirmation dialogues inconsistent It was felt that the confirmation methods when adding or deleting a priority were inconsistent with the rest of the design. A confirmation dialogue would only be necessary if a priority was to be removed.

Sense predicate and value missing It was also noticed that a feature of the plan files could not be edited — the predicate or value of a sense could not be changed. This conflicts with the functional requirements and as such is a bug.

Some general comments made by Bryson were:

- Child windows, instead of panels, would be useful to allow editing as they could be resized and hence show more information
- The tree should either expand all children by default, or remember the state of existing nodes when a node is added or removed. That is, whether each node is collapsed or expanded to show its children — each node’s actual presence in the tree is handled automatically

When comparing to PoshEdit, it was felt that PoshEdit’s method of displaying priorities was more intuitive. Bryson felt that competence element and drive element creation was easier in PoshEdit, but Wood and Korvin felt the opposite (see below).

6.4.2 The Second Tester: Mark Wood

Wood, at time of testing, was a PhD student who was planning to use either the POSH or PyPOSH system for use in his research, and as such understands the BOD theory. It is likely that he will be one of the first users of POSH|IDE.

Adding Primitives Wood did not have the same problem with loading primitives as Bryson (he found the menu item straight away), however he still needed to be guided to the correct directory that contained the primitives file.

Drive and competence priorities This was the single most important criticism made by each tester — that POSH|IDE’s implementation of drive and competence priorities does not mirror that of the POSH systems closely enough. Wood suggested that more intuitive ways of handling priorities would be to treat a drive priority as a property of an element and as such add a spin control box on
the element edit panel, or to emulate the PoshEdit child window opened specifically for altering element priorities ('orders' within PoshEdit). Unfortunately this second method does not allow for parallel priorities, but it allows scope for further investigation.

**Tree should automatically expand** As Bryson commented, it would be helpful if when an item is added, the tree automatically expanded to show it. Alternatively the state of the tree should be remembered so that every time the tree is modified, the expanded/collapsed state of the nodes that are unchanged does not change. At time of testing, the tree would default to show all drive elements, but the drive elements themselves would be collapsed. This has since been changed, the tree now expands to show all nodes by default.

**Panel redraw bug** Wood found a bug: when the drive collection edit panel is shown and either a competence or action pattern is added, the name of the added component is not shown in the drop-down menu used to enter a POSH root.

**Double clicking item in list** It was felt that double clicking on a component (action pattern, competence or drive collection) in the respective 'available' list should select the appropriate node in the tree. This is a similar observation to Bryson's wish to modify triggers or goals by double clicking on primitive list items.

**Add 'order' buttons for action patterns** At time of testing, the only way to order the elements of an action pattern is to remove each primitive, one by one, from the tail-end of the list and then add the primitives in the correct order. It is hence possible to reorder the elements, but there is no efficient method for doing so (e.g. 'Move up'/'Move down' buttons).

Wood had more problems than Bryson using PoshEdit. The same set of tasks were attempted again using PoshEdit, and Wood felt that the interface of POSH|IDE was an improvement in most areas. However, PoshEdit allows two common tasks to be automated by double clicking the relevant item of the GUI, and its handling of element priorities ('ordering') was more intuitive than POSH|IDE.

Wood had fewer problems using POSH|IDE than Bryson, and more problems using PoshEdit. This may be because Bryson has previously used PoshEdit; or there may not be a specific reason.
6.4.3 The Third Tester: Emily Korvin

Korvin was an undergraduate who was modelling primate behaviour in Bryson's BOD system. She had been using the Lisp version and the GUI that Bryson developed, and so was able to make a comparison between the lisp GUI, POSH|IDE and PoshEdit. The issues she found with POSH|IDE were as follows:

Adding primitives As with Bryson and Wood, Korvin had to be guided to the correct location of the primitives files.

Drive and competence priorities Korvin did not find adding a drive element very easy; she had to be told to add a priority first. It is now very clear that this is a major issue with the interface.

Competence not added to tree When a new competence is created, Korvin expected a representative node to appear on the tree, but eventually found that it would appear by setting a POSH root in a drive element as the competence name.

Ordering elements When trying to reorder the elements in a DC or competence it quickly became apparent that there was no direct method to do this, and that the only way was to remove all and then add the elements in the correct order.

Double clicking Korvin expected that double clicking on list items would perform a relevant action, as did Bryson.

Korvin did, however, have more problems using PoshEdit than POSH|IDE and tasks took longer to complete in PoshEdit. It was also apparent that PoshEdit has some stability issues as it crashed twice whilst trying to perform the same set of tasks as in POSH|EDIT, without allowing to save changes.

She also found using POSH|IDE easier to use than Bryson's original GUI (POSH-GUI) as an editor, however POSH-GUI contains a larger featureset — it contains features allowing it to integrate with the POSH system such as stepping so a comparison could not be made here.

6.4.4 Observations

From the three tests it has become clear that there were some usability issues that were common with each of the testers, mainly:

- The location of the primitives file
- The tree hiding information by default
• Drive and competence priorities ‘hiding’ elements
• A bug in the competence edit panel
• The labelling of ‘Move left’ and ‘Move right’ buttons
• Double clicking on lists and expecting a reaction

However, even with these issues, two out of the three users preferred POSH|IDE, and two users also crashed PoshEdit within approximately five minutes of use where POSH|IDE suffered no stability problems.

6.4.5 Improvements Made

The usability testing has been taken into account, and a few changes have been made in an attempt to improve the interface. The ‘Move...’ buttons have been relabelled ‘Add’ and ‘Remove’, the tree displaying the hierarchy of the agent plan now expands to show all information by default and the bug in the competence panel has been fixed.

The major, outstanding issue is that of the priorities. This change would not be trivial, and would unfortunately require more time than available as it would require some major changes to the underlying structure of the system.

6.4.6 Summary

This chapter has explained the testing done to POSH|IDE, and has shown how the first three users felt about its interface. Overall, the results were positive but with some issues that required attention.

Some of the comments were taken into account and some fixes were made, but some remain. The next chapter is a discussion of the process undergone in creating POSH|IDE, achievements, reflections and future work.
Chapter 7

Discussion and Conclusions

7.1 Introduction

The aims of this project were originally very ambitious; they involved creating an editor for the POSH-BOD .lap plan files and integrating that editor with the BOD systems, and then creating a BOD agent for the purposes of testing the newly created IDE. Unfortunately, what has been created is but a subset of these aims, only the first aim has been achieved — a functional editor has been created, but the work toward integration did not succeed (which is of course due to time restrictions).

Lack of time (or, more accurately, misjudgement of time and over-ambitious planning) has in effect cut this project short. However, the end product has proven to be more usable and stable than the system it was intended to replace, so in that sense some success has been achieved. Hopefully it will prove to be more useful, too.

There are two main reasons why the project had not progressed as much as hoped. First and foremost is the delay in getting a working parser — too much time was wasted trying to implement a full parse tree, instead of using Kwong's functions to begin with. The second problem (although this one could not be avoided) was the fact that my knowledge of BOD was poor before starting to code. This meant that more time than planned was spent on learning the BOD theory and on creating the POSH component classes, and pushed back development of the rest of the system.

However, the iterative and rapid / throw-away prototyping techniques helped requirements to emerge and my understanding of BOD to develop in parallel with the software.
7.2 Achievements

This project has created a replacement for PoshEdit. Although the aims of this project included more than an editor, the amount of time necessary to implement all the wanted features was underestimated by such an extent that only the bare minimum requirements were fulfilled.

A success, but not a complete success.

7.3 Reflections

The aims of this project were ambitious from the start — this much was clear. Creating a fully integrated development environment for a complex system I had had no previous experience with or knowledge of, followed by using and testing the IDE by creating an agent to be used in a game were (in hindsight) unrealistic aims (this much is now obvious). As is often the case, the problem was lack of available time — aside from the initial problems in creating a working parse tree, there were no significant problems during the development process that stalled progress so development was able to continue at a reasonable rate.

A significant amount of time was spent on designing and re-evaluating the user interface. The previous chapters have noted that there were some usability issues with the end product — if so much time had not been spent on redesigning the interface the problems here would have been far greater, so although more time could have been spent on adding features, the system would not have been easy to use. This is a trade-off that needs to be considered when creating any system, but one that is particularly important when its primary purpose is to make some other system easier to use. With this in mind, having a usable system with fewer features is a better end product than a difficult to use and frustrating system with more features.

The usability testing helped an enormous amount in this respect. It pointed out several major issues, a few of which have corrected, meaning the end product should be in a better state than when it was tested. Unfortunately not enough time was available to re-test the system, so this cannot be verified.

The original plan for this project was to have the implementation of the IDE completed by April, so that there would be time to write this report, with a view to spending time extending Kwong's work on the UT bot. The editor was in a usable state by April, but of course there was no integration with the PyPOSH engine. Although the implementation
was behind schedule, work commenced on the report in April to attempt to further develop the editor (to add the functionality that had been left out) once the report was complete; however not enough time was left for this to happen after finishing the report.

7.4 Future Work

7.4.1 Introduction

Because some desired functionality was left out, there is plenty of work that can be done on POSH IDE in the future to improve it.

7.4.2 Problems, issues and bugs

All except one of the bugs that were identified by the testing have been fixed; the remaining bug is the lack of a facility to edit the predicate and value of a sense primitive.

The major issue remaining with the interface is the fact that the users felt that POSH IDE did not treat drive and competence priorities the same way that POSH AI did. One possible method of improving this is to remove drive priorities from the tree altogether, and create a field in the element edit panel in which an integer could be entered to represent the priority. This would remove the need to create a priority node before the first element of a DC or competence can be created.

Of course another issue (though not with the interface) is that of the lack of integration with either of the POSH systems, meaning that POSH IDE will not be useful for testing or debugging an agent in its current state. If this was implemented the usefulness of the system would improve significantly.

7.4.3 New features

The new features that could be added in the future are taken from the requirements, namely:

- Logging to a file or to a window
- Linking to a text editor to modify primitives, or to create a primitives editor. Doing this could also fix the bug mentioned above (not being able to modify sense predicate/values)
• Create API for integration with PyPOSCH system, and ensure compatibility with the Lisp version of POSH AI

### 7.4.4 Game Creation or Modification

An early aim of the project was to test the IDE by developing an agent that would run in a gaming environment. Doing this would give the system a trial run and it would alert me to any usability issues that had not been found by the three usability tests. It would also give an indication as to the performance of BOD when in a fast moving, dynamic environment.

### 7.5 Summary

This document has introduced POSH|IDE, an editor for the POSH AI plan files. POSH|IDE was developed with ease of use and completeness in mind, so that it will encourage usage of the promising BOD AI software and methodology created by Bryson.
Bibliography


Appendix A

Code listing

Please see the attached CD-ROM for source code.
Appendix B

Requirements Specification

The following is a formalisation of the requirements that emerged both before and during the development of POSH IDE.

B.1 Non-functional Requirements

The system should be:

- As portable (cross-platform) as possible
- As usable as possible

B.2 Functional Requirements

B.2.1 Necessary

The system must be able to create, modify and delete the following:

1. Drive Collection(s)
   (a) Name
   (b) Goal
   (c) Realtime property
   (d) Drive Elements
      i. Priority
      ii. Name
iii. Trigger
iv. POSH root

2. Action Patterns
   (a) Name
   (b) Timeout
   (c) Elements

3. Competences
   (a) Name
   (b) Time
   (c) Elements

The system should also:
1. Have the ability to load an existing set of behaviours
2. Be able to load existing .lap plan files
3. Create new .lap plan files
4. Be able to save files to the .lap specification
5. Provide a facility to modify the predicate and value attributes of a sense primitive mentioned in the .lap specification

B.2.2 Desirable

The system should:
- Integrate with PyPOSH and POSH AI systems
- Provide logging facilities that contain information about a running agent
- Provide a facility to either:
  - Link with an editor to allow for editing of primitives
  - Provide an in-built facility to edit primitives
B.3 User Requirements

The user must have some knowledge of the BOD theory — the author recommends Bryson (2003b). This paper provides a practical overview to creating a BOD agent, and as such is a good introduction to the theory.

B.4 System Requirements

The system operating system must be any that runs Python and wxPython — primarily Windows, MacOS or Linux.

B.5 Requirements Testing

Each of the functional requirements passed testing, except ‘Provide a facility to modify the predicate and value attributes of a sense primitive mentioned in the .lap specification’. No functionality is currently in place to meet this requirement.