



**University of South Australia**

**Division of Information Technology, Engineering and the Environment  
School of Electrical and Information Engineering  
Knowledge Based Intelligent Engineering Systems Centre**

# **REASONING AND LEARNING FOR INTELLIGENT AGENTS**

**Christos Sioutis**

B.Eng (Hons 1) Computer Systems Engineering, UniSA, Australia

**Thesis submitted in fulfillment of the requirements for the Degree of  
Doctor of Philosophy in Computer Systems Engineering**

**6th January 2006**

*Αυτή η διδακτορική διατριβή αφιερώνεται στους γονείς μου, σας ευχαριστώ για όλα.*  
This thesis is dedicated to my parents, thank you for everything.

What we have to learn to do, we learn by doing...  
(Aristotle)

## Abstract

Intelligent Agents that operate in dynamic, real-time domains are required to embody complex but controlled behaviours, some of which may not be easily implementable. This thesis investigates the difficulties presented with implementing Intelligent Agents for such environments and makes contributions in the fields of Agent Reasoning, Agent Learning and Agent-Oriented Design in order to overcome some of these difficulties.

The thesis explores the need for incorporating learning into agents. This is done through a comprehensive review of complex application domains where current agent development techniques are insufficient to provide a system of acceptable standard. The theoretical foundations of agent reasoning and learning are reviewed and a critique of reasoning techniques illustrates how humans make decisions. Furthermore, a number of learning and adaptation methods are introduced. The concepts behind Intelligent Agents and the reasons why researchers have recently turned to this technology for implementing complex systems are then reviewed. Overviews of different agent-oriented development paradigms are explored, which include relevant development platforms available for each one.

Previous research on modeling how humans make decisions is investigated, in particular three models are described in detail. A new cognitive, hybrid reasoning model is presented that fuses the three models together to offset the demerits of one model by the merits of another. Due to the additional elements available in the new model, it becomes possible to define how learning can be integrated into the reasoning process. In addition, an abstract framework that implements the reasoning and learning model is defined. This framework hides the complexity of learning and allows for designing agents based on the new reasoning model.

Finally, the thesis contributes the design of an application where learning agents are faced with a rich, real-time environment and are required to work as a team to achieve a common goal. Detailed algorithmic descriptions of the agent's behaviours as well as a subset of the source code are included in the thesis. The empirical results obtained validate all contributions within the domain of Unreal Tournament. Ultimately, this dissertation demonstrates that if agent reasoning is implemented using a cognitive reasoning model with defined learning goals, an agent can operate effectively in a complex, real-time, collaborative and adversarial environment.

# Contents

<b>1</b>	<b>Introduction</b>	<b>14</b>
1.1	The Problem of Agent Reasoning and Learning . . . . .	14
1.2	Developing Learning Agents . . . . .	15
1.3	Aims and Objectives . . . . .	15
1.4	Structure of the Thesis . . . . .	16
<b>2</b>	<b>Complex Environments</b>	<b>17</b>
2.1	Introduction . . . . .	17
2.2	Agent-Environment Interface . . . . .	17
2.2.1	Properties of Environments . . . . .	18
2.3	Agent Applications . . . . .	20
2.3.1	Computer Games . . . . .	20
2.3.2	RoboCup Challenge . . . . .	28
2.3.3	Military Applications . . . . .	30
2.3.4	Business Applications . . . . .	33
2.4	Unreal Tournament Environment . . . . .	34
2.4.1	Types of Gameplay . . . . .	36
2.4.2	Multi-Player Mode . . . . .	37
2.4.3	Interfacing with UT . . . . .	38
2.5	Summary . . . . .	40
<b>3</b>	<b>Reasoning and Learning</b>	<b>42</b>
3.1	Introduction . . . . .	42
3.2	Reasoning and Behaviour . . . . .	43
3.2.1	Rational Reasoning . . . . .	43
3.2.2	Human Reasoning . . . . .	47
3.3	Learning and Adaptation . . . . .	55
3.3.1	Machine Learning . . . . .	56
3.3.2	Reinforcement Learning . . . . .	60
3.3.3	Agents and Learning . . . . .	68
3.4	Summary . . . . .	70
<b>4</b>	<b>Intelligent Agents</b>	<b>72</b>
4.1	Introduction . . . . .	72
4.2	Using Intelligent Agents . . . . .	74

4.3	Types of Intelligent Agents . . . . .	76
4.3.1	Deliberate Agents . . . . .	76
4.3.2	Reactive Agents . . . . .	78
4.3.3	Communication Agents . . . . .	80
4.3.4	Hybrid Agents . . . . .	81
4.4	Agent-Oriented Development . . . . .	83
4.4.1	The Prometheus Methodology . . . . .	83
4.5	JACK Intelligent Agents . . . . .	84
4.5.1	The JACK Agent Language . . . . .	85
4.6	Summary . . . . .	89
<b>5</b>	<b>Cognitive Hybrid Reasoning Intelligent Agent System</b>	<b>91</b>
5.1	Introduction . . . . .	91
5.2	Conceptual Model . . . . .	91
5.2.1	Fusing the Reasoning Model . . . . .	91
5.2.2	Integrating Learning . . . . .	95
5.2.3	The Complete Picture . . . . .	97
5.3	Implementation Framework . . . . .	97
5.3.1	The General Architecture . . . . .	98
5.3.2	Extending the JACK Platform . . . . .	98
5.3.3	The Hybrid Reasoning Layer . . . . .	99
5.3.4	The Learning Layer . . . . .	106
5.3.5	Logging How Decisions Are Made . . . . .	108
5.4	Example Agents . . . . .	108
5.4.1	The 10-Armed Bandit . . . . .	109
5.4.2	The Tic-Tac-Toe Player . . . . .	113
5.5	Discussion . . . . .	115
<b>6</b>	<b>Agent-Oriented and Learning Design</b>	<b>118</b>
6.1	Introduction . . . . .	118
6.1.1	Extending the Prometheus Methodology . . . . .	118
6.2	System Specification . . . . .	119
6.2.1	Interface Description . . . . .	119
6.2.2	System Goals . . . . .	123
6.2.3	Functionalities . . . . .	124
6.2.4	Scenarios . . . . .	125
6.2.5	Learning Problems . . . . .	130
6.2.6	Data Stores . . . . .	132
6.3	Architectural Design . . . . .	134
6.3.1	Agent Types . . . . .	134
6.3.2	Behaviour Adaptation . . . . .	137
6.3.3	Agent Interactions . . . . .	141
6.3.4	Overall System Architecture . . . . .	142
6.4	Detailed Design . . . . .	145

6.4.1	Partial UASTroop Learning Agent . . . . .	145
6.4.2	Reasoning Process . . . . .	148
6.5	Performance Results . . . . .	151
6.6	Discussion . . . . .	152
<b>7</b>	<b>Conclusions</b>	<b>154</b>
7.1	Review . . . . .	154
7.2	Contributions . . . . .	156
7.3	Future Directions . . . . .	157
7.4	Concluding Remarks . . . . .	158
<b>A</b>	<b>The Gamebots Protocol</b>	<b>169</b>
A.1	Synchronous Messages . . . . .	169
A.2	Asynchronous Messages . . . . .	170
A.3	Commands . . . . .	171
<b>B</b>	<b>Reasoning and Learning Model</b>	<b>173</b>
<b>C</b>	<b>The 10-Armed Bandit</b>	<b>175</b>
<b>D</b>	<b>Extending the Prometheus Methodology</b>	<b>179</b>
<b>E</b>	<b>Finding a Path in UT</b>	<b>181</b>
E.1	Orientation of UT . . . . .	181
E.2	Actions as Navigation Choices . . . . .	182
E.3	Learning the Quickest Path . . . . .	182
<b>F</b>	<b>Which Weapon to Use?</b>	<b>184</b>

# List of Figures

2.1	Agent-Environment Loop [115]p3 . . . . .	18
2.2	Human-Agent Teaming [111]p727 . . . . .	21
2.3	The Quake Environment[80] . . . . .	23
2.4	The Simulated Soccer Environment [103]p28 . . . . .	29
2.5	Situation Assessment [111]p730 . . . . .	31
2.6	The Unreal Tournament Environment . . . . .	35
2.7	Unreal Tournament Multiplayer Mode . . . . .	37
2.8	The Gamebots Server [3]p2 . . . . .	38
2.9	The TclViz Viewer . . . . .	39
2.10	The Javabots Project [70] . . . . .	40
3.1	Reasoning Dimensions . . . . .	43
3.2	Searching The State Space of Tic-Tac-Toe [68],p43 . . . . .	45
3.3	Production Systems Operation Cycle . . . . .	46
3.4	BDI Reasoning Process . . . . .	48
3.5	Relating Work Environment to Cognitive Resource Profiles of Actors [88],p25 .	51
3.6	Rassmusen’s Decision Ladder [88],p65 . . . . .	52
3.7	Gulfs of Execution and Evaluation, adapted from [88],p127 . . . . .	53
3.8	Boyd’s Observe, Orient, Decide and Act Loop [14] . . . . .	54
3.9	Pumping up The OODA Loop Speed [30] . . . . .	55
3.10	Examples of The Concept “arch” [68]p357 . . . . .	57
3.11	Learning the Concept “ball” [68]p364 . . . . .	58
3.12	The Artificial Neuron [68]p420 . . . . .	59
3.13	General Form of The Genetic Algorithm [68]p471 . . . . .	60
3.14	The RL Agent-Environment Interface [104]p52 . . . . .	61
3.15	An Enhanced Agent-Environment Interface . . . . .	62
3.16	The Cart-Pole Reinforcement Problem [49]p3 . . . . .	63
3.17	The Car-On-Hill Reinforcement Problem [49]p3 . . . . .	63
3.18	First Visit Monte Carlo Algorithm [104]p113 . . . . .	65
3.19	Sarsa RL Algorithm [104]p146 . . . . .	66
3.20	QLearning RL Algorithm [104]p149 . . . . .	67
3.21	Model of a Profit Sharing Agent [8]p508 . . . . .	69
4.1	Agent Technology [101] . . . . .	72
4.2	What is an Agent? [95]p33 . . . . .	73
4.3	Deductive Reasoning Agents [95]p52 . . . . .	77

4.4	Production Agents [95]p49 . . . . .	78
4.5	Reactive Agent [95]p47 . . . . .	79
4.6	Goal Based Agents [95]p50 . . . . .	81
5.1	The Observation Stage . . . . .	92
5.2	The Orientation Stage . . . . .	93
5.3	The Decision Stage . . . . .	94
5.4	The Action Stage . . . . .	94
5.5	Introducing Learning in BDI Agents [110],p82 . . . . .	95
5.6	The Learning Stage . . . . .	96
5.7	A Screen Capture of GoalsGenerator . . . . .	104
5.8	Example Goal Hierarchy Graph Generated by GoalsGenerator . . . . .	104
5.9	10 Armed Bandit RL Problem . . . . .	109
5.10	Results Obtained With Bandit . . . . .	114
5.11	Learning Agent Performance when Playing Tic-Tac-Toe . . . . .	116
6.1	The Architecture of UtJackInterface [100]p746 . . . . .	120
6.2	UnrealAgents System Goals . . . . .	124
6.3	Graphical Summary of Functionalities . . . . .	125
6.4	Functionally Descriptors . . . . .	126
6.5	Functionality Data Coupling for UnrealAgents . . . . .	134
6.6	Agent-Functionality Coupling for UnrealAgents . . . . .	135
6.7	Agent Acquaintance . . . . .	136
6.8	Interaction Diagrams . . . . .	141
6.9	Percept Linking for UnrealAgents . . . . .	143
6.10	System Overview for UnrealAgents . . . . .	144
6.11	Process Specifications for UnrealAgents . . . . .	146
6.12	Partial UASTroop Agent Overview . . . . .	147
6.13	UT Movement Capability . . . . .	147
6.14	Partial UASTroop Reasoning Overview . . . . .	148
6.15	UASTroop Decision Goal Hierarchy . . . . .	149
6.16	Which Action to Take? . . . . .	151
6.17	Finding The Quickest Path Between The Home Flag And Enemy Flag . . . . .	152
B.1	Cognitive Hybrid Reasoning Model . . . . .	173
B.2	Cognitive Hybrid Reasoning and Learning Model . . . . .	174
D.1	The Prometheus Methodology [83],p24 . . . . .	179
D.2	Extension to Incorporate New Reasoning Model . . . . .	180



# List of Tables

3.1	Execution of a Simple Production System [68],p173 . . . . .	47
3.2	BDI Behaviour Completion Terminology . . . . .	48
3.3	Examples Of The <i>LORA</i> Logic [115] . . . . .	49
3.4	Expert Systems . . . . .	57
4.1	Behavior Architecture of a Robot on Mars [116] . . . . .	79
5.1	The CHRISConstants . . . . .	99
5.2	Decision Stage Subgoal Behaviour Types . . . . .	102
5.3	GoalGenerator Graph Codes . . . . .	104
5.4	LearningGoal Parameters . . . . .	107
5.5	Learning Options for Bandit Runs . . . . .	113

# List of Code Listings

4.1	Extending the JACK Agent Class . . . . .	86
4.2	Defining a Beliefset . . . . .	86
4.3	Defining a Plan . . . . .	88
4.4	Defining a Capability . . . . .	88
5.1	Implementing a Learning Agent . . . . .	100
5.2	Defining a State Using a Beliefset . . . . .	101
5.3	Implementing a State Manually . . . . .	101
5.4	Fields of the Goals Beliefset . . . . .	103
5.5	Extending the Action Class . . . . .	105
5.6	Implementing an Action Plan . . . . .	106
5.7	Defining a LearningGoal . . . . .	106
5.8	Logging Levels of CHRIS Framework . . . . .	108
5.9	Extending the CHRIS Framework DebugLog . . . . .	109
6.1	Enhancing the Javabots Parser . . . . .	120
C.1	BanditAgent.agent . . . . .	175
C.2	BanditLearningGoal.java . . . . .	176
C.3	BanditAction.java . . . . .	176
C.4	BanditPlan.plan . . . . .	176
C.5	BanditState.bel . . . . .	177
C.6	BanditStateInstance.java . . . . .	177
C.7	BanditPerception.java . . . . .	177
C.8	RandomSelectionPlan.plan . . . . .	178
C.9	TenChoiceTestBed.java . . . . .	178
C.10	BanditSimulation.java . . . . .	178
E.1	Collective State for UtJackInterface . . . . .	181
E.2	StateInstance for UtJackInterface . . . . .	181
E.3	Action for Choosing a UT Navigation Point . . . . .	182
E.4	Plan for Moving in UT . . . . .	182
E.5	Action for Choosing a UT Navigation Point . . . . .	182
E.6	FindEnemyFlag Learning Goal . . . . .	183
E.7	FindHomeFlag Learning Goal . . . . .	183

# List of Abbreviations

ABLE	Agent Building and Learning Environment
AI	Artificial Intelligence
ATC	Air Traffic Control
BDI	Beliefs, Desires and Intentions Model
$C^2$	Command and Control
CGF	Computer Generated Forces
CHRIS	Cognitive Hybrid Intelligent Agent System
CLARET	Consolidated Algorithm for Relational Evidence Theory
CTF	Capture the Flag
CWA	Cognitive Work Analysis
DARPA	US Defence Advanced Research Projects Agency
DDF	Distributed Data Fusion
DP	Dynamic Programming
DSTO	Australian Defence Science and Technology Organization
DVE	Digital Virtual Environment
FPS	First Person Shooter Games
Gaia	Gaia Agent Development Methodology
GPI	Generalised Policy Iteration
GPL	GNU General Public Licence
JACK	JACK Intelligent Agents Platform
JDL	Joint Directors of Laboratories
KB	Knowledge Base
KES	Knowledge Based Intelligent Engineering Systems Centre
MC	Monte Carlo Reinforcement Learning
OAA	Open Agent Architecture
PDT	Prometheus Design Tool
Prometheus	Prometheus Agent Development Methodology
PRS	Procedural Reasoning System
RL	Reinforcement Learning
RPG	Role Playing Games
RTS	Real Time Strategy
SOAR	State Operator And Result
TD	Temporal Difference Learning

OODA	Observe, Orient, Decide and Act loop
ORTS	Open RTS project
OTB	OneSAF Testbed Baseline
SA	Situation Awareness
UAV	Unmanned Aerial Vehicle
UniSA	University of South Australia
UA	UnrealAgents Design
UT	Unreal Tournament
UtJI	UTJackInterface

# Declaration

I declare that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge it does not contain any material previously published or written by another person except where due reference is made in the text.

Christos Sioutis

January 6, 2006

# Acknowledgments

The author would like to express his sincere appreciation to the following for the way each has assisted in this PhD thesis.

Foremost, I would like to express my gratitude to my academic supervisor and great friend Professor Lakhmi Jain. Professor Jain is the leader of the Knowledge Based Intelligent Engineering Systems Centre (KES) at the University of South Australia (UniSA). Professor Jain's determination and kind words of encouragement gave me the determination and strive to succeed in completing my PhD.

I would also like to extend my gratitude to my research team for helping me invaluablely both by discussing my ideas and supporting me in difficult times, specifically I would like to thank Mr Nikhil Ichalkaranje from UniSA, Dr Pierre Urlings and Mr Jeffrey Tweedale from the Australian Defence Science and Technology Organization, Dr Dennis Jarvis and Ms Jaquie Jarvis from Agent Oriented Software. I am further grateful to many colleagues at UniSA for their support and hours of interesting discussion, especially Patrick O'Sullivan, Bill Daniels, Quoc Do and Nimrod Lilith.

Last, but not least, I would like to thank my parents Leonidas and Athanasia Sioutis as well as my brother George Sioutis for all their support, encouragement and sacrifices. Without them I would not be where I am today, for this I thank you. My fiancée, Irene Sofianos for her love, support and countless hours of discussing my ideas and editing this thesis. I would also like to thank the Sofianos family for their support throughout my PhD candidature.

# Publications by Author

The thesis includes material from the following book chapters, conference and journal papers, published by the author:

Sioutis C and Ichalkaranje N (2005), *Learning to Behave: Learning and Intelligent Agents*, IEEE Transactions on Systems Man and Cybernetics - Part C, Submitted for possible publication.

Sioutis C and Ichalkaranje N (2005), *Cognitive Hybrid Reasoning Intelligent Agent System*, Proceedings of the 9th International Conference on Knowledge Based Intelligent Information and Engineering Systems (KES 2005), Springer Verlag, Berlin, pp. 838-843.

Urlings P, Sioutis C, Tweedale J, Ichalkaranje N and Jain LC (2004), *A Future Framework for Interfacing BDI Agents in a Real-Time Teaming Environment*, Journal of Network and Computer Applications, in press.

Sioutis C, Tweedale J, Urlings P, Ichalkaranje N and Jain LC (2004), *Teaming Humans and Agents in a Simulated World*, Proceedings of the 8th International Conference on Knowledge Based Intelligent Information and Engineering Systems (KES 2004), Springer Verlag, Berlin, pp. 80-86.

Sioutis C, Ichalkaranje N, Jain L, Urlings P and Tweedale J (2004), *A Conceptual Reasoning and Learning Model for Intelligent Agents*, proceedings of the 2nd International Conference on Artificial Intelligence in Science and Technology (AISAT 2004), University of Tasmania, Hobart, pp. 301-307.

Sioutis C, Urlings P, Tweedale J and Ichalkaranje N (2004), *Teaming Humans and Intelligent Agents*, in Applied Intelligent Systems, J Fulcher and LC Jain (eds), Springer Verlag, Berlin, pp. 255-279.

Sioutis C, Ichalkaranje N and Jain LC (2003), *A framework for interfacing BDI Agents to a real-time simulated environment*, Proceedings of the 3rd International Conference of Hybrid Intelligent Systems (HIS 2003), IOS-Press, Amsterdam, pp. 743-748.

Urlings P, Tweedale J, Sioutis C and Ichalkaranje N (2003), *Intelligent Agents and Situation Awareness*, Proceedings of the 7th International Conference on Knowledge Based Intelligent Information and Engineering Systems (KES 2003), Part II, Springer Verlag, Berlin, pp. 723-733.

Urlings P, Tweedale J, Sioutis C and Ichalkaranje N (2003), *Intelligent Agents as Cognitive Team Members*, Proceedings of the 10th International Conference on Human Computer Interaction (HCI 2003), Lawrence Erlbaum Associates, Publishers, USA, pp. 355-359.

# Biography of Author



Christos Sioutis was born on the 27th of July 1978 in the farms of Barmera, South Australia. Mr Sioutis spent his early years of schooling in Kalamata, Greece and his latter years in Adelaide, South Australia where he graduated with a Bachelors Degree in Computer Systems Engineering (Honors 1) in 2001 from the University of South Australia.

Mr Sioutis is currently a PhD candidate within the Knowledge Based Intelligent Engineering Systems Centre (KES) of the School of Electrical and Information Engineering, University of South Australia. His personal research interests are focused in the area of artificial intelligence, specifically when applied to intelligent agents and their application domains such as software simulations, robotics and unmanned aerial vehicles. Mr Sioutis is especially interested on the design and implementation of internal components in agents to more closely match the reasoning performed by humans.



# Chapter 1

## Introduction

Agent technology has proved to be a promising platform for future development in two complementary aspects. The first aspect involves helping out humans manage with the vast amount of information available to them. The second aspect involves putting the now extremely powerful computer systems into more useful and efficient use. Even though computer processing power has grown over the last few years, the functionality provided by software to users is based on information accessibility and manipulation. The main tasks of managing and controlling the information have remained a problem of the user, consequently, as the sheer amount and complexity grows, the tasks performed by humans are becoming increasingly difficult.

Simply put, agent oriented development is a move to incorporate into computer systems the ability to do some of the tasks that the user is normally required to do. Agents are expected to operate intelligently and reliably irrespective of the situations they are presented with. An extension of agent oriented development is the concept of combining several agents together into a multi-agent system. In this case, all the agents run concurrently and independently, with each given the responsibility of only a small and specific subtask of the system. The power of multi-agent systems emerges from the agents communicating with each other and working together to accomplish common goals.

### 1.1 The Problem of Agent Reasoning and Learning

Application domains that are good candidates for agent oriented development require complex and timely assessment of unfolding situations and then possibly interaction with multiple systems that may result in changes affecting the physical world. The complexity of situation assessment arises from the dynamic nature of information observed (which may include a certain degree of measurement error) and the agent's limited interface. For various systems discussed in chapter 2 it becomes very hard to define what an agent based system should do in all possible situations that may be encountered. The ability to cope with unknown situations is a key requirement in achieving a fundamental agent property, autonomy. This is especially true for agent systems required to operate for extended periods of time with no or limited human guidance. Additionally, some systems must be able to exhibit changes in behaviour during operation in order to compensate for drifts in dynamic factors of the environment. This can include tweaking the behaviour of agents in real-time in order to maximize performance

for a particular environment. The only way that this can be achieved is by agents being able to learn from their experiences.

Reasoning is the thinking process that occurs within an agent that needs to make a particular decision. This may involve performing rational reasoning where decisions made through a direct reflection of knowledge or echoing the way that humans perform reasoning. Reasoning begins by observing the environment, then considering goals to achieve and ends with taking appropriate actions. The significance of directly considering the reasoning process when thinking about how to integrate learning in agents, lies in the generality of the solution obtained.

## 1.2 Developing Learning Agents

Initial research on achieving autonomy involves creating agents with the ability to *learn* how to behave in unknown situations. The techniques and algorithms developed within the machine learning area of artificial intelligence research have provided a rich arsenal of options for mixing and matching to different applications. The design of learning agents has traditionally occurred by firstly designing the agent (or multi-agent system) itself and then integrating a learning technique to satisfy a particular requirement. Mature agent development frameworks have therefore evolved by taking two divergent paths. Firstly, concentrating only on agent-oriented development with no learning support. Learning algorithms must be implemented and integrated from scratch to operate in conjunction with the development framework and the required application. Secondly, providing a range of generic components (which may include learning algorithms) that can be combined to create the agent. This is still however not sufficient, as developers are required to obtain a considerable understanding of the theory behind machine learning in order to know how to apply the algorithms correctly and efficiently.

This thesis defines agent learning as the union of three properties. The first being dynamic development of new behaviours through the combination of simpler tasks. Secondly, an improvement in the agent's performance such that it performs faster and/or more efficiently. Thirdly, being able to adapt to changes in the environment. Therefore, if an agent is placed in an unfamiliar situation it will still make (and improve upon) decisions towards achieving its desires. When the agent subsequently encounters a familiar environment it will use its previous knowledge for further decisions that it needs to make. Learning also provides the additional feature of making agents act more human-like, as they are capable to adapt to changing circumstances including improving their tactics when placed in opposition to a human.

## 1.3 Aims and Objectives

Previous research conducted on how to successfully integrate learning into agents has followed the approach of focusing on a specific application and applying suitable learning techniques to solve their problem. This however, suffers from the fact that it becomes hard for subsequent researchers to understand, transform and apply the contributions presented, to their own problem domain. The aim of this thesis is to firstly illustrate how the concept of learning can fit into a model of agent reasoning, and secondly, define how such a reasoning and learning model can be implemented as a generic extension to a mature agent development platform.

The enhanced agent development platform can in turn, be used to solve problems on many different application domains.

The thesis is segmented into two main parts as per the objectives of the chapters. The first part of the thesis builds an understanding of the underlying topics on which the contributions are made. It justifies the need to use learning by describing the complexities that agents face in different environments. The thesis then builds an understanding of previous research conducted on the topics of reasoning and learning. Additionally, it presents a general overview of the current state of intelligent agent technology.

Once the background research has been understood, the thesis contributes a new reasoning and learning model and illustrates how it can be developed as an extension to an agent development platform. The final objective of the thesis is to emphasize the need of identifying and working on learning problems early in the agent-oriented design process. This is in contrast to considering learning after the entire agent system has been designed and/or implemented.

## 1.4 Structure of the Thesis

Chapter 2 describes some of the possible features that may be present in environments, where an environment includes all parts of a system that are outside the agent. It then reveals the need for learning such that agents are able to cope with, and operate autonomously within such environments. Chapter 3 describes previous research that has been performed in order to understand and model how humans make decisions. Some of the models presented have been used as a basis for creating agent development frameworks. Chapter 4 provides a thorough description of agent technology. It begins with a quick historical background, then describes different types of agents that are being researched and ends with a review of some of the more popular and mature agent development frameworks. Chapter 5 describes a new, hybrid model for agent decision making that is based on human reasoning. This new model provides additional detail to indicate how learning can be effectively integrated into the agent's reasoning process. It then presents a framework that implements a subset of the functionality presented by the model. Chapter 6 provides the detailed design and construction of a learning agent for a dynamic environment and analyzes its performance. The thesis ends with some concluding remarks and future work in chapter 7.

# Chapter 2

## Complex Environments

This chapter gives a comprehensive outlook on the need to incorporate learning in certain environments due to their complexity. It describes the way that agents interface with their environment, it then analyzes a number of agent applications where an agent would be faced with such a complex environment. Finally, it provides a detailed overview of the environment which is used as a testbed for this thesis.

### 2.1 Introduction

Agent technology has been applied to many different application areas, each focusing on a specific aspect of agents that is applicable to the domain at hand. The role that agents play in their environment distinctly depends on the application domain. The agent research community is very active. Environments are mostly viewed as testbeds for developing new features in agents and showing how they are successfully used to solve a particular problem. Fortunately, in most cases this is a two-sided process, by understanding, developing and improving new agent technologies it becomes possible to solve similar real life problems. Consequently, as the underlying foundation of agent software matures, new publications describe how agents are being applied successfully in increasingly complex application domains.

Section 2.2 describes the issues involved when interfacing agents with their environments, this includes the properties of environments that must be considered when designing the agent. Section 2.3 describes different types of agent applications that yield complex environments which cannot be solved with traditional agent development techniques. Finally, section 2.4 focuses upon the complex environment chosen for this research where agents are situated in a rich, simulated world and are tasked with working as a team to achieve a common goal.

### 2.2 Agent-Environment Interface

This section describes the interface between an agent and its environment. The agent is understood to be a decision-maker and anything that it interacts with, comprising everything outside the agent itself, is referred to as the environment. The environment has a number of features and generates *sensations* that contain some information about the features. A *situation* is understood as a complete snapshot of the environment for a particular instance

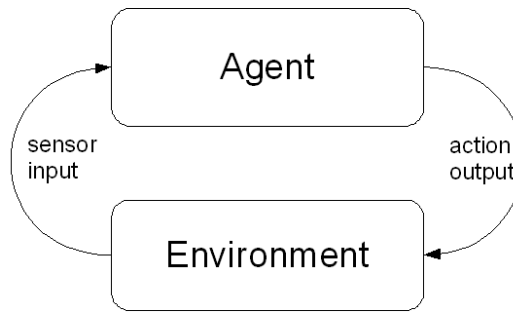


Figure 2.1: Agent-Environment Loop [115]p3

in time. In a number of literature however, the term *state* is used with the same meaning. In this thesis a clear distinction is made between the two terms, a situation is defined as a *complete* snapshot of the *real* environment. Hence, if an agent is able to obtain or deduce the situation of its environment it would know everything about the environment at that time. A state is defined as a snapshot of the *agent's beliefs* corresponding to its limited understanding of the environment. This means that the state may or may not be a complete or accurate representation of the situation. This distinction supports research being conducted on improving the agent's Situation Awareness (SA), whereby SA measures how similar the state is as opposed to the situation. Section 2.3.3 provides more details on the topic.

The agent and the environment interact continually, the agent selects actions and the environment responds to the actions by presenting new sensations to the agent [104]. The interaction is normally segmented in a sequence of discrete time steps, whereby, at a particular time step the agent receives data from the environment and on that basis selects an action. In the next time step, the agent finds itself in a new state, this operation loop is illustrated in figure 2.1.

## 2.2.1 Properties of Environments

Russel and Norvig [95] have classified the different properties of environments into five categories, they are described in this section in detail.

### 2.2.1.1 Fully Observable or Partially Observable

A fully observable environment provides the agent with complete, accurate and up-to-date information, of the entire situation. However, as the complexity of environments increases they become less and less observable. The physical world is considered a partially observable environment because it is not possible to know everything that happens in it [114].

On the other hand, depending on the application, the environment should not be expected to be completely observable. For example, if an agent is playing a card game it should not be expected to know the cards of every other player. Hence, in this case, even though there is hidden information in the environment and this information would be useful if the agent knew it, is not necessary for making rational decisions [104].

An extension of this property is when sensations received from the environment are able to summarize past sensations in a compact way such that all relevant information from the

situation can be deduced. This requires that the agent maintains a history of all past sensations. When sensations succeed in retaining all relevant information, they are said to have the *Markov* property. An example of a Markov sensation for a game of checkers is the current configuration of the pieces on the board, this is because it summarizes the complete sequence of sensations that led to it. Even though much of the information about the sequence is lost, all important information about the future of the game is retained [104].

A difficulty encountered when dealing with partially observable environments is when the agent is fooled to perceiving two or more different situations as the same state, this problem is known as *perceptual aliasing* [7]. If the same action is required for the different situations then aliasing is a desirable effect [7], and can be considered a core part of the agent's design, this technique is commonly called *state generalization* [104]. However, if each of the situations requires a different action then the agent becomes *confused* and takes the wrong action. This is described in detail in chapter 3.3.

### 2.2.1.2 Deterministic or Stochastic

Determinism is the property when actions in the environment have a single guaranteed effect. In other words, if the same action is performed from the same situation, the result is always the same. A useful consequence of a deterministic environment is the ability to predict what will happen before an action is taken, giving rise to the possibility of evaluating multiple actions depending on their predicted effects. The physical world is classified as a stochastic environment as stated by Wooldridge [116]. However, if an environment is partially observable it may appear to be stochastic because not all changes are observed and understood [95], if more detailed observations are made, including additional information, the environment becomes increasingly deterministic.

### 2.2.1.3 Episodic or Sequential

Within an episodic environment, the situations generated are dependent on a number of distinct episodes, and there is no direct association between situations of different episodes. Episodic environments are simpler for agent development because the reasoning of the agent is based only on the current episode, there is no reason to consider future episodes [114]. An important assumption made when designing agents for episodic environments, is that all episodes eventually terminate no matter what actions are selected [104]. This is particularly true when using learning techniques that only operate on the completion of an episode through using a captured history of situations that occurred within the episode. Actions made in sequential environments, on the other hand, affect all future decisions. Chess is an example of a sequential environment because short-term actions have long-term consequences.

### 2.2.1.4 Static or Dynamic

A static environment is one that remains unchanged unless the agent explicitly causes changes through actions taken. A dynamic environment is one that contains other entities that cause changes in ways beyond the agents control. The physical world continuously changes with external means and is therefore considered a highly dynamic environment [114]. An example

of a static environment, is an agent finding its way through a 2D maze. In this case all changes are caused by the same agent. An advantage of static environments is that the agent does not need to continuously observe the environment while its deciding the next action. It can take as much time as it needs to make a decision and the environment will be the same as when previously observed [95].

#### 2.2.1.5 Discrete or Continuous

An environment is discrete if there is a fixed, finite number of actions and situations in it [114]. Simulations and computer games are examples of discrete environments because they involve capturing actions performed by entities, processing the changes caused by the actions and providing an updated situation. Sometimes however, this process is so quick that the simulation appears to be running continuously. An example of a continuous environment is taxi driving, because the speed and location of the taxi and other cars changes smoothly over time [95].

#### 2.2.1.6 Single-Agent or Multi-Agent

Although the distinction between single and multi-agent environments may seem trivial, recent research has surfaced some interesting issues. These arise from the question of what in the environment may be viewed as another agent [95]. For example, does a taxi driver agent need to treat another car as an agent? [95] What about a traffic light or a road sign? An extension to this question is when humans are included as part of the design of the system, giving rise to the new research area called human-agent teaming.

Urlings [111] describes using human-agent teaming for gathering Situation Awareness in military scenarios by an order of magnitude greater than that of a human alone. A re-organization of human-machine interaction is proposed by imposing a paradigm shift as shown in figure 2.2, this realizes an improved teaming environment in which humans, assistant agents and cooperative agents are complementary team members. In formulating such teams, the main change required from a simulation point of view is that humans and machines are *not interchangeable*, they are *complementary* [111]. One approach to achieve this is through the use of *liaison agents* for representing the human to other agents and vice-versa. This is achieved by maintaining beliefs about the human in regards to activities performed and the required roles in the team [71].

The remainder of this chapter explores a number of interesting applications where agents have been used. The complexity of applications varies with the aims of the research.

## 2.3 Agent Applications

### 2.3.1 Computer Games

The role of agents in computer games is based on making the AI of characters in the games more human-like such that they harder to beat. This involves the characters being able to adapt

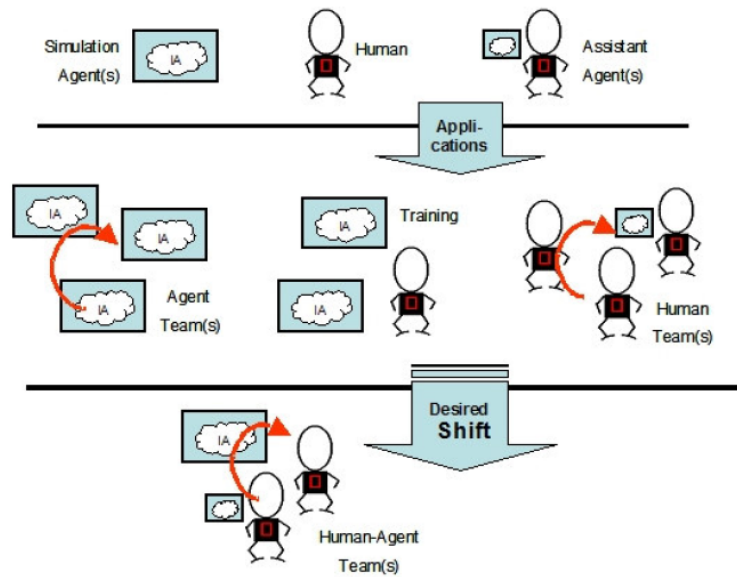


Figure 2.2: Human-Agent Teaming [111]p727

to repetitive tactics used by a human player, predicting possible outcomes, taking appropriate actions and forming teams in order to win.

There has been considerable research in the development of AI in computer games and there are many web sites on the internet devoted on this topic. This is due to the fact that most game developers publish documentation on how to interface with their game software and create modifications to the game. When a game is reviewed after release, great emphasis is placed on the AI, this is because the better the AI is, the game becomes more enjoyable and interesting to play. In most cases, the level of AI in characters is customizable and is usually set according to how hard the human wants the game to be. In the hardest setting, the simulated characters can be extremely hard if not impossible to beat.

Computer video games provide a perfect platform for agent environments, this is because they are developed by commercial companies for making profit and therefore are required to be feature rich, stable and fun to play with. They have also been stress tested by many people that have played the games over years. Game producers regularly release updates in order to fix problems encountered by people or introduce new features. A number of games also have documentation on how to make modifications to the game, third party modifications created by enthusiasts are commonly called *mods*.

Computer games may also be used as a basis for conducting research for military simulation. This is because, while defence agencies may have any number of simulations that could be used for agent-based research, they may not be easily accessible to different personnel working on similar research due to security reasons or required specialized hardware. Therefore, defence agencies have used computer games for conducting unclassified common research while using the developed techniques and results for internal purposes [80]. Different types of computer games are described in the following subsections.



### 2.3.1.1 First Person Shooters

This section focuses on a very popular type of game called First Person Shooters (FPS). The human player is emerged into a 3D simulated world and can see and interact with other entities that are governed by a common, customizable physics engine. It is even possible to modify parameters within the physics engine in order to create subtle differences in the environment. Some FPS games are based on fictional scenarios and characters while others are based on real events such as World War 2. Agents may be used in FPS games to simulate players in the game. In most FPS games the player is required to collect weapons and kill enemies while protecting foes. There are two avenues of research for researching agents with FPS games.

The first avenue involves using agents to control characters in the games such that they appear to be more intelligent and harder to beat. This involves improving their skills, such that they operate efficiently while at the same time making them less predictable. This is important because normally a human player initially finds the game challenging. However, once the human learns how to beat the game, it subsequently becomes easy to predict what the computer will do. The player can therefore easily defend against it while also doing maximum damage.

The second avenue is only applicable when the game supports human-agent teaming. In this case, it becomes important for the human to be able to communicate effectively with agent-controlled team mates while at the same time being able to trust the agents to perform as a human would. Currently, it is mainly up to the human to satisfy the objectives while the rest of the team provides support by simply attacking enemy players. Agents allow for this to be changed, such that the human is able to provide support to an agent and trust it to complete the objective effectively.

The complexity of the environment that agents are faced with in FPS games is due the great amount of information that they need to consider, and the very short time that they have to make decisions. An agent in an FPS game is presented with a continuous flow of data corresponding to the agent's vision, health status, etc. Accordingly, information is also generated asynchronously for important events, like when the agent hears something or is injured by another player. In order to make the game interesting for humans players, the game is executed in real time.

Unreal Tournament (UT), is an FPS game that was chosen for use as the testbed for the work in this research and is therefore described in detail in the last section of this chapter.

The Quake game series was developed by ID Software. Quake and Quake 2 are primarily *single player* mission-type games where the player is given a mission and has to travel through several levels shooting everything in sight. Quake 3 however takes a completely different approach, instead the game has enclosed arenas, where all players have to fight against each other. Players can be controlled by the game itself, called *bots* (short for robots) or by other humans. Quake 3 implements a *DeathMatch* type game where the aim is to simply score the most amount of kills. In order to perform well, players need to seek out and collect powerful weapons. When a player is killed (called *fragged*), they simply reappear (called *re-spawn*) at randomly selected areas in the arena. When a player re-spawns, it has full health again but has lost all its weapons previously collected. Quake 3 has been used for agent research by having

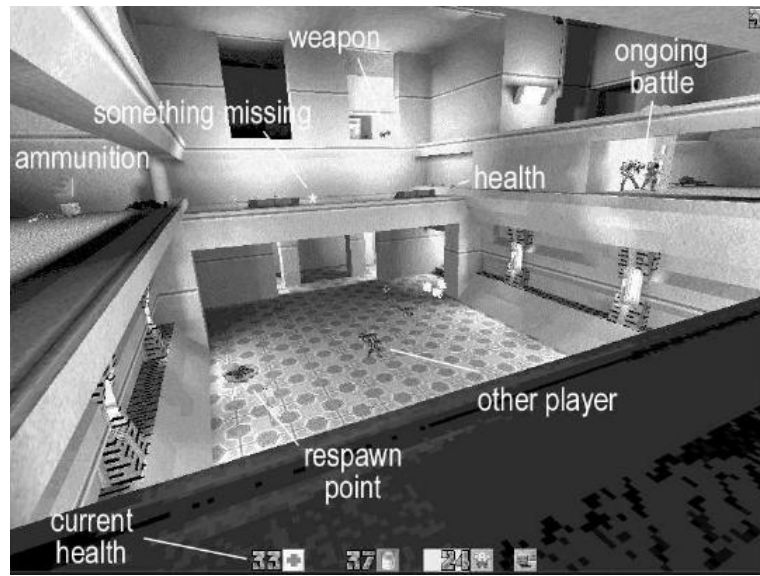


Figure 2.3: The Quake Environment[80]

agents control characters in the game. They are tasked with playing against other agents, game controlled bots and human players.

A major reason that Quake has been used is because ID Software have released the source code under the GNU General Public License (GPL) [39]. The GPL license forces the source code to be freely available and only be redistributed under the same license. Even though people are allowed to sell a product created using GPL software, they must also provide their entire source code including their modifications under the GPL.

Recent research at the University of Rochester have developed an extension for Quake 2 for research and teaching purposes called *Quagents* [17] that allows Quake to be used with the JESS [60] agent development platform. Quagents modifies the game to contain a new set of items that are collected by the agent, for example tofu increases health, gold increases wealth, battery increases energy and data increases wisdom. Each of the different parameters have a different effect on the ability of agents to operate in the environment.

### 2.3.1.2 Real Time Strategy

Real Time Strategy (RTS) games involve the management of a range of assets, that may include virtual people and/or machines and buildings. This requires the delicate sharing of resources to concurrently create new assets like bases/cities, protect assets from enemies while even conquer the assets of other players. The original RTS game that started this genre was called *Civilization* by Sid Meyer that had an addictive blend of building, exploring, discovery and conquest. In civilization, players were tasked to build the ultimate civilization to stand the test of time while competing with other civilizations for resources [74]. Using this idea, other companies built similar games with different themes. Microsoft created *Age of Empires* that involves guiding a small stone-age tribe with minimal resources, into a nation. The game is won via directly conquering enemy nations, or developing an economic victory through the accumulation of wealth [76]. Other popular incarnations include the use of science fiction themes such as *Star Trek: Armada* [2] and *StarCraft* [13] where the player leads the human

race against aliens.

RTS games can be viewed as simplified military simulations where several players struggle over resources scattered over a simulated terrain while setting up economies, building armies, and guiding them into battle. The most successful RTS games are the ones that support multi-player mode, where multiple humans can join the same game and command opposing forces. The complexity of the environment generated by RTS games arises from the following factors:

- They feature many entities, it is not uncommon for each player to have tens or hundreds of entities running concurrently in one game.
- They present incomplete information about the world. RTS games provide information only for areas that are in the ‘field of view’ of entities owned by the player.
- They provide an asynchronous, parallel execution world with multiple possible actions for each step.

Due to the popularity of the multi-player mode for RTS games, developers have placed little emphasis on the AI controlling the entities. This is mainly due three reasons. Firstly, market dictated resource limitations have caused developers to rely on streamlining the multi-player mode and creating detailed graphics as the driving force in sales. Secondly, a lack of competition in developing the AI, caused by the investment needed for implementing the AI. Finally, developers have so far been reluctant to release documentation on how to interface with RTS games and therefore there have been no third-party AI developments in this area [18].

There have been some exceptions where the AI has received greater attention by developers, in-fact in some cases it has proved the driving force behind the popularity of the game and hence sales. *Black And White* is an RTS game that was created by LionHead Studios with an innovative AI engine. The role of a player is to watch over a tribe of people as their village and population grows. The player however, only has a very limited influence on the people and other objects in the world. Therefore, some additional help is needed in accomplishing desired tasks. The player is given a *creature* for this reason. The interesting AI of the game is encapsulated within the creatures available in the game.

The creature is initially given to the player as a baby and its behaviour is random with emphasis on exploring different actions, learning new things and above all, satisfying desires such as hunger or curiosity. The creature for example learns that eating fruit will better satisfy its desire than eating rocks by trying out both and learning the effects. The most popular feature is the player’s ability to train the creature such that its behaviour reflects the wishes of the player. For example, training the creature to attack enemy villagers and protect friendly villagers is done by letting the creature attack both and rewarding it for attacking an enemy while punishing it for attacking a friendly villager. The creature also learns by looking and mimicking what it sees. For example, after watching a villager do a particular task (e.g. carry some wood) the creature tries to do the same thing and the player rewards it accordingly [66].

The developer wanted the creature to appear as a person, this was done by making the creature appear psychologically plausible, malleable and lovable. In order to make the creature

look psychologically plausible the authors used BDI reasoning (see section 3.2.2.1) to dictate how the creature behaves. This way it became possible to explain *why* a creature is behaving in particular way. For example, the creature may be angry because it is hurt, or if a creature wants something there must be an explanation on why it wants it.

The creature was made to look malleable by making it learn different things, such as facts about the environment, how to do things, how to satisfy desires, how to behave towards other entities and what are the best courses of action in different situations. Learning is initiated via feedback, like commands from the player, observation, or reflecting on the effects from its experiences.

The creature was made to look lovable by making the player feel some sort of emotional attachment to the creature via a display of empathy. The developers argue that it is not possible for a human to feel attached to inanimate objects because they do not reciprocate the feelings. Hence, the creature's mind was designed to include a simplified model of the player's mind, it is updated by the creature watching what the player is doing and trying to explain the goals of the player. The creature has the desire to relate to its master, the desire to help its master, the desire to play with its master and the desire for attention.

Finally, a conflict was identified between the creature's usefulness and its person-like requirement. This is because being person-like implies being autonomous while the useful requirement precludes it. The solution of this problem was to make the creature initially completely autonomous, but over time and training, the player can teach it to behave exactly as wanted. This however has both an advantage and a disadvantage. The advantage is that the player receives a feeling of satisfaction as the creature is trained to be useful. The disadvantage is that the creature becomes more focused and robot-like, inevitably losing some of its charm [36].

As mentioned previously, developers have been reluctant to release documentation on how to interface with their RTS games. Therefore, there is no way for third-party researchers to investigate the use of custom software like agents to control entities within RTS games. Also, multi-player RTS games have traditionally relied on every client to run an instance of the entire game and simply hide any information that the player should not be able to see. This approach is prone to people creating software cracks that allow players to cheat, hence spoiling the game experience.

To overcome both of these problems, the Open RTS (ORTS) project was created with an aim to create an RTS platform for AI research. The long term goal of the ORTS project is the creation of AI systems whose performance surpasses human experts in realtime Command and Control ( $C^2$ ) domains. The project uses a client-server approach where a central RTS server runs the entire simulation and allows researchers to connect their agents using openly documented interfaces. Interesting problems encountered when developing agents for ORTS include: adversarial real-time planning, decision making under uncertainty, opponent modeling/learning, spatial and temporal reasoning, resource management, collaboration and pathfinding [19]. ORTS also has the following advantages over commercial RTS games [19]:

- Released under the GPL which allows anyone to download the source code at no cost and contribute to the project.

- Provides an infrastructure for RTS games including a server and a graphics client. The game itself can be modified to generate relevant scenarios for the research at hand.
- Uses a server-client architecture where the server maintains the entire simulation and sends only applicable information to each player.
- Employs an open client communication protocol that allows AI researchers to connect whatever client software they like.
- Allows for AI to be distributed as each client can be running on a different machine if desired.
- Created with  $C^2$  research in mind.

### 2.3.1.3 Role Playing Games

Role Playing Games (RPG) are a form of interactive storytelling. Players take on the role of a character within an imaginary world and advance through a storyline and subplots that are affected by the choices made by the player. The story unfolds via inbuilt *scripts* that are executed in order to guide the player through the game. This involves the designer telling a story via writing small fragments of computer code distributed among characters, various entities, and locations [73].

The main challenge of RPG games is creating believable motivations and behaviours for the different characters that the player may have interaction with. Increasing demands for sophisticated AI and story lines in RPG games have resulted in developers bundling tools to augment the game with third party AI software. The role of AI in RPG games involves creating characters that are able to learn, remember events in the game, and have rich behaviours influenced by their environments. The complexity of environments in RPG games is caused by the following reasons [73]:

- There are many entities in the game, each with a specific purpose in the storyline. Hence, it becomes difficult to organize and track them all during development.
- The simulated world is consisted of many entities with simplistic behaviours. The behaviour of each entity must be explicitly defined.
- Testing is difficult as RPG games are highly interactive. Consequently, players may initiate a series of events that the designer may not have intended, causing conflicts in the storyline.

### 2.3.1.4 Platform Games

Platform games ranging from adventure games to puzzle games. This section describes some of research conducted in three popular platform games.

Chess has received a lot of attention by the AI community, spawning a great deal of successful research. The development of AI for playing chess received much publicity when a computer called *Deep Blue* designed by IBM was able to win the human world championship

in 1997 [53]. Chess is an example of a completely observable environment because both players can see the entire situation at all times, simply by looking at the pieces the board. Deep Blue was able win using brute force, deep search computations. Every move considered involved the calculation of many possible counter-moves and the assessment of relative strengths and features that are likely to lead to victory [64]. Agent-based systems have also been developed for playing chess in order to investigate whether or not agent-specific tactical behaviours can replace a global strategy. The advantage of agent based systems is the distributed approach, each chess piece is controlled by an agent with its own behaviour and perception. Each agent knows the directions in which it can move and evaluates its options accordingly [33]. The complexity of a chess game relies on the assessment of the all the possible moves available, and the fact that agents need to collaborate in order to follow strategies that are best for the entire team.

Checkers research begun with the intention to implement a program that will learn to play a better game of checkers than the person who wrote the program [96]. A program called *Chinook* was created that worked similarly to Deep Blue in that it used deep search to process roughly 1000 possible moves per second, and had databases about all possible positions with 6 pieces or less on the board. Chinook was able to enter the world checkers championships and play against the human world champion [98]. Recent checkers efforts have used a combination of neural and evolutionary computation to allow a program to learn how to play checkers [21]. The environment presented to an agent is somewhat similar to chess, since the both have the same board, and both are completely observable. However there are some subtle differences as checkers has only two types of pawns, and the rules of game play are also different. The complexity is again related to the assessment of the all the possible moves available to an agent and the need to collaborate in order to win.

*Arimaa* was created for a specific purpose, to prove that computers are not even close to matching the intelligence used by humans. Arimaa is played using the same board and pieces provided in a standard chess set but the rules of the game have been changed such that they are very easy for humans to understand but intentionally difficult for computers to play [106]. In-fact the inventor has made a challenge, posting a reward of US\$10,000 to the first computer program that defeats a chosen human representative in an official Arimaa match before the year 2020 [107].

Arimaa is difficult for computers to play because of several reasons. Firstly, the initial position of some of the pieces is left up to the player, this leads to a vast amount of starting combinations. Secondly, the number of possible outcomes in each turn run into the thousands, with a much bigger range of possibilities it becomes harder for computers to calculate all the different combinations. Thirdly, Arimaa uses less tactics than chess but is reliant upon understanding whether a particular position is more advantageous for one side or the other [108].

The intention for the creation of Arimaa and its accompanying challenge is the development of some new and radically different approaches to autonomous strategy game playing. Consequently, fostering new breakthroughs in the field of AI, followed by applications in many other fields [106]. The concepts behind agent-oriented development seem like a promising approach for this problem.

### 2.3.2 RoboCup Challenge

RoboCup is a popular, international research and education initiative that uses the game of soccer as a primary AI research domain. The long-term goal of RoboCup is to create a team of humanoid robots that can beat the best human soccer team by the year 2050 [93]. RoboCup has several different leagues, some involve playing physical soccer games and others simulated games. Building successful teams requires clever design, implementation and integration of hardware (if applicable) and software into a robustly functioning autonomous agent or robot. The agent (or multi-agent system) is also considered as the software acting as the ‘brain’ within robots. The complexity of the environment presented to an agent therefore is due to:

- Real-time mechanical control of components in the robot’s body such that they operate collectively and allow the robot to effectively play soccer.
- Infinite possible unexpected situations that may occur like sensor inaccuracies, collisions with other robots, and being out-numbered with opponents.
- Enforcement of FIFA soccer rules that must be followed by robots in order to play a reasonable game of soccer.
- Developing a number of low-level behaviours (e.g. dribbling the ball) in order to achieve efficient soccer playing.
- Integrating tactical behaviours for trying to out-skill opponents in order to score a goal.
- Working as a team in order to gain an advantage over the opponents.

RoboCup rules are designed such that they encourage technical and creative development via ensuring a fair competition while not directly describing how the game is played. In return, robots are required not to have specific expectations on their environment and not oblige other teams to make modifications to their robots to accommodate individual design decisions.

#### 2.3.2.1 Soccer Simulation League

Two teams of eleven virtual agents each play with each other, based on a realistic simulation of soccer as shown in figure 2.4. Each agent is realized as a separate process that communicates with the simulation server by sending motion commands and receiving back limited information about the situation in the form of sensor observations (noisy and partial) of the surrounding environment.

#### 2.3.2.2 Soccer Humanoid League

The humanoid league uses robots that act autonomously with no external control to play a modified version of soccer. Robots must consist of one head, two legs, two arms and one body. Additionally, they must walk using two legs (no wheels are allowed). The league follows the FIFA rules closely however a number of exceptions are included that are applicable when using robots. For example, games last 16 minutes, each team consists of two players with one being the goal keeper. The game is interrupted if a robot falls on the floor and cannot get up within



Figure 2.4: The Simulated Soccer Environment [103]p28

thirty seconds. Additionally, penalties are issued if a robot acts in an unfairly or damaging manner to other robots [91].

#### 2.3.2.3 Soccer Four Legged Robot League

The four-legged robot league uses cleverly designed robots by Sony called AIBO that look like a small puppy dog and are packed with features. The body of the AIBO has a total of twenty degrees of freedom divided between the mouth, legs, ears and tail. It has an in-built camera and microphones in order to see and hear as well as integrated acceleration and vibration sensors and a wireless link. The behaviour of AIBOs is governed by the *AIBO MIND 2* software that is provided on a removable Sony memory stick. This method provides Sony with an easier way to distribute upgrades. Additionally, enthusiasts are given the ability to create customized software for specific applications. This feature is exploited by this league as teams write soccer-playing behaviours and feed them to their AIBOs via memory sticks [102].

#### 2.3.2.4 Soccer Middle Sized Robot League

In the middle size robot league, two teams of four mid-sized robots with on-board sensors play soccer. The game involves two teams of up to six players, however the number of robots playing the game is dependent on the number of working robots available. For example, if one of the teams only has three players, the other team can only use up to four players even if it has more available. Additionally, robots must be built with safety in mind, not damage other robots, not pose a threat to people and not cause interference with sensor or communication devices of other robots [24].

#### 2.3.2.5 Soccer Small Sized Robot League

The small size robot league uses two teams of five robots. The design of the robots is left up to the researcher with the condition that it must fit inside a cylinder with a diameter 180mm and height of 150mm. Small size robots can have on-board vision sensors, or use a provided



global vision that consists of an overhead camera. The camera transmits a video signal of the entire field to a PC for analysis and communication to the robots [92].

### **2.3.2.6 Robotic Rescue**

Disaster rescue is one of the most serious social issues which involves very large numbers of heterogeneous agents, operating in an hostile environment. The RoboCup rescue project was launched in 1999. The main goal of the project is to promote research and development of advanced technologies of intelligent robotics and AI for emergency response and disaster mitigation. This domain involves researching aspects of multi-agent team work coordination, physical robots for search and rescue, information infrastructures, personal digital assistants for rescue workers, decision support systems, evaluation benchmarks for rescue strategies and simulation platforms. There are two sub-projects within this league, the Simulation Project and Robotics and Infrastructure Project. It is intended that future integration of these activities will yield better rescue operations [93].

## **2.3.3 Military Applications**

Agent technology has already been widely used for research in defence involving the use of Semi-Automated Forces (SAF) to conduct simulated war games. A war game is a game in which participants seek to achieve a specified military objective and are given pre-established resources and constraints. The purpose of running military simulations includes obtaining advice in regards to acquisition of military platform or systems, the creation and evaluation of tactics and concepts and, training military personnel [97].

Data fusion is a very important area in military research. This is because the latest military machinery are equipped with a many sensors. The data obtained from these sensors needs to be fused into information that is directly relevant to the machine operator. The Joint Directors of Laboratories (JDL) fusion model was developed in order to minimize confusion over the many elements comprising the fusion process. It provides a common frame of reference to understanding problems and recognizing common attributes within different problems of data fusion. The levels present in the model are defined as: Sub-Object Data Association and Estimation, Object Refinement, Situation Refinement and Impact Assessment. The operator is presented with information in a compact summarized format. An extension of data fusion involves the notion of Distributed Data Fusion (DDF) where there is an additional need to fuse local information with information received over a distributed communications network. Advantages of DDF include redundancy, and performance improvement [67].

Agent-based applications are being developed based on the process defined in JDL model with support of different levels of abstraction during the data fusion. Examples include the development of a framework for knowledge used in Unmanned Aerial Vehicles (UAV). The framework is used for timely generation of information about entities in the environment and internal to the UAV itself. Resulting prototypes have provided a more detailed insight into what is important in making such architectures a reality. For example, development of such problems leads to systems that are not strictly hierarchical and often involve complex interactions among the JDL layers, implying that it is not feasible to specify and implement each level separately

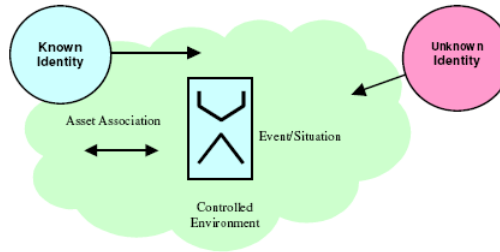


Figure 2.5: Situation Assessment [111]p730

[50].

Building accurate SA is the third level of the JDL. It applies equally to both land and air scenarios. In both cases protecting an asset from an unknown entity requires gathering SA on which to formulate a plan in order to react to the perceived threat. Figure 2.5 illustrates this example. Gathering SA is usually the first of a series of processes that are used for decision making in a military scenario, examples of processes that follow are: situation assessment, threat assessment, tactic selection and asset allocation [51, 86, 82, 78].

### 2.3.3.1 Military Simulations

Computers can be used to produce Digital Virtual Environments (DVE) where different military personnel can interact. Through the use of mock-up vehicles and high-fidelity visual systems, personnel can obtain a window into a virtual world populated by interacting simulated entities [10]. In battle simulations participants make battlefield decisions and a computer determines the results of those decisions [84]. There are three types of battle simulations [97]:

1. When there are no humans, simulations are purely virtual and are called *operations research*, in such cases the simulation can be allowed to run faster than real time in order to obtain quicker results.
2. When there are both humans and machines they are called *training simulations*.
3. When there are only humans they are called *experimentation*.

Computer Generated Forces (CGF) is a term that refers to characters in a military simulation that are not controlled by a human. Intelligent agents have been used for modeling the AI within CGF [97]. When some of the entities are controlled by human trainees, it is essential that they find the behaviour of computer-controlled entities realistic. It is also advantageous if the simulation is able to accommodate the command structure used by the army, this enables a hierarchical decomposition of the problems where high-level commanders are given objectives that are used to produce lower-level objectives for their subordinates. Information flows both up and down the command chain and therefore agents are required cooperate in order to achieve their orders. This decomposition, allows higher-level agents to work on long-term plans while individual agents carry out orders designed to achieve more immediate objectives [10].

Another complex problem in the battlefield domain is called battle planning. This involves an analysis of the actions of other agents, both friendly and hostile in order to identify counter

moves from opposing forces. Effective cooperation of agents in a simulated battlefield therefore requires the use of a Command and Control structure that includes the dynamic re-allocation of roles to suit unfolding circumstances [11].

*Plan recognition* is applicable in military simulations when agents are required to actively cooperate, but direct communication is unavailable due to being in a hostile environment. In such cases an agent must be able to develop models of the mental state of other agents based on observations and inferential reasoning. Plan recognition is also used to predict the actions of an enemy agent before they are executed. This allows for an agent to take preemptive defensive measures and also make efficient strikes. Achieving plan recognition requires the development of an *instructor agent* with a machine learning module that is able to interface with the simulation and observe the environment in detail. The agent forms a model based on the behaviours observed by other agents and human-controlled entities. Each of the behaviours observed is stored in a library of learned examples. During operation of an agent equipped with the learned data, a pattern-matching module continually attempts to recognize the behaviours of other agents with respect to learned behaviours [51].

The OneSAF Testbed Baseline (OTB) system was developed as a CGF platform that can represent a full range of military operations, systems, and control processes. It supports the control of individual combatants to battalion level, with a variable level of fidelity [27]. Recent research has focused on using agents to control different characters within the simulations. Each agent controls a single entity within the simulation with other agents operating at a higher-level and managing teams [69, 25]. The operator and commander can use the graphical representation of the wargame on the OTB graphical user interface to acquire an adequate level of situation awareness, undertake terrain appreciation and monitor the progress of the battle [69]. One of the applications of OTB is for conducting operation analysis studies and using intelligent agents to potentially reduce the amount of human resources needed. Agents have been developed that can analyze raw data, understand the situation, plan on how to carry out orders and assign tasks to subordinate agents. Cognitive Work Analysis (CWA) was used to model the military decision making process within agents. This was done to allow for traceability in the agent’s decision-making within the military structure. When objectives of an attack are provided to agents, they use reasoning that is based on military doctrine to produce plans and courses of action. This is followed by execution of the relevant tasks by sending commands back to the simulation [69].

DTank is a competitive environment that is useful for architectural comparisons of competitive agents, as well as comparing human and agent behaviour. It was created in order to provide a lightweight alternative to OTB and make it possible to develop agents using different agent frameworks for performance comparison. DTank was designed such that all agents have a universal interface for connection and human and software players having equal capabilities available to them [77]. DTank is intended to serve in three distinct roles. Firstly, as a teaching tool dTank provides an environment for experimenting with agent programming. Secondly, as a modeling tool dTank provides an environment for modeling both individual and team scenarios. Thirdly, as a developmental test-bed for advanced AI applications dTank can be used as a tool to investigate the usability of distributed multi-agent systems [28].

### 2.3.3.2 Military Hardware

The United States Government Defence Advanced Research Projects Agency (DARPA) has recently organized a competition that serves as a common ground for bringing together individuals and organizations from industry, government, academia and others to pursue a difficult but interesting challenge. The challenge involves the research and development of autonomous ground vehicles that can navigate to defined waypoints in an intelligent manner while dealing with unknown obstacles (eg. ditches, sandy ground, water) and different terrains like paved roads, trails, and off-road desert areas. The total length of the route is under 175 miles but the exact route is not revealed until two hours before the event begins. A big incentive for winning the competition is the prize for winning the challenge, a respectable US\$2 million. This figure was doubled from US\$1 million as an extra incentive because no teams were successful in completing the 2004 challenge. This has resulted in renewed, widespread attention to autonomous vehicle research while challenging the most capable and innovative researchers to produce breakthroughs in capability and performance [29]. Although great emphasis is placed on hardware specifications and the design and processing of sensors in the vehicles, some teams have used agents for implementing some of the more intelligent operations such as boundary detection, navigation, path planning and recovery [23, 89].

Other examples include the use of agents for managing the life support systems in a spacecraft [71] and personal assistant agents that *speak* with astronauts during missions for consolidating information and transmitting to other astronauts or back to Earth. In addition, an astronaut can use their personal assistant agents to issue commands to nearby robots to follow the astronaut or take photos or samples [87].

The development of UAVs is still in early stages but there has recently been increased interest in this field. Currently, UAVs are not fully autonomous or unmanned because they rely on a significant amount of help from ground operators [59]. Agent technology is being applied in different ways with UAVs, examples include software modeling [56], and implementing the control systems [59].

## 2.3.4 Business Applications

### 2.3.4.1 Pervasive Computing

Pervasive computing is a marriage between telecommunication and computing that is inspired from a vision of the future where digital content, applications and services are made available in an integrated, personalized way to users. Information is presented using a diverse range of methods and devices which requires manufacturers, content providers, infrastructure operators and others to collaborate their efforts [42]. Benefits offered by pervasive computing include mobile computing, mobile banking, directory assistance, news, smart appliances, and others [42]. Communication and connectivity is extremely important to pervasive computing environments, a key requirement is that the users *benefit* from the feature rich environment instead of being *overloaded* with choices. Intelligent agents have been suggested to help manage these services more transparently [65]. The intelligent home/office is a branch of research within pervasive computing that focuses on viewing a home/office as an intelligent agent with

a goal to maximize comfort and productivity for people while minimizing operational costs [26, 1, 94, 48].

#### **2.3.4.2 Space Exploration**

As budgets for space missions come under increased pressure, there is now great interest in saving costs through the use of new software technologies for automating different aspects of space operation. This is partly done by reducing the number of hours operators must be present in control centers [40]. Software agents are being designed to improve the autonomous behaviour of satellite and ground systems by automating on-board satellite functions, enabling satellites to operate longer without ground contact and autonomously react to local events. Additionally, new satellite designs involve a distributed agent system that encompass both satellite and ground operations that allows for tighter integration and autonomous cooperation [40].

#### **2.3.4.3 Air Traffic Control**

The extremely busy area of Air Traffic Control (ATC) has received large attention for integrating AI techniques. The use of cooperative, agent-based negotiation techniques have been proposed in order to efficiently resolve air traffic conflicts. Software agents running in each aircraft negotiate with one another to determine a safe and acceptable solution when a potential air traffic conflict is detected. The benefits of using such a system to handle the resolution of air traffic conflicts include an improvement in safety by reducing the workload of air traffic controllers. Furthermore, the robustness of the system is improved by the decentralization provided by the agents running in each aircraft, this reduces the dependence on a single ground based system to coordinate all aircraft movements. The pilots, passengers, and carriers benefit as well due to the increased efficiency of the solutions reached by negotiation [113].

Other approaches include:

- Analyzing ATC as a distributed system with a focus on the interface requirements for increasing safety and usability [38].
- Developing agents that can stand-in for human air traffic controllers in large-scale simulation studies. Agents use a high-level model to structure the ATC and are able to plan and formulate clearances [20].
- An agent based program that monitors and interacts with flights and presents suggestions to the operator before execution. Suggestions are timed, and when the time expires, the agent carries out the action. This results with a degree of autonomy in the agent [52].

## **2.4 Unreal Tournament Environment**

Unreal Tournament (UT) is a game that was co-developed by Epic Games, InfoGrames and Digital Entertainment that provides a simulated world with a complete 3D environment. The environment is formed using a pre-designed 3D map and a physics engine. A map can be

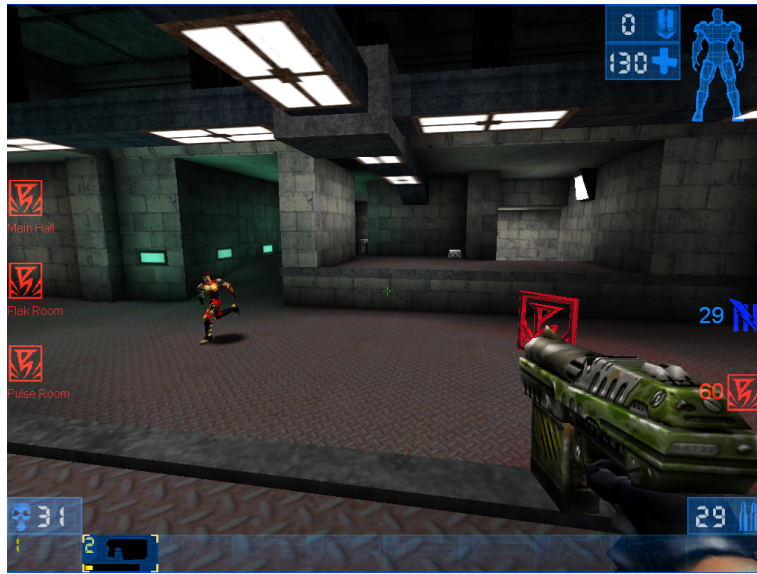


Figure 2.6: The Unreal Tournament Environment

customized in many different ways. For example it can have buildings, doors, elevators, water etc. The physics engine can also be customized such as varying the gravity to simulate a game in outer space, figure 2.6 shows a screen capture of UT in action.

UT was chosen as the testbed for the research described in this thesis after conducting a review of a number of other virtual environments, some of which are briefly introduced previously in this chapter. The review was conducted with three requirements in mind, which UT readily satisfies, they are:

1. The testbed shall be a powerful and robust simulation that allows intelligent agents to operate in real-time.
2. The testbed shall allow humans to access the simulation and interact with it, any changes initiated by humans shall influence the environment observed by the agent and vise-versa.
3. The testbed shall support team based scenarios in order to promote communication and collaboration in the following configurations:
  - Agents only teams.
  - Human only teams.
  - Agent-Human teams.

The first requirement is satisfied because UT is based on the commercially developed and supported unreal engine. The popularity of UT among gamers, has stressed tested the game and a number of official updates were released over the years to improve the features and stability. The original UT was released in 1999 with updated versions in 2003 and 2004 for improved graphics, physics and AI. Although, the original UT is dated compared to the newer versions, it provides a very stable, rich and cost-effective environment for agent development. There is still work to be done until external agents are able to out-perform the built-in bots within UT.

The second requirement is satisfied because UT was designed for gamers. It allows a human to join a game by controlling an entity within the game, the human views on the screen a simulated world and issues commands through the keyboard and mouse.

The third requirement is satisfied because some of the game types provided within UT require multiple players to work as a team in order to win the game. Teams may be comprised of any combination of humans and agents, this introduces an extra level of difficulty and leads into research on human-agent teaming.

### 2.4.1 Types of Gameplay

The most important feature of UT is the ability to apply modifications (called *mods*) to the game, there are three types of mods [90].

1. *Mutators*, are mini-mods that slightly change (or mutate) gameplay, allowing for interesting combinations. They have very limited functionality defined by the unreal engine's *Mutator* class and are required to follow certain rules, for example: they should be able to work with any other mutator, they should only change gameplay in a slight fashion and they should share resources with other mutators. Examples of mutators include the *LowGravity* mutator that reduces the gravity in the world by half in order to provide the feeling of floating in outer space and the *InstantGib* mutator that causes all characters to be killed with single shot whereas normally it would take several shots.
2. *GameTypes*, are bigger mods that do a lot more than mutator mods. GameTypes also do not impose rules like with mutators, because gametypes are executed separately. An example of a community-based UT gametype is called *FragBall*, it gives a soccer flavour to unreal tournament where players are required to carry a ball over to the enemy team's goals [99].
3. *TotalConversions*, are extreme mods that involve using the bare minimum unreal engine and creating an entirely different game. An example of a total conversion mod is called *AirFightUT* that allows players to fight with airplanes over open spaces [58].

The official UT ships with four types of gametypes, they are:

#### 2.4.1.1 DeathMatch

In Deathmatch, every player is working by themselves. The goal is to frag as many competitors as possible and also avoid being fragged by them. A player that reaches the game's frag limit first (or has the most frags when the time limit is reached) is the winner [35].

#### 2.4.1.2 Capture The Flag

Capture the flag introduces team play. The players are divided into two teams, blue and red. Each team has a base with a flag that they must defend. Points are scored when a team member captures the opposing team's flag by bringing it back to the team's base while their flag is on it [35].

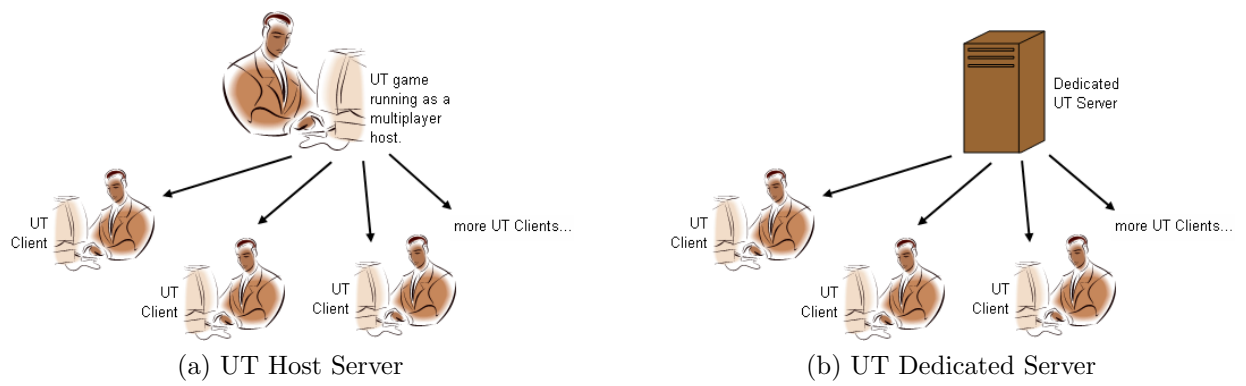


Figure 2.7: Unreal Tournament Multiplayer Mode

### 2.4.1.3 Domination

Two teams fight for the possession of several control points scattered throughout the map. To take a control point a player needs to simply touch it. When a team owns a control point their score increases steadily until the other team touches the control point [35]. The more control points that a team possesses the quicker the score will increase.

### 2.4.1.4 Assault

Players are divided into two teams, attackers and defenders. The attackers are attempting to achieve an objective such as blowing up a computer terminal or escaping a castle. The defender's job is to prevent them from doing this. The roles are then switched and the attackers become the defenders. The way to win an assault is to reach an objective in less time than the opponent's team. For example, if the first team completes the objectives in three minutes, then the opposing team needs to complete all objectives in less time. If they fail, they lose [35].

## 2.4.2 Multi-Player Mode

Unreal Tournament offers a powerful multi-player mode that allows two or more humans to enter the same game and play in the same team or in opposite teams. The multi-player mode has a server-client architecture. There are two types of UT servers. The first type is used when a player selects to host a multi-player game. This way the player enters the game and other players can then join the game using UT clients. The second way is to use a machine that runs a *dedicated* UT server. When dedicated mode is selected, UT exits from the client and runs only core components as a background process with minimal resources. Once the dedicated server has started, players can then join the game using clients. The differences are illustrated in figures 2.7a and 2.7b.



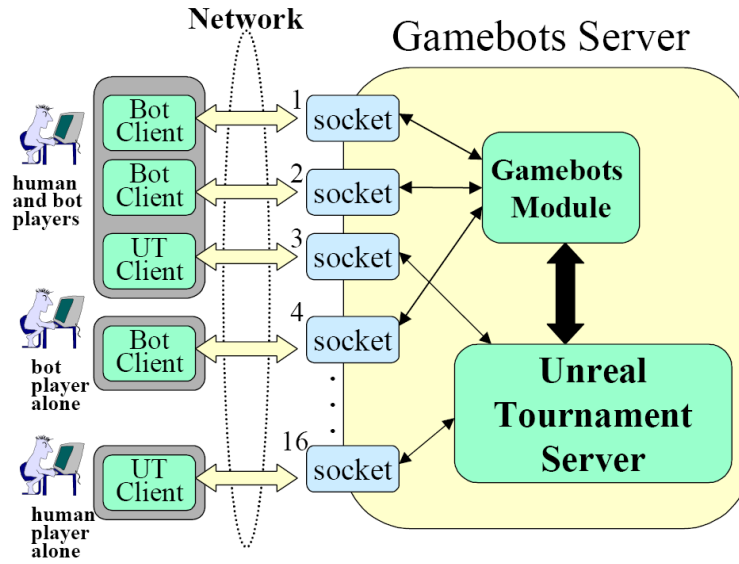


Figure 2.8: The Gamebots Server [3]p2

## 2.4.3 Interfacing with UT

### 2.4.3.1 Gamebots: An Open Source Server for UT

The aim of the Gamebots system is to allow the UT characters in the game to be controlled via network sockets connected to bot clients as shown in figure 2.8. The Gamebots server provides information to clients and based on this, the agent can decide for itself what actions it needs to take.

In the Gamebots documentation it is mentioned that a parser for the Gamebots protocol can make three assumptions.

1. All characters up to, but not including the first space are the message type.
2. Everything else in the message is of the form of attribute-value pairs enclosed by “{}”  
The attribute name consists of every character up to, but not including, the first space.  
The value includes all characters after the space terminating at the “}” and may include spaces.
3. Commands originating from a client need to follow the same basic format.

This means that the messages sent and received and from the Gamebots server are of the form [4]:

```
1 MSGTYPE {arg1 arg1val} {arg2 arg2val} ...
```

The rotation and location for a bot are sent to clients using UT’s own *units*. The measurement of UT’s units have no intuitive scale correspondence to real world measurements. However, in order to gain a perspective of the measurements, the length and angle dimensions are given in relative terms. Specifically, with respect to length, a player in UT has a 17 units collision radius and is 39 units tall. With respect to angles, a full 360 degree rotation is 65535 units. Locations are described in “x,y,z” format while rotations are described in “Pitch,Yaw,Roll” format.

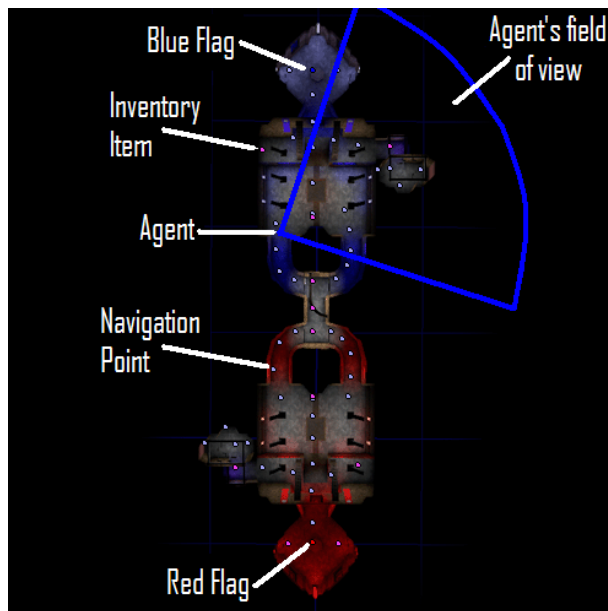


Figure 2.9: The TclViz Viewer

An agent can see the environment through synchronous messages that come in a batch at a configurable interval (the default interval is 10ms). At the start of a sync batch, the server transmits a “BEG” message. All messages received until a subsequent “END” message are assumed to be part of the synchronous batch, all these messages refer to a single discrete time instance of the game. A listing of the different types of synchronous messages is shown in appendix A.1.

An agent can sense important events that happen through asynchronous messages. These come as events happen in the game, however, in order to minimize confusion, they do not appear between a “BEG” and its associated “END” of a synchronous block. Asynchronous events can happen at any point in the game at random, less frequent intervals. Examples include taking damage, a message broadcast by another player, or running into a wall. A listing of the different asynchronous messages is shown in appendix A.2.

Bots take action in the world by transmitting commands to the server. They are formatted just like messages received from the server. The server parses the messages ignoring the case of characters and does not require a specific order of parameters. Some commands have persistent effects. For example, movement, once started, will continue until the bot reaches its destination. This is done in order to minimize the number of messages sent to the server as there is no reason to repeat commands. A listing of the command messages available to a client is shown in appendix A.3.

Some useful tools have been designed for use with Gamebots. Firstly, VizClients are visualization tools for observing and analyzing the behaviour of agents in the game, they show a bird’s-eye view of the map and also display the movements of the bots in the game in real-time. A screen-shot of a VizClient showing the CTF-Simple map with an agent present is shown in figure 2.9

The researchers that created Gamebots have also developed two sample bot clients in order to demonstrate how the system works. One of the bots is implemented in TCL and the other is implemented in Java. Both example bots can be used as a template for further development.

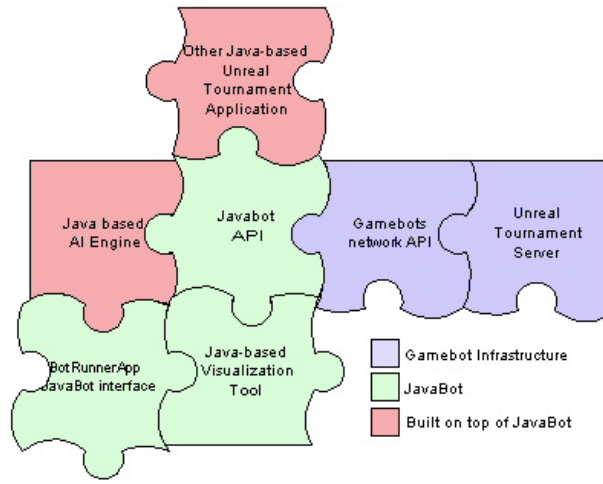


Figure 2.10: The Javabots Project [70]

#### 2.4.3.2 Javabots: A Java Client for Gamebots

Javabots provides a selection of Java packages that are designed for handling the low level communication to the Gamebots server. Figure 2.10 shows how the different components fit together in the development of a Java-based agent player in Unreal Tournament. The Gamebots infrastructure is coloured blue, and the components coloured green are contained in the Javabot package. The red components are those built on top of the Javabot package. It is intended that not much knowledge of the specifics of the Gamebots protocol is necessary to implement an agent player, just the ability to implement the Bot interface in the Javabot package.

The Javabots project also offers three example bots. Firstly, *ExampleBot* simply roams around the map by walking to different navigation points and inventory items. Secondly, *CMUJBot* is an extended *ExampleBot* that shoots at other players when it sees them. Thirdly, *HumanBot* provides a user interface for allowing a person to experiment with, learn and debug the Gamebots protocol.

## 2.5 Summary

The interface between an agent and its environment can be simplified down to a loop involving agents receiving sensations from the environment, making decisions, and taking actions that affect the environment. Each environment can have a number of features based upon being observable, being deterministic, being episodic, being static, being discrete and being comprised of multiple agents or not.

Agent technology has been used within many different types of applications. In computer games, for improving the AI of computer generated players. In research and education initiatives, for the clever design and implementation of hardware and software into robustly functioning autonomous robots. In defence, for imitating humans as semi-automated forces within simulated war games. In business, for helping people manage digital content, applications and services in an integrated, personalized way. In space exploration, for enabling satellites to operate longer without ground contact and autonomously react to local events.

Finally, in air traffic control, for resolving air traffic conflicts and consequently boosting the safety for passengers.

We have now seen the different properties of environments that give rise to their complexity and have developed an appreciation for the need to use learning in complex application domains. The following chapter will introduce the concepts behind the way that agents perform reasoning and describe a number of learning techniques developed in the field of artificial intelligence.

# Chapter 3

## Reasoning and Learning

This chapter reviews the theoretical foundations of agent reasoning and learning. It covers a review of techniques on reasoning as performed by machines, as well as models that illustrate how humans make every day decisions. It then discusses a number of learning and adaptation methods as developed in artificial intelligence research and gives a brief overview of some previous research on agents and learning.

### 3.1 Introduction

The environments described in the previous chapter illustrated the need for learning when agent systems are required to interact with complex environments. This chapter now describes how agents and humans are understood to perform reasoning and learning when they are faced with a particular environment.

Reasoning is understood as the thinking process that occurs within an agent that needs to make a particular decision. This topic has been tackled via two parallel directions with two different schools of thought. The first school of thought focuses on how agents can perform rational reasoning where the decisions made are a direct reflection of knowledge. The advantage of this approach is that decisions made by an agent can be understood simply by looking within its internal data structures, as the agent only makes decisions based on what it knows. This process includes maintaining the agent's knowledge base such that it contains accurate information about its environment. This is done by performing operations to keep all knowledge consistent. Decisions are made through a collection of rules applied on the knowledge base that define what should occur as knowledge changes.

Another school of thought is concerned with the way that humans perform reasoning and apply any concepts developed to agent technology. Humans are known to perform practical reasoning every day, their decisions are based on their desires and their understanding in regards to how to go about achieving them. The process that takes place between observing the world, considering desires and taking actions can be broken up into four main stages, each of which consists of a number of smaller components.

Through learning, it becomes possible to create agents that are able to change the way that they were originally programmed to behave. This can be advantageous when an agent is faced with a situation that it does not know how to proceed. Furthermore, it is useful when

an agent is required to improve its performance with experience.

Section 3.2 describes the correlation between reasoning and behaviour, a number of reasoning models from previous research are also reviewed and their associated behaviour is analyzed. Section 3.3 then gives a comprehensive outlook into the features that constitute the ability to learn, followed by different types of learning techniques.

## 3.2 Reasoning and Behaviour

Research on artificial reasoning and behaviour has been tackled from different angles that can be categorized along two main dimensions as shown in figure 3.1. The vertical dimension illustrates the opposing nature of reasoning and behaviour that correspond to thinking versus acting respectively. This is an important concept in *every* application using AI techniques. Great emphasis is given to the balance between processing time for making better decisions, and the required speed of operation. Approaches falling on the left side of the vertical axis are based on how humans reason and behave while approaches falling on the right side are concerned with building systems that are rational, meaning that they are required to think and act as best they can, given their limited knowledge [95].

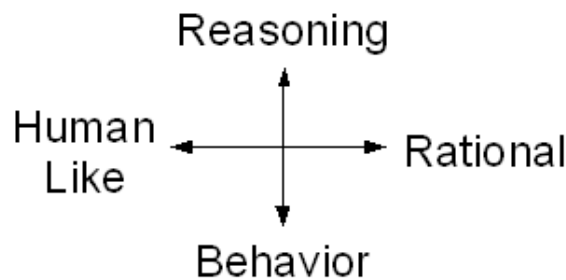


Figure 3.1: Reasoning Dimensions

This section reviews a number of approaches that have been described in literature on artificial reasoning and behaviour modeling.

### 3.2.1 Rational Reasoning

#### 3.2.1.1 Representation and Search

The way that information is *represented* and *used* for intelligent problem solving forms a number of important but difficult challenges that lie within the core of AI research. Knowledge representation is concerned with the principles of correct reasoning. This involves two parallel topics of research. One side is concerned with the development of formal representation languages with the ability to maintain consistent knowledge about the world, the other side is concerned with the development of reasoning processes that bring the knowledge to life. The output of both of these areas results in a Knowledge Base (KB) system. KBs try to create a *model* of the real world via the collection of a number of *sentences*. An agent is normally able to add new sentences to the knowledge base as well as query the KB for information. Both of these tasks may require the KB to perform *inference* on its knowledge, where an inference

is defined as the process of deriving new sentences from known information. An additional requirement of KBs is that when an agent queries the KB, the answer should be inferred from information previously added to the KB and not from unknown facts. The most important part of a KB is the *logic* in which its sentences are represented. This is because all sentences in a KB are in fact expressed according to the *syntax* and *semantics* of the logic's representation language. The syntax of the logic is required for implementing well formed sentences while the semantics define the truth of each sentence with respect to a model of the environment being represented [95].

Problem solving using KBs involves the use of search algorithms that are able to search for solutions between different states of information within the KB. Searching involves starting from an initial state and expanding across different successor state possibilities until a solution is found. When a search algorithm is faced with a choice of possibilities to consider, each possibility is thoroughly searched before moving to the next possibility. Search however has a number of issues [68]:

- Guarantee of a solution being available.
- Termination of the search algorithm.
- The optimality of a particular solution found.
- The complexity of the search algorithm with respect to the time and memory usage.

State space analysis is done with the use of graphs. A graph is a set of nodes with arcs that connect them, each node can have a label to distinguish it from another node and arcs can have directions to indicate the direction of movement between the nodes. A path in the graph connects a sequence of nodes with arcs and the root is a node that has a path to all other nodes in the graph. An example of a graph illustrating the different possible states of a simple KB representing the board game of tic-tac-toe is shown in figure 3.2.

There are two ways to search a state space, the first way is to use data-driven search, which starts with a given set of *facts* and *rules* for changing states. The search proceeds until it generates a path that leads to the goal condition. Data driven search is more appropriate for problems in which the initial problem state is well defined, or there are a large number of potential goals and only a few facts to start with, or the goal state is unclear [68].

The second way is to use goal-driven search by which the search starts by taking the goal state and determining what conditions must be true to move into the goal state. These conditions are then treated as *subgoals* to be searched. The search then continues backwards through the subgoals until it reaches the initial facts of the problem. Goal driven search is more appropriate for problems in which the goal state is well defined, or there are a large number of initial facts making it impractical to prefer data driven search, or the initial data is not given and must be acquired by the system [68].

The choice of which of the options to expand first is defined by the algorithm's *search strategy*. Two well known search strategies are: Breadth-first, where all successors of a given depth are expanded first before any nodes at the next level. Depth-first search involves expanding the deepest node for a particular option before moving to the next option. There are also

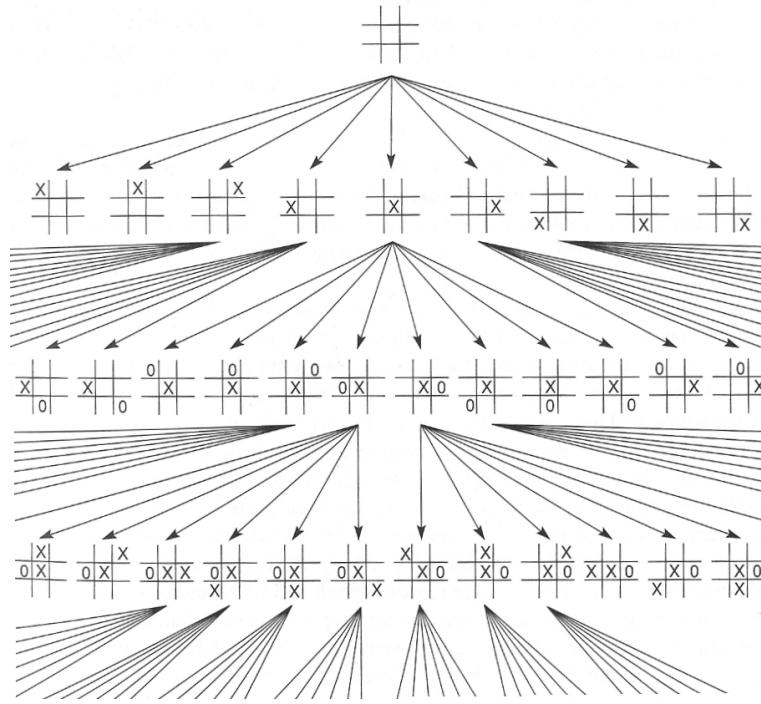


Figure 3.2: Searching The State Space of Tic-Tac-Toe [68],p43

strategies that include both elements, for example defining a depth limit for searching in a tree. It is also possible to use *heuristics* to help with choosing branches that are more likely to lead to an acceptable solution. Heuristics are usually applied when a problem does not have an exact solution or the computational cost to find an exact solution is too big. They reduce the state space by following the more promising paths through the state space [95].

An additional layer of complexity in knowledge representation and search is due to the fact that agents almost never have access a truly observable environment. Which means that agents are required to act under *uncertainty*. There are two techniques that have been used for reasoning in uncertain situations. The first involves the use of *probability theory* in assigning a value that represents a degree of belief in facts in the KB. The second method involves the use of *fuzzy sets* for representing how well a particular object satisfies a vague description [95].

### 3.2.1.2 Expert Systems

Knowledge-based reasoning systems are commonly called *expert systems* because they work by accumulating knowledge extracted from different sources, and use different strategies on the knowledge in order to solve problems. Simply put, expert systems try to replicate what a human expert would do if faced with the same problem. They can be classified into different categories depending on the type of problem they are used to solve [68]:

**Interpretation:** Making conclusions or descriptions from collections of raw data.

**Prediction:** Predicting the consequences of given situations.

**Diagnosis:** Finding the cause of malfunctions based on the symptoms observed.

**Design:** Finding a configuration of components that best meets performance goals when considering several design constraints.



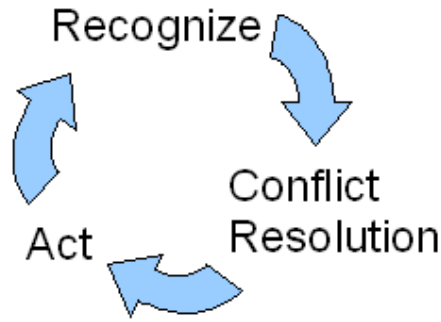


Figure 3.3: Production Systems Operation Cycle

**Planning:** Finding a sequence of actions to achieve some given goals using specific starting conditions and run-time constraints.

**Monitoring:** Observing a system’s behaviour and comparing it to its expected behaviour at run-time.

**Debugging:** Finding problems and repairing caused malfunctions.

**Control:** Controlling how a complex system behaves.

A common way to represent data in expert systems is using first-order *predicate calculus* formulae. For example, the sentence “If a bird is a crow then it is black” is represented as:

$$\forall X(crow(X) \rightarrow black(X))$$

### 3.2.1.3 Production Systems

Production systems are based on a model of computation that uses search algorithms and models human problem solving. Production systems consist of *production rules* and a *working memory*. Production rules are pre-defined rules that describe a single segment of problem-solving knowledge. They are represented by a *condition* that determines when the production is applicable to be executed, and an *action* which defines what to do when executed. The working memory is an integrated KB that contains an ever-changing state of the world.

The operation of production systems generally follows a recognize-act cycle as illustrated in figure 3.3. Working memory is initialized with data from the initial problem description and is subsequently updated with new information. At every step of operation, the state presented by the working memory is continuously captured as *patterns* and applied to conditions of productions. If a pattern is recognized against a condition, the associated production is added to a *conflict set*. A *conflict resolution* operation chooses between all enabled productions and the chosen production is *fired* by executing its associated action. The actions executed can have two effects. Firstly, they can cause changes to the agent’s environment which indirectly changes the working memory. Secondly, they can explicitly cause changes in the working memory. The cycle then restarts using the modified working memory until a situation when no subsequent productions are enabled. Some production systems also contain the means to do *backtracking* when there are no further enabled productions but the goal of the system has still not been

Iteration	Working memory	Conflict set	Rule fired
0	cbaca	1,2,3	1
1	cabca	2	2
2	acbca	2,3	2
3	acbca	1,3	1
4	acabc	2	2
5	aacbc	3	3
6	aabcc	None	Halt

Table 3.1: Execution of a Simple Production System [68],p173

reached. Backtracking allows the system to work backwards and try some different options in order to achieve its goal [68].

A very simple example of a production system is one that rearranges the string “cbaca” into alphabetical order. Such a system can have three productions, each one rearranging two letters in the correct order as shown below. The execution of the production system to solve this problem is illustrated in table 3.1 [68].

1.  $ba \rightarrow ab$
2.  $ca \rightarrow ac$
3.  $cb \rightarrow bc$

## 3.2.2 Human Reasoning

### 3.2.2.1 Practical Reasoning

Practical reasoning is concerned with studying the way that humans reason about what to do in everyday activities and applying this to the design of intelligent agents. Practical reasoning is specifically geared to reasoning towards actions, it involves weighing conflicting considerations of different options that are available depending on what a person desires to do [115].

Practical reasoning can be divided into two distinct activities as illustrated in figure 3.4. The first activity is called *deliberation*, it involves deciding on what state to achieve. The second activity is called *means-ends reasoning* and it involves deciding on how to achieve this state of affairs [115].

The central component of practical reasoning is the concept of *intention* because it is used to characterize *both* the action and thinking process of a person. For example ‘intending to do something’ characterizes a persons thinking while ‘intentionally doing something’ characterizes the action being taken [15].

The precursors of an intention are a persons’s *desires* and *beliefs* and hence all of the beliefs, desires and intentions must be *consistent*. In other words, intending to do something must be associated with a relevant desire, as well as the belief that the intended action will help to achieve the desire. Maintaining this consistency is challenging due to the dynamic nature of desires and beliefs. Desires are always changing according to internal self-needs while beliefs are constantly updated using information obtained from senses through a process called belief

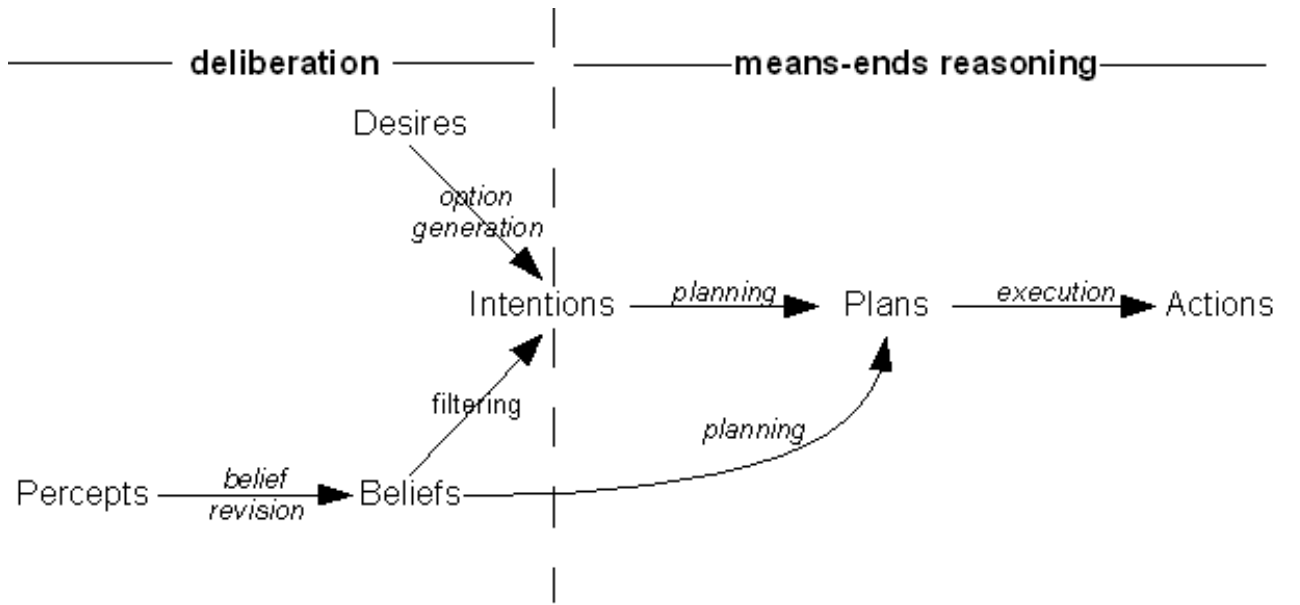


Figure 3.4: BDI Reasoning Process

revision, from the external environment. The distinction between desires and intentions is emphasized in this thesis by using specific completion terminology for each one as shown in table 3.2. Agents developed using Beliefs, Desires and Intentions are known as BDI intelligent agents.

Completion Terminology	
<b>Desires</b>	Satisfied
<b>Intentions</b>	Achieved

Table 3.2: BDI Behaviour Completion Terminology

Forming an intention involves performing two concurrent operations. Firstly, *option generation* uses the current desires to generate a set of possible alternatives. Secondly, *filtering* chooses between these alternatives based on the current intentions and beliefs [115]. An intention also requires assigning a degree of *commitment* toward performing a particular action or set of actions in the future [15]. There are four important characteristics emerging by this commitment as summarized below [115]:

- Intentions drive means-ends reasoning by forcing the agent to decide on how to achieve them.
- Intentions persist by forcing a continuous strive to achieve them. Hence, after a particular action has failed, other alternative actions are attempted until it comes to be believed that it is not possible to achieve the intention, or the relevant desire is not longer present.
- Intentions constrain future deliberation because it is not necessary to consider desires that are inconsistent with the current intentions.
- Intentions influence beliefs by introducing future expectations. This is due the requirement of believing that a desired state is possible before and during execution the intention to satisfy it.

Formula	Interpretation
$(\text{Bel } i \ \varphi)$	Agent $i$ believes $\varphi$
$(\text{Bel } \textit{george} \ \textit{Raining}(\textit{adelaide}))$	The agent $\textit{george}$ believes it is raining in $\textit{adelaide}$
$(\exists i. \text{Bel } i \ \textit{Raining}(\textit{adelaide}))$	Somebody believes it is raining in $\textit{adelaide}$
$(\forall i. \text{Bel } i \ \textit{Raining}(\textit{adelaide}))$	Everyone believes it is raining in $\textit{adelaide}$
$(\text{Des } i \ \varphi)$	Agent $i$ desires $\varphi$
$(\text{Des } \textit{irene} \ \textit{Find}(\textit{job}))$	Agent $\textit{irene}$ desires to find a job
$(\text{Int } i \ \varphi)$	Agent $i$ intends $\varphi$
$(\text{Des } \textit{irene} \ \forall i. (\text{Int } i \ \textit{Hire}(\textit{irene})))$	Agent $\textit{irene}$ desires that everyone will intend to hire her
$\bigcirc \varphi$	$\varphi$ is true next
$\Diamond \varphi$	$\varphi$ is eventually true
$\Box \varphi$	$\varphi$ is always true
$\exists x. \Box \textit{PrimeMinister}(x)$	There is always someone who is Prime Minister
$\forall x. \Diamond \textit{Free}(x)$	Everybody will be free at some point in the future
$(\text{Happens } \alpha)$	Action $\alpha$ happens next
$(\text{Achvs } \alpha \ \varphi)$	Action $\alpha$ occurs and achieves $\varphi$
$(\text{Agtvs } \alpha \ g)$	Group $g$ is required to do action $\alpha$
$(\text{Int } i \ \varphi) \Rightarrow (\text{Des } i \ \varphi)$	If an agent intends $\varphi$ then it desires it
$(\text{Int } i \ \varphi) \Rightarrow (\text{Bel } i \ \varphi)$	If an agent intends $\varphi$ then it believes its possible

Table 3.3: Examples Of The *LORA* Logic [115]

The process that occurs after forming an intention in order to take action is identified as *planning*, it involves selecting and advancing through a sequence of plans that dictate what actions to take [15]. Plans are understood to consist of a *pre-condition* that characterizes the state in which a plan is applicable for execution, and a *post-condition* that characterizes the resulting state after executing the plan. Finally, a *body* containing the *recipe* defining the actions to take [115]:

Recently, researchers have started to focus on the definition of a formal logic that can be used to describe an agent's beliefs, desires and intentions. For example, Wooldridge's *Logic of Rational Agents (LORA)* [115] provides a number of techniques to mathematically derive new sentences from combinations of other sentences as well as define some important properties of BDI agents. Table 3.3 contains a number of examples of *LORA* sentences.

From the theory of practical reasoning, researchers have been able to develop intuitive agent development architectures. The transition between the theory and implementation has required the identification of equivalent software constructs for each of the BDI components. The details of the ways that different projects have implemented the BDI model are described in chapter 4.

### 3.2.2.2 Cognitive Systems Engineering

During the design and implementation of systems, Cognitive Systems Engineering takes into account that systems will be used by humans. It acknowledges that humans are dynamic entities that are part of the system itself but cannot be modeled as static components of a system. When humans use a system they adapt to the *functional characteristics* of the system. In addition, sometimes they can modify the system's functional characteristics in order to suit

their own needs and preferences. This means that in order to understand the behaviour of the system once the adaptation has happened is to abstract the structural elements into a purely functional level and identify and separate the functional relationships. This concept can best be understood using a simple example from Rasmussen [88].

When a novice is driving a car, it is based on an instruction manual identifying the controls of the car and explaining the use of instrument readings, that is, when to shift gears, what distance to maintain to the car ahead (depending on the speed), and how to use the steering wheel. In this way, the function of the car is controlled by discrete rules related to separate observations, and navigation depends on continuous observation of the heading error and correction by steering wheel movements. This aggregation of car characteristics and instructed input-output behaviour makes it possible to drive; it initiates the novice by synchronizing them to the car functions. However, when driving skill evolves, the picture changes radically. Behaviour changes from a sequence of separate acts to a complex, continuous behavioural pattern. Variables are no longer observed individually. Complex patterns of movements are synchronized with situational patterns and navigation depends on the perception of a field of safe driving. The drivers are perceiving the environment in terms of their driving goals. At this stage, the behaviour of the system cannot be decomposed into structural elements. A description must be based on abstraction into functional relationships [88],p7.

A new design approach is introduced that shifts away from the traditional software engineering perspective to a functional perspective. There are two different ways to define functional characteristics. Firstly, relational representations are based on mathematical equations that relate physical, measurable environments. Secondly, casual representations are connections between different events. Rasmussen [88] presents a framework that makes it possible to relate conceptual characteristics. The framework takes into account that in order to bridge system behaviours into human profiles and preferences, several different perspectives of analysis and languages of representation are needed as illustrated in figure 3.5.

*Work Domain Analysis* is used to make explicit the goals, constraints and resources found in a work system. They are represented by a general inventory of system elements that are categorized by functional elements and their means-ends relations. The analysis identifies the structure and general content of the global knowledge of the work system. *Activity analysis* is divided into three different dimensions. Firstly, activity analysis in *domain terms* focuses on the freedom left for activities after the constraints posed by time and the functional space of the task. Generalizations are made in terms of objectives, functions and resources. Secondly, activity analysis in *decision terms* use functional languages to identify decision making functions within relevant tasks. The results of this analysis are used to identify prototype knowledge states that connect different decision functions together. Thirdly, *mental strategies* are used to compare task requirements with cognitive resource profiles of the individual actors and how they perform their work, this supplies the designer with mental models, data formats and rule sets that can be incorporated into the interface of the system and used by actors of varying expertise and competence [88].

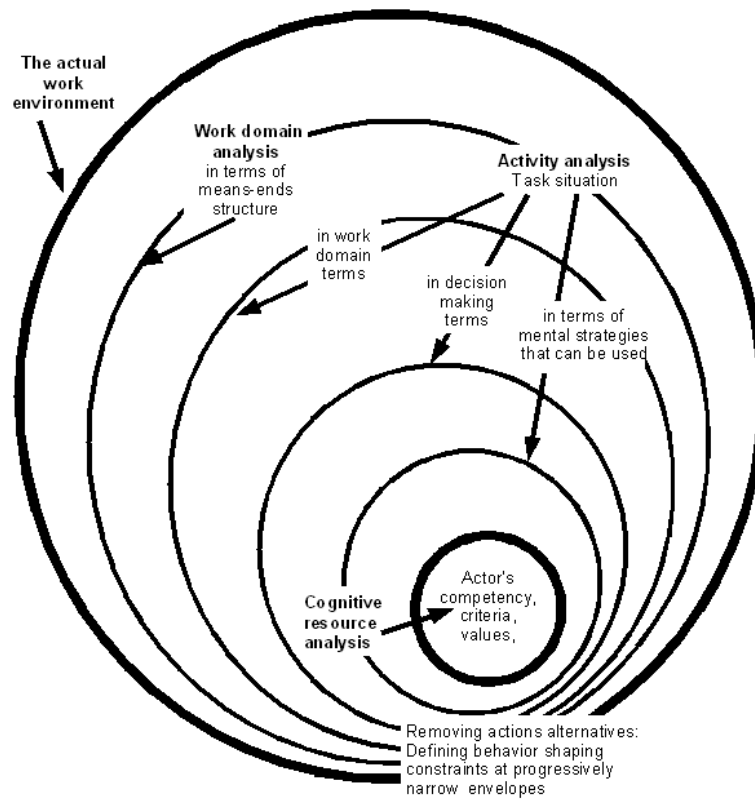


Figure 3.5: Relating Work Environment to Cognitive Resource Profiles of Actors [88],p25

*Work Organization Analysis* is used to identify the actors involved in the decisions of different situations. This is done by finding the principles and criteria that govern the allocation of roles among the groups and group members. This allocation is dynamically dependent on circumstances and is governed by different criteria such as actor competency, access to information, minimizing communication load and sharing workload.

*Social Organization Analysis* focuses on the social aspect of groups working together. This is useful for understanding communication between team members, such communication may include complex information, like intentions used for coordinating activities and resolving ambiguities or misinterpretations. Finally, *User Analysis* is used to help judge which strategy is likely to be chosen by an actor in a given situation focusing on the expertise and the performance criteria of each actor [88].

Rasmussen proposes a framework for representing the various states of knowledge and information processes of human reasoning, it is called the *Decision Ladder* and is illustrated in figure 3.6. The ladder models the human decision making process through a set of generic operations and standardized key nodes or states of knowledge about the environment. The circles illustrated are states of knowledge and the squares are operations. The decision ladder was developed as a model for performing work domain analysis, however, the structure of the ladder is generic enough to be used as a guide in the context of describing agent reasoning.

The decision ladder can be segmented into three levels of expertise [88]. The *skill* (lowest) level represents very fast, automated sensory-motor performance and it is illustrated in the ladder via the heuristic shortcut links in the middle. The *rule* (medium) level represents the use of rules and/or procedures that have been pre-defined, or derived empirically using

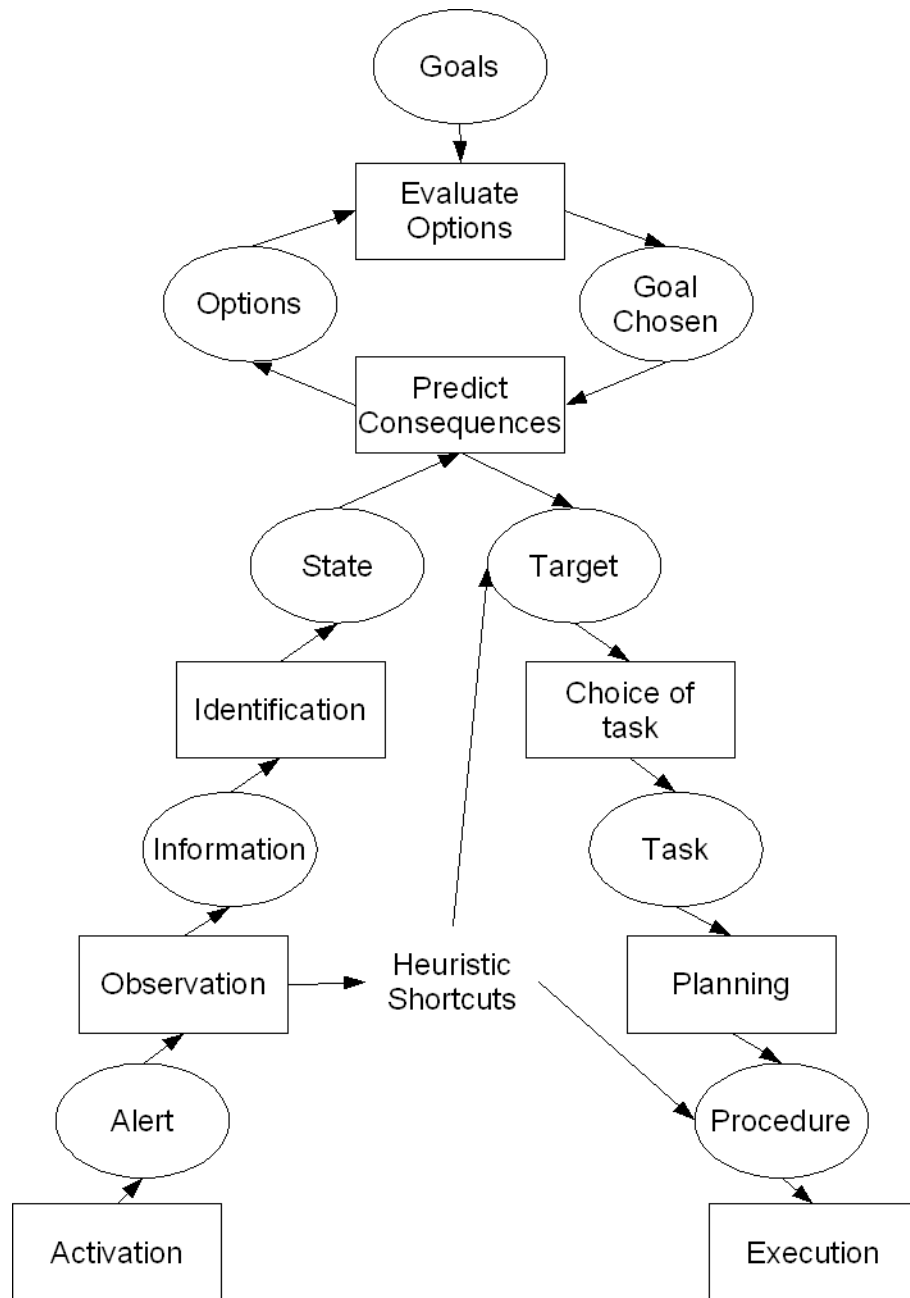


Figure 3.6: Rasmussen's Decision Ladder [88],p65

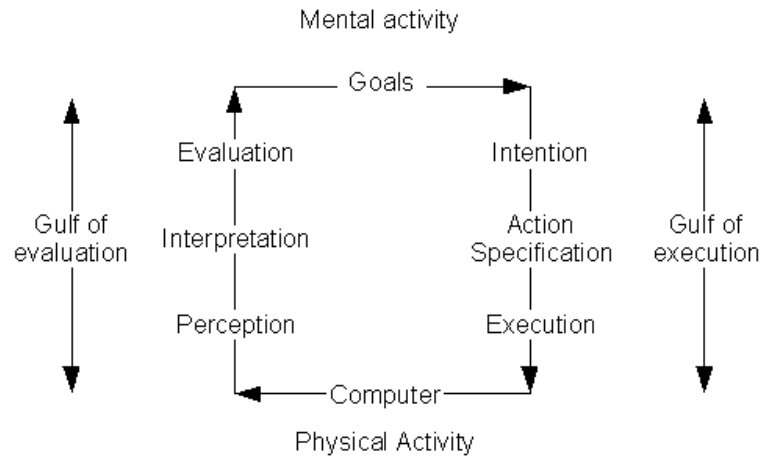


Figure 3.7: Gulfs of Execution and Evaluation, adapted from [88],p127

experience, or communicated by others, it traverses the bottom half of the ladder. Finally, the *knowledge* (highest) level represents behaviours during less-familiar situations when someone is faced with an environment where there are no rules or skills available, in such cases a more detailed analysis of the environment is required with respect to the goals the agent is trying to achieve, the entire ladder is used for this case. Urlings [111] has also recently added that learning involves the crossing of skill level boundaries. When people are placed in an unfamiliar environment they would first operate in the knowledge level. As they gain experience, they build a mental rule-base for situations encountered. When the rule-base becomes large and is used exclusively, reasoning drops to the rule level. In a similar way, when the mental rules provide good performance, confidence on using them increases, and they are automatically performed with less or no thinking involved. This means reasoning has crossed to the skill level of operation.

Another important topic considered in cognitive systems engineering is the issue of human-computer interaction. The main problem encountered is that a person's goals are expressed in psychological terms that are relevant to the person while the states of a system are expressed in physical, quantitative terms. This mismatch is characterized as two gulfs between the person and the machine as shown in figure 3.7. The gulf of execution refers to the gap between a person's goals (what they intend to achieve) and inputs recognized by the computer. Conversely, the gulf of evaluation refers to the gap between what the computer outputs and what the human understands. Bridging these gulfs is usually left to users to slowly adapt to, placing them into an increased cognitive burden during operation. An important feature to notice is the reasoning loop between the person and the machine. It provides an indication with respect to where a person's goals and intentions are located in the reasoning loop. This is used for the contributions put forth in this thesis.

### 3.2.2.3 Tactical Assessment

Col. John R. Boyd is considered as one of the premier military theorists of the twentieth century [22]. Boyd's contributions begin with his early work on his *Energy Maneuverability* theory that is able to plot the basic characteristics of a military plane such as how far, how



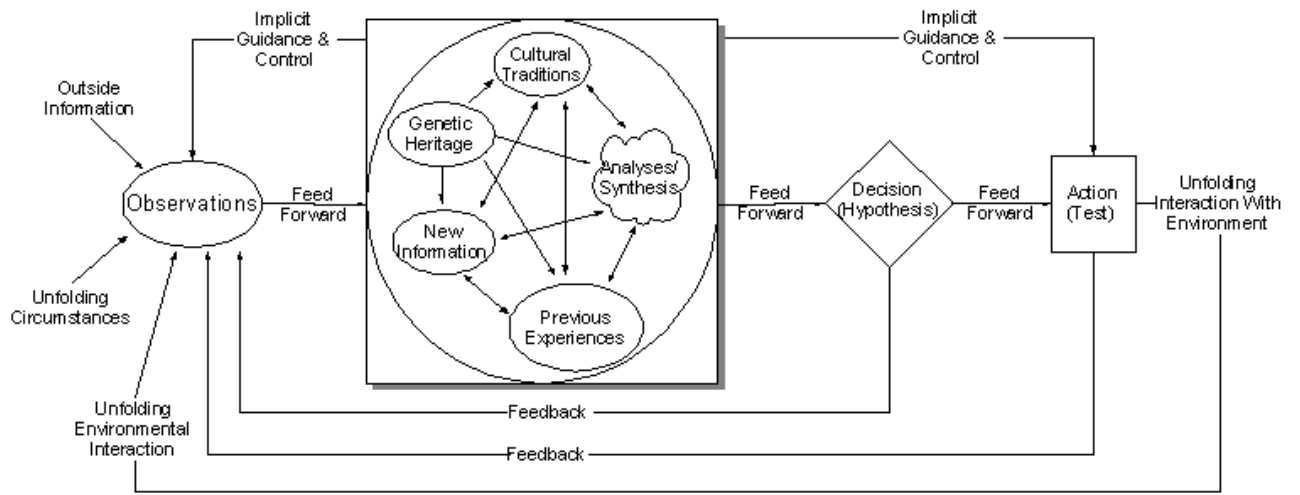


Figure 3.8: Boyd's Observe, Orient, Decide and Act Loop [14]

fast, how high, as well as its maneuverability at different conditions like different altitudes, g-forces and turning radii. The energy maneuverability theory served as the cornerstone for the design of the US Air Force's F-15 and F-16 fighter planes. Later works are based on an analysis of famous military strategists such as Sun Tzu, Alexander and Napoleon [47].

Boyd summarized his entire work into five slides called *The Essence of Winning and Losing* where he presented the final revision of the Observe Orient Decide and Act (OODA) loop as shown in figure 3.8. The OODA loop reduces the decision cycle into four steps [22]:

**Observe:** Gather data from the surrounding environment

**Orient:** Create a mental model (or mental map) of the circumstances surrounding the decision.

**Decide:** Make the decision.

**Act:** Implement the decision.

According to Boyd's theory, the goal of a strategist is to operate at a faster reasoning tempo and proceed through the OODA steps more rapidly than the enemy. This allows the strategist to operate inside the mind-time-space of the adversary, when this happens the enemy *appears to move in slow motion*. Hence, the side with the faster OODA loop will triumph in a conflict. Improving the OODA loop speed involves proceeding through its four stages more rapidly. The speeds of the first two stages are heavily dependent on the speed of information gathering and processing, whereas the speed of the last two stages are dependent on the confidence in the output of the previous two stages [22].

Genetic heritage, cultural traditions and previous experiences present in the orientation stage illustrate the possession of an implicit repertoire of psychological skills that have been shaped by past experiences. Analysis and synthesis is performed via an implicit cross-referencing process of projection, empathy, correlation and rejection. It evolves new behaviour repertoires to deal with unfamiliar phenomena or unforeseen changes. The orientation stage therefore embodies the ability to comprehend, shape, adapt to, and in turn be shaped by an unfolding, evolving reality that is uncertain, ever-changing and unpredictable [14].

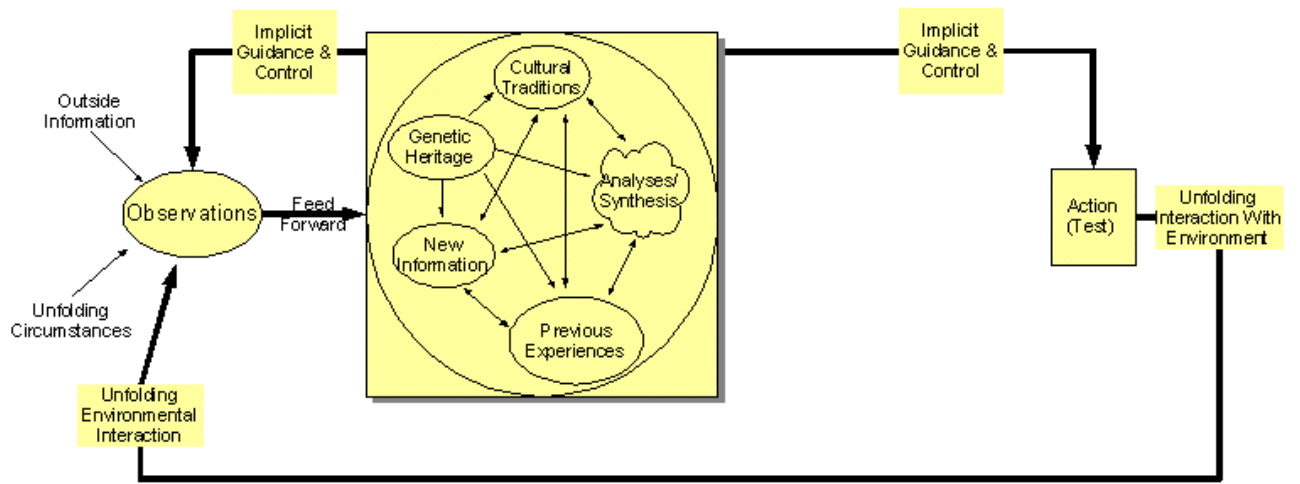


Figure 3.9: Pumping up The OODA Loop Speed [30]

The OODA loop also identifies a number of important feedbacks in the decision process. It explicitly shows the feedback caused by the environment after acting on it. It also shows how the observation stage shapes both the decision and action stages through *implicit guidance* feedbacks extending from orientation to all other stages of the process. These feedbacks can also introduce short-cuts in the decision process that cause it to loop faster as shown in figure 3.9.

### 3.3 Learning and Adaptation

The concept of learning is vague and has many definitions, this is because learning is used for problem solving in many different areas. Maystel [75] has summarized the important points of what learning is from several definitions found in different sources such as dictionaries, encyclopedias, and researchers working in different areas. Arguably learning should have the following features [75]:

1. Should be a process of obtaining skill where “skill” means a pre-existing program that produces useful behaviour.
2. Produces alteration of an individual behaviour where “alteration” implies that the program that produces the behaviour should be changed.
3. Produces a change in a behavioural potentiality where “potentiality” refers to a storage of the learning program.
4. Develops an inner program that is better adapted to its task where “better” implies a measurement should be built into the learning mechanism.
5. It is enabled such that it can perform a task more efficiently where “efficiency” implies that the performance should be measured.
6. It changes the quality of the output behaviour where “quality” implies that the output behaviour should be measured.

7. Making useful changes in mind where “useful” refers to measurement and “mind” refers to the ability to store and modify the learning program.
8. Constructing and modifying representations where “constructing” implies the ability to construct new learning programs.
9. Learning is essentially discovery (creating new programs).
10. Formation of new classes and categories via generalization.
11. Changing the algorithm where “algorithm” means the learning program.
12. Forcing the system to have a particular response to a specific signal by repeating the input signals.
13. Evolving the model of the world.

Maystel used all of the above features to define learning as:

Learning is a process based upon experience of functioning of intelligent systems (including their sensory perception, world representation, behaviour generation, value judgment, communication, etc.) which provides a better value(s) of the cost-functional(s) considered to be a subset of the (externally given) assignment for the intelligent system [75]p10.

According to [68], learning is a change in a system that allows it to perform better on the second time when a task of the system is repeated, or on a similar task. The main problem here is selecting the changes to a system that will change the performance of the system. An extension to this problem is changing system parameters in a way that will improve and not hinder the system performance.

The above is a broad description of what learning is and as it can be seen that the concept of learning is not trivial. So the next question is: What technologies are already available that attempt to introduce some learning into systems? This is a question answered by research in AI and the following sections in this chapter provide a brief description of current AI theories on learning.

### **3.3.1 Machine Learning**

#### **3.3.1.1 Symbol Based Learning**

Symbol-based learning relies on learning algorithms that can be characterized and grouped into dimensions as shown in table 3.4. Each dimension focuses on a different sub-area of symbol-based learning [68].

The following example [68] shows the concept of identifying an arc when three blocks are arranged in a different way, it illustrates what is considered to be an arc and what is not. Figure 3.10a is an arc because it shows two bricks supporting a third brick on them. Figure 3.10b is also an arc but the two bricks support a triangle instead of another brick, this suggest that it is possible to form an arc supporting any polygon on the two bricks. Figure 3.10c

Grouping	Description
Data and Goals	Here the learning problem is described according to the goals of the learner and the data it is initially given.
Knowledge Representation	Using representation languages with programs to store the knowledge learned by the system in a logical way.
Learning Operations	An agent is given a set of training instances and it is tasked to construct a generalization, heuristic rule or a plan that satisfies its goals.
Concept Space	The representation language along with the learning operations define a space of possible concept definitions, the learner needs to search this space to find the desired concept. The complexity of this concept space is used to measure how difficult the problem is.
Heuristic Search	Heuristics are used to commit to a particular direction when searching the concept space.

Table 3.4: Expert Systems

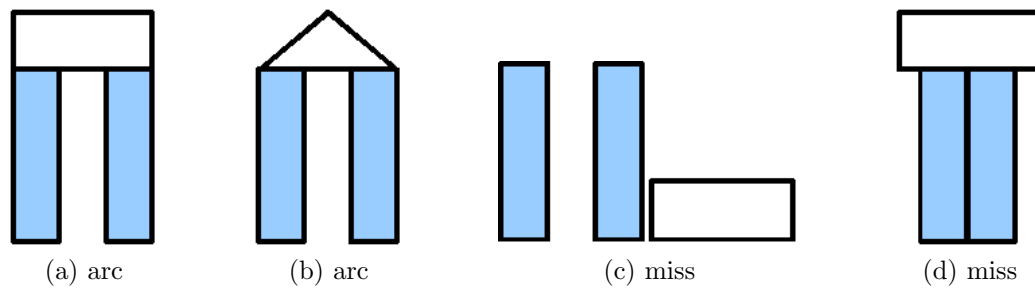


Figure 3.10: Examples of The Concept “arch” [68]p357

is not an arc because the third brick is not on the two support clocks. Figure 3.10d is also not an arc because the two support bricks are touching and therefore there is no opening under the third brick. From the above explanations we can conclude that an arc is obtained when the following two conditions are true:

1. Two bricks are supporting a polygon on them.
2. The two supporting brick are not touching each other.

Three powerful algorithms for searching the concept space are presented by Lugger [68]. They work by reducing the size of the concept space as more examples are made available. Figure 3.11 shows a concept space search for the concept “ball” using predicate calculus. The concept that the system is trying to learn is enclosed within the set S.

### 3.3.1.2 Connectionist Learning

Connectionist learning is performed using neural networks, which are systems comprised of a large number of interconnected artificial neurons. They have been widely used for [68]:

**Classification:** Deciding the category or grouping where an input value belongs.

**Pattern Recognition:** Identifying a structure in sometimes noisy data

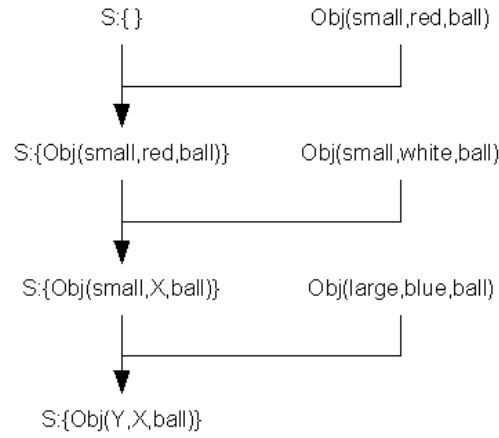


Figure 3.11: Learning the Concept “ball” [68]p364

**Memory Recall:** Addressing the content in memory

**Prediction:** Identifying an effect from different causes

**Optimization:** Finding the best organization within different constraints

**Noise Filtering:** Separating a signal from the background noise or removing irrelevant components to a signal

A neuron takes an input vector of weights and uses a function to determine its output value. The diagram of an artificial neuron can be seen in figure 3.12, it consists of [68]:

**Input Signals  $x_i$ :** That may come from the environment or the output of other neurons.

**Weights  $w_i$ :** That describe connection strengths between an input and the neuron.

**Activation Level  $\sum w_i x_i$ :** That is determined by the sum of all weights multiplied by their respective weights.

**Threshold Function  $f$ :** That computes the output by comparing the activation level with a specific threshold value.

The knowledge of the network is encapsulated within the organization and interaction of the neurons. Specifically, the global properties of neurons are characterized as [68]:

**Network Topology:** The topology of the network is the pattern of connections between neurons.

**Learning Algorithm:** The algorithm used to change the weight between different connections.

**Encoding Scheme:** The interpretation of input data presented to the network and output data obtained from the network.

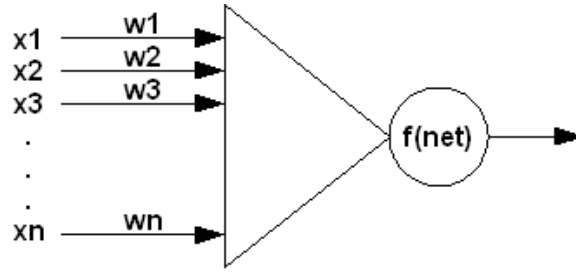


Figure 3.12: The Artificial Neuron [68]p420

Learning is achieved by modifying the structure of the neural network, via adjusting weights, in order to map input combinations to required outputs. There are two general classes of learning algorithms for training neural networks, they are *supervised* and *unsupervised* learning.

Supervised learning requires the neural network to have a set of training data, consisting of the set of data to be learned as well as the corresponding answer. The data set is repeatedly presented to the neural network. In turn, the network adapts by changing the weights of connections between the neurons until the network output corresponds closely to the required answers. The goal of supervised learning is to find a model or mapping that will correctly associate its inputs with its targets. Supervised learning is suited to applications when the outputs expected from the network are well known. This allows the designer (or another fully trained network) to provide feedback [68].

In the case of unsupervised learning the target value is not provided and the information in the training data set is continuously presented until some convergence criteria is satisfied. This involves monitoring the output of the network and stopping its training when some desired output is observed. The main difference to supervised learning is that the desired output is not known when the training starts. During training, the network has to continuously adapt and change its output until it demonstrates a *useful* output behaviour at which time it receives a single feedback to stop. The input data provided to the network will need to include sufficient information so that the problem is unambiguous. Unsupervised learning is suitable in situations where there is no clear-cut answer to a given problem [68].

The biggest problem of using neural networks with agents is that the concepts cannot intuitively fit within the agent oriented paradigm. However, neural networks have been used to implement part of a system such as pattern recognition and classification. It is also believed that neural learning concepts and techniques will play an important role in future research.

### 3.3.1.3 Social and Emergent Learning

This learning method focuses on learning algorithms using the underlying concept of evolution, in other words, shaping a population  $P(t)$  of candidate solutions  $x_i^t$  through the survival of the fittest members at time  $t$ .  $P(t)$  is defined as:

$$P(t) = \{x_1^t, x_2^t, \dots, x_n^t\}$$

The attributes of a solution are represented with a particular pattern that is initialized by a genetic algorithm. As time passes, solution candidates are evaluated according to a specific

fitness function that returns a measure of the candidate's fitness at that time. After evaluating all candidates the algorithm selects pairs for recombination. Genetic operators from each individual are used to produce new solutions that combine components of their parents. The fitness of a candidate determines the extent to which it reproduces, the general form of the genetic algorithm is shown in figure 3.13.

```

1:  $t \leftarrow 0$ 
2: Initialize population  $P(t)$ 
3: while termination condition not met do
4:   for Each member  $x_i^t$  within  $P(t)$  do
5:      $fitness(member) \leftarrow FitnessFunction(member)$ 
6:   end for
7:   select members from  $P(t)$  based on  $fitness(member)$ 
8:   produce offspring of selected members using generic operators
9:   replace members of  $P(t)$  with offspring based on fitness
10:   $t \leftarrow t + 1$ 
11: end while

```

Figure 3.13: General Form of The Genetic Algorithm [68]p471

### 3.3.2 Reinforcement Learning

Reinforcement learning (RL) is designed to allow computers to learn by trial and error. It is an approach to machine intelligence that combines two disciplines to solve a problem that each discipline cannot solve on its own. The first discipline, dynamic programming is a field in mathematics used to solve problems of optimization and control. The second discipline, supervised learning was discussed in section 3.3.1.2. In most real-life problems the correct answers required with supervised learning are not available, using RL the agent is simply provided with a reward-signal that implicitly trains the agent as required, figure 3.14 illustrates the agent-environment interaction used with RL. The agent and the environment interact in a discrete sequence of time steps  $t = 0, 1, 2, 3, \dots$ , for each time step the agent is presented with the current instance of the state  $s_t \in S$  where  $S$  is the set of all possible states. The agent then uses the state to select and execute an action  $a_t \in A(s_t)$  where  $A(s_t)$  is the set of all possible actions available in state  $s_t$ . In the next time step the agent receives a reward  $r_{t+1} \in R$ , and is presented with a new state  $s_{t+1}$ . The system learns by mapping an action to each state for a particular environment. A specific mapping of actions and states is known as a *policy*  $\pi$  where  $\pi_t(s, a)$  is the probability that  $a_t = a$  if  $s_t = s$ . Actions available to agents can be separated into three different categories [49]:

- Low level actions (i.e. supplying voltage to a motor)
- High level actions (i.e. making a decision)
- Mental (i.e. shifting attention focus)

An important point to note is that according to figure 3.14 the reward is calculated by the environment which is external to the agent. This is a confusing concept because at first

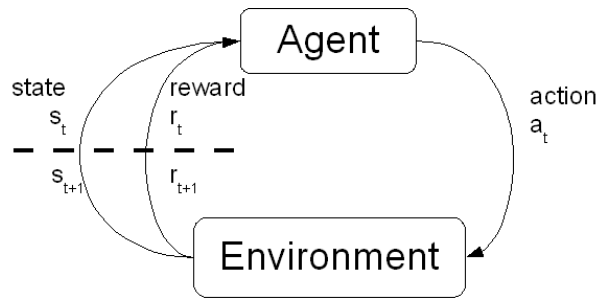


Figure 3.14: The RL Agent-Environment Interface [104]p52

it seems that the designer of an RL system is required to somehow implement something in the environment in order to provide an agent with appropriate rewards. The RL literature overcome this problem by explaining that the *boundary* between the agent and the environment need not be distinctively physical. The boundary of the agent is shortened to include only the reasoning process, everything outside the reasoning process which includes all other components of the agent, are treated as part of the environment. In the context of human reasoning, this is analogous to treating the human brain as the agent and the entire human body as part of the environment.

Note that for the purpose of this thesis, this is still not sufficient because when an agent is implemented there is definitely a clear interface between the software components of the agent and the environment. It becomes confusing when trying to consider only the reasoning process as the agent, while at the same time implementing a number of software components that are all internal to the the agent. In addition, figure 3.14 does not make it clear how to use multiple reward functions in order to have an agent learn multiple things concurrently, any moderately complex system like the ones described in chapter 2 would require this feature. Consequently, the diagram illustrated in figure 3.14 is enhanced to overcome these limitations as shown in figure 3.15 which provides a sneak preview of the contributions of this thesis. This enhanced interface considers the boundary of the agent as the physical boundary, rewards are generated through a number of internal learning goals that are explicitly defined by the designer. A separate RL process is executed for each learning goal, they all collectively provide *suggestions for actions* to the main reasoning process of the agent which may or may not decide to perform them based on other factors such as beliefs, desires and intentions. These new features presented in this paragraph are described in detail in chapter 5, however, the remainder of this section focuses on traditional RL theory.

### 3.3.2.1 Markov Property

The *Markov property* is concerned with the way that the state signal received from the environment is represented. This is an important issue when developing an RL system because all actions are directly dependent on the state of the environment. In a causal system the response of the environment for an action taken at time  $t$  will depend on all actions previously taken as shown in equation 3.1. Therefore, to make good decisions, a state signal must include



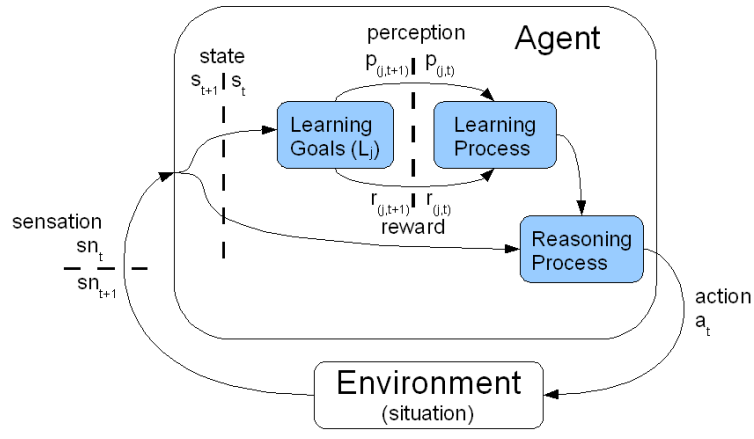


Figure 3.15: An Enhanced Agent-Environment Interface

immediate sensations, as well as structures containing information from all previous sensations.

$$PR\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \dots, s_0, a_0\} \quad (3.1)$$

The state signal however, should not be expected to represent everything about the environment because certain information might be inaccessible or intentionally made unavailable. For example, if an agent is controlling a player in Unreal Tournament, it should not be required to know exactly where every other player is located within the game, or even where itself was located 5 minutes before. It would simply need to know where it is currently located as well as the locations of enemy players within close range.

When the response of the environment depends only on the state and action representations at time  $t$ , is it said to have the Markov property and can be defined as shown in equation 3.2. This means that the state signal is able to summarize all past sensations compactly such that all relevant information is retained for making decisions.

$$PR\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\} \quad (3.2)$$

When a reinforcement learning problem satisfies the Markov property it is called a *Markov Decision Process* (MDP), additionally if the states and actions sets are finite then it is called a *finite MDP*. In some cases even when a particular problem is non-Markov it may be possible to consider it as an approximation of an MDP for the basis for learning, in such cases the learning performance will depend on how good the approximation is.

### 3.3.2.2 Reward Function

The reward function  $R_{ss'}^a$  provides rewards depending on the actions of the agent. The sequence of rewards received after time step  $t$  is  $r_{t+1}, r_{t+2}, r_{t+3}, \dots$ , the agent learns by trying to *maximize* the sum of rewards received when starting from an initial state and proceeding to a terminal state. An additional concept is the one when an agent tries to maximize the expected *discounted return* as shown in equation 3.3, this involves the agent discounting future rewards by a factor

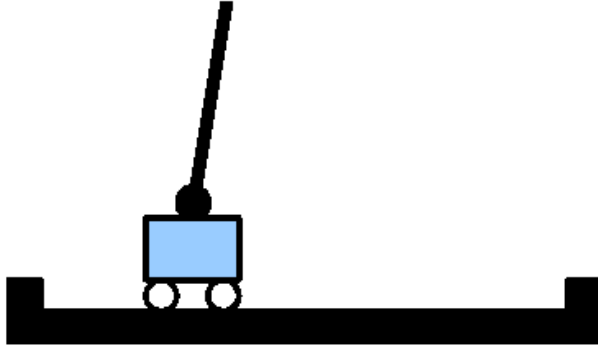


Figure 3.16: The Cart-Pole Reinforcement Problem [49]p3

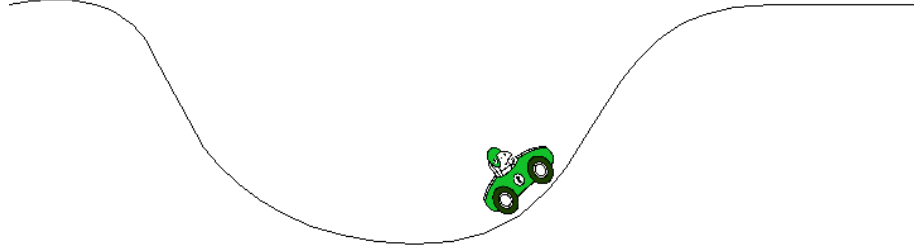


Figure 3.17: The Car-On-Hill Reinforcement Problem [49]p3

of  $\gamma$  where  $0 \leq \gamma \leq 1$ .

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad (3.3)$$

There are two important classes of reward functions [49]. In *pure delayed* reward functions, rewards are all zero except at a terminal state where the sign of the reward indicates whether it is a goal or penalty state. A classic example of pure delayed rewards is the cart-pole problem as illustrated in figure 3.16. The cart is supporting a hinged inverted pendulum and the goal of the RL agent is to learn to balance the pendulum in an upright position. The agent has two actions in every state, move left and move right. The reinforcement function is zero everywhere except when the pole falls or the cart hits the end of the track, when the agent receives a  $-1$  reward. Through such a set-up an agent will eventually learn to balance the pole and avoid the negative reinforcement [49].

Using *minimum time* reward functions it becomes possible to find the shortest path to a goal state. The reward function returns a reward of  $-1$  for all actions except for the one leading to a terminal state for which the value is again dependent on whether it is a goal or penalty state. Due to the fact that the agent wants to maximize its rewards, it tries to achieve its goal at the minimum number of actions and therefore learns the optimal policy. An example used to illustrate this problem is driving a car up the hill problem as shown in figure 3.17. The problem is caused by the car not having enough thrust to drive up the hill on its own and therefore the RL agent needs to learn to use the momentum of the car climb the hill [49].

### 3.3.2.3 Value Function

The issue of how an agent knows what is a *good* action is tackled using the *value function*  $V^\pi(s)$  which provides a value of “goodness” to states with respect to a specific policy. For MDPs, the information in a value function can be formally defined as shown in equation 3.4 where  $E_\pi\{\}$  denotes the expected value if the agent follows policy  $\pi$ , this is called the *state value function*. Similarly, the value function, if starting from  $s$ , taking action  $a$ , and thereafter following policy  $\pi$  is defined as shown in equation 3.5, this is called the *action value function*.

$$V^\pi(s) = E_\pi \{R_t | s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\} \quad (3.4)$$

$$Q^\pi(s, a) = E_\pi \{R_t | s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\} \quad (3.5)$$

A value function that returns the highest value for the best action in each state is known as the *optimal value function*.  $V^*(s)$  and  $Q^*(s, a)$  denote the optimal state and action value functions and are shown in equations 3.6 and 3.7 respectively. These are otherwise called the *Bellman optimality equations*, they express that the value of a state under an optimal policy is equal to the expected return for the best action from that state.

$$V^*(s) = \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^*(s')] \quad (3.6)$$

$$Q^*(s, a) = \sum_{s'} P_{ss'}^a \left[ R_{ss'}^a + \gamma \max_{a'} Q^*(s', a') \right] \quad (3.7)$$

### 3.3.2.4 Learning Algorithms

Learning algorithms are concerned with *how* and *when* to update the value function using provided rewards. The differences in algorithms range depending on the required data that they need to operate, how they perform calculations and finally when this update takes place. Learning algorithms can be divided into three major classes: Dynamic Programming, Monte Carlo and Time-Difference. Each is briefly described in this section with particular emphasis given to differences between algorithms.

Dynamic programming (DP) works by assigning blame to the many decisions a system has to do while operating, this is done using two simple principles. Firstly, if an action causes something bad to happen immediately, then it learns not to do that action from that state again. Secondly, if all actions from a certain state lead to a bad result then that state should also be avoided. DP requires a *perfect model* of the environment in order to find a solution. Therefore, the environment must have finite sets of states  $S$  and actions  $A(s)$ , and also finite sets of transition probabilities  $P_{ss'}^a = Pr \{s_{t+1} = s' | s_t = s, a_t = a\}$  and immediate rewards  $R_{ss'}^a = E \{r_{t+1} | s_{t+1} = s', s_t = s, a_t = a\}$  for all  $s \in S, a \in A(s)$ . The value function in DP is updated using equation 3.8.

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')] \quad (3.8)$$

```

1:  $\pi \leftarrow$  policy to be evaluated
2:  $V \leftarrow$  an arbitrary state-value function
3:  $Returns(s) \leftarrow$  an empty list, for all  $s \in S$ 
4: while true do
5:   Generate an episode using  $\pi$ 
6:   for Each state  $s$  appearing in the episode do
7:      $R \leftarrow$  return following the first occurrence of  $s$ 
8:     Append  $R$  to  $Returns(s)$ 
9:      $V(s) \leftarrow average(Returns(s))$ 
10:   end for
11: end while

```

Figure 3.18: First Visit Monte Carlo Algorithm [104]p113

Starting from the far right in the equation it can be seen that the reward received for taking an action is added to the discounted value of the resulting state of that action. However, a single action may have multiple effects in a complex environment leading to multiple resulting states. The value of each possible resulting state is multiplied by the corresponding transition probability and all results are added to obtain the actual value of a single action. In order to calculate the value of the state itself, the value of each action is calculated and added to produce the full value of the state.

The two biggest problems encountered when developing applications using DP are:

- The requirement of previously knowing all effects of actions taken in the environment.
- The exponential increase in computation required to calculate the value of a state for only a small increase in possible actions and/or effects.

Monte Carlo (MC) methods do not assume complete knowledge of the environment and require only experience through sampling sequences of states, actions and rewards from direct interaction with an environment. They are able to learn by segmenting sequences of actions into episodes and averaging rewards received as shown by the algorithm illustrated in figure 3.18, notice that the algorithm requires the generation of an entire episode (line 5) before performing any updates to the value function.

MC is also able to estimate action values rather than state values, in this case policy evaluation is performed by estimating  $Q^\pi(s, a)$  which is the expected return when starting in state  $s$ , taking action  $a$ , and thereafter following policy  $\pi$ . The relevant algorithm has the same structure as figure 3.18. When MC is used for approximating optimal policies, Generalized Policy Iteration (GPI) is used. GPI maintains an approximate policy and an approximate value function, it then performs *policy evaluation*<sup>1</sup> and *policy improvement*<sup>2</sup> repeatedly. This means that the value function is updated to reflect the current policy while the policy is then improved with respect to the value function. Using these two processes GPI is able to maximize its rewards.

Temporal-Difference (TD) learning combines ideas from both MC and DP methods. Similarly to MC, TD methods are able to learn from experiences and do not need a model of the

<sup>1</sup>Policy evaluation calculates the value function of a given policy

<sup>2</sup>Policy improvement changes the policy such that it takes the best actions as dictated by the value function

```

1: Initialize  $Q(s, a)$  arbitrarily
2: for each episode do
3:   initialize  $s$ 
4:   Choose  $a$  from  $s$  using policy derived from  $Q$ 
5:   for Each state  $s$  in episode do
6:     Take action  $a$ , observe  $r, s'$ 
7:     Choose  $a'$  from  $s'$  using policy derived from  $Q$ 
8:      $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$ 
9:      $s \leftarrow s'; a \leftarrow a';$ 
10:  end for
11: end for

```

Figure 3.19: Sarsa RL Algorithm [104]p146

environment's dynamics. Like DP, TD methods update the value function based in part on estimates of future states (this feature is called *bootstrapping*) and hence do not require waiting for the episode to finish. *Sarsa* is an example TD learning, the general algorithm is illustrated in figure 3.19, the most important part of the algorithm is line 8 where the action value function is updated as per equation 3.9.  $\alpha$  is called the *step-size parameter* and it controls how much the value function is changed with each update. Sarsa is an *on-policy* TD algorithm and it requires the agent to select the following action before updating  $Q(s, a)$ , this is because  $Q(s, a)$  is calculated by subtracting  $Q(s, a)$  from the discounted value of  $Q(s', a')$  which can only be known by selecting  $a'$ . Notice that actions are selected using a policy that is based on the value function and in turn the value function is updated from the reward received.

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)] \quad (3.9)$$

*Off-Policy* TD is able to approximate the optimal value function independently of the policy being followed. *QLearning* is an example an off-policy TD algorithm, it is listed in figure 3.20. The main difference between Sarsa and QLearning lies in the calculation that updates the value function, the QLearning update function is shown in equation 3.10. With Sarsa the value function is updated based on the *next chosen* action, conversely with QLearning it is updated based on the *best known* future action even if that action is actually *not selected* in the next iteration of the algorithm.

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \quad (3.10)$$

### 3.3.2.5 Exploration versus Exploitation

One of the more well known problems within the RL literature is the exploration/exploitation problem. During operation the agent forms action estimates  $Q^\pi(a) \approx Q^*(a)$ . The best known action at time  $t$  would therefore be  $a_t^* = \arg \max_a Q_t(a)$ . An agent is said to be *exploring* when it tries an new action for a particular situation  $a \neq a_t^*$ . The reward obtained from the execution of that action is used to update the value function accordingly. An agent is said to be *exploiting* its learning knowledge when it chooses the *greedy* (best) action indicated by

```

1: Initialize  $Q(s, a)$  arbitrarily
2: for each episode do
3:   initialize  $s$ 
4:   for Each state  $s$  in episode do
5:     Choose  $a'$  from  $s'$  using policy derived from  $Q$ 
6:     Take action  $a$ , observe  $r, s'$ 
7:      $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
8:      $s \leftarrow s'$ 
9:   end for
10: end for

```

Figure 3.20: QLearning RL Algorithm [104]p149

its value function in a particular state  $a = a_t^*$ . In this case, the agent also updates the value function according to the reward received. This may have two effects, firstly, the reward may be similar to the one expected by the value function, which means that the value function is converging on the problem trying to be solved. Secondly, it may be totally different to the value expected, therefore changing the value function and possibly the ordering of the actions with respect to their values. Hence, another action may subsequently become the ‘best’ action for that state.

An action selection policy controls the exploitation/exploration that is performed by the agent while learning. There are two types of policies considered in this thesis. Firstly, the *EGreedy* policy explores by selecting actions randomly but only for a defined percentage  $\epsilon$  of all actions chosen as shown in equation 3.11. For example, if  $\epsilon = 0.1$  then the agent will explore only 10% of the time, the rest of the time it chooses the greedy action.

$$a_t = \begin{cases} a_t^* & \text{probability} = (1 - \epsilon) \\ \text{random} & \text{probability} = \epsilon \end{cases} \quad (3.11)$$

Secondly, the *SoftMax* action selection is more complex. It makes its choice based on equation 3.12 where  $\tau$  is called the *temperature* value. A high temperature selects all actions randomly, while a low temperature selects actions in a greedy fashion. An intermediate temperature value causes SoftMax to select actions with a probability that is based on their value. This way actions with a high value have a greater chance of being selected while actions with a lower value have less chance of being selected. The advantage of SoftMax is that it tends to select the best action most of the time followed by the second-best, the third-best and so on, an action with a very low value is seldom executed. This is useful when a particular action is known to cause extremely bad rewards. Using SoftMax, that action will always get a very small probability of execution. With *EGreedy* however, it has the same probability as any other action when exploring.

$$a_t = \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^n e^{Q_t(b)/\tau}} \quad (3.12)$$

### 3.3.3 Agents and Learning

This section briefly describes some of the work that has been done to integrate learning capabilities into agent based systems.

#### 3.3.3.1 Plan Recognition

The Consolidated Algorithm for Relational Evidence Theory (CLARET) [85] is a learning module for the development of agents capable of plan recognition. This has been applied in the recognition of airplane manoeuvres in order to predict the intentions of pilots. Some of the difficulties inherent in plan recognition are due to the possible combination of behaviours resulting of:

- Team tactics where the behaviour is influenced by the role played in a team.
- Execution of multiple concurrent plans that could produce a mix of actions from different plans.
- Ambiguous action caused by agents not having a correct situation awareness.
- Plan switching caused by the agent re-evaluating its goals and intentions.
- Deliberate deception caused by the enemy trying to have a degree of unpredictability.

CLARET is able to learn to detect which plans an agent is executing based on the actions that it takes. The advantage of this ability is that after an agent has detected which plans an adversarial agent is using, it becomes possible to predict the actions of the adversary. The agent therefore has time to develop a strategy on how to defend against an attack while also achieving an effective offense [51].

#### 3.3.3.2 Profit Sharing

Profit Sharing is a reinforcement learning approach that has been used realize adaptive behaviours in a multi-agent environment. It is used to overcome three problems that have been encountered when reinforcement learning approaches are applied to multi-agent domains. The first problem arises from the agent's sensory limitation in which the agent is fooled into perceiving two or more different states as the same state. This is known as *perceptual aliasing*, if the different states require the same action, then perceptual aliasing is a desirable outcome and is commonly called *generalization*. However, if a different action is required, then the agent may become confused and hence, perform irrational actions. The second problem is due to the dynamics of the environment varying unpredictably and each agent modifying its behaviours asynchronously. This causes the agent to not be able learn effectively because it never knew what the available actions were. The third problem involves the amount of memory required to make an agent behave effectively, which in complex environments may be too great to be useful. Profit-sharing avoids these problems using trial and error experiences by being an action-preference learning method, that maintains preferences for each action instead of action-value estimates. The weight of each rule is reinforced according to its distance from a given goal [8]. The general structure of an agent that learns using profit sharing is illustrated in figure 3.21.

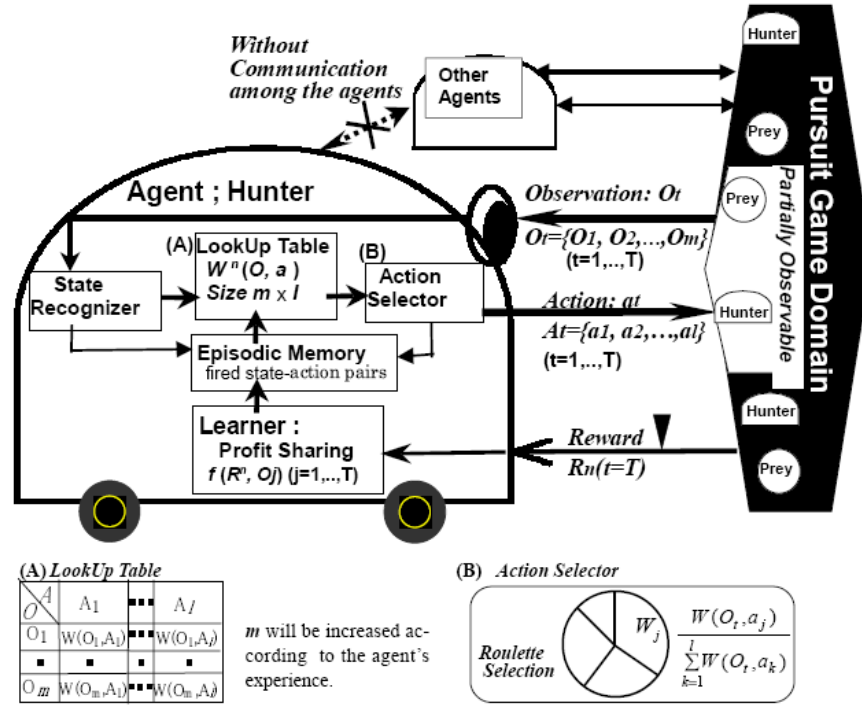


Figure 3.21: Model of a Profit Sharing Agent [8]p508

### 3.3.3.3 Collaborative Learning

An overly-simple definition of collaborative learning is when two or more people learn or attempt to learn something together. Learning can take the form of following a certain course of action, studying some materials, performing activities like problem solving and learning from lifelong practice [31]. There are several issues involved when analyzing the properties of collaboration in multi-agent systems. Some interesting concepts are summarized in this section.

There are three different mechanisms that multi-agent systems can use to handle learning a new task. Firstly, the multiplication mechanism involves each agent learning independently from the others. An important property that emerges from multiplication learning is the notion of conflict. This happens when two agents disagree at some point and are pressured by the system to solve the conflict. The resolution of the conflict may lead to an agent changing its viewpoint and learning occurs from the explanations and justifications that have lead to the acceptance of both agents. The division mechanism involves a single learning task being divided among several agents. The division may be dependent on the problem algorithm or the data that is processed to solve the problem. Thirdly, the interaction mechanism involves agents constantly interacting during learning. This means that interaction encompasses all steps of the learning process. This is different from the first two mechanisms where communication involves only data exchange between agents [112].

Grounding is related to collaborative learning because it is the name given to the process where a common ground is constructed and maintained between agents. Common ground includes mutual understanding, beliefs, knowledge and intentions. Depending on the situation, certain assumptions are made when grounding, such as co-membership, attention and ratio-



nality. Additionally, some sort of feedback is required from each member during grounding in order to show that the learner is listening and comprehending. There are several different problems related to maintaining common ground, for example: the issue of the learner being willing to continue learning, the issue of whether a learner is willing to perceive new information, the issue whether a learner is willing to understand the message, and finally, whether the learner is willing and able to react and adequately respond to messages.

Baker, Hansen, Joiner and Traum [9] have defined a framework for investigating the role of grounding in collaborative learning. According to this framework collaborative learning is the pursuit of certain goals by a group of agents. Learning is achieved when the entities and their relationships change through time. Each agent is able to communicate with the other and build a representation of the mental state of the other. Agents are also able to be held responsible towards another agent for specific actions. Goals are external or internal to the agents to account for the fact that the goals of the whole system may not be the same as those perceived by the learner agents.

### 3.4 Summary

The reasoning performed by agents can be tackled from different angles corresponding to thinking versus acting. Research on rational reasoning is concerned with agents acting as best they can given their limited knowledge. Instead, research on human reasoning is concerned with the development of models that describe how humans make decisions.

Knowledge representation involves the development of formal representation languages with the ability to maintain consistent knowledge about the world and also different processes that bring the knowledge to life. Expert systems are examples of such systems that try to replicate what a human expert would do by accumulating knowledge extracted from different sources and using different operations on the knowledge in order to solve the given problem. Production systems extend expert systems using pre-defined rules and a working memory for making decisions. Rules are consisted of a condition that determines when it is applicable to be executed and an action which defines what to do when executed.

Practical reasoning is concerned with studying the way that humans reason about what to do in everyday activities via their Beliefs, Desires and Intentions. It is specifically geared to reasoning towards actions by weighing conflicting considerations of different options depending on what a person desires to do. It can be divided into two distinct activities, the first activity is called deliberation and involves deciding on what has to be achieved, the second activity is called means-ends reasoning which involves deciding on how to achieve the required result.

Cognitive Systems Engineering takes into account that systems will be used by humans. It acknowledges that humans are dynamic entities that are part of the system and cannot be modeled as static components of the system. The decision ladder developed in this area models the human decision making process through a set of operations and states of knowledge. The ladder defines three key levels of expertise: skill, rule and knowledge, and learning involves the crossing of skill level boundaries.

Another model developed as part of military tactical assessment is called the OODA loop that reduces the decision cycle into four steps: Observation performs gathering data from the

surrounding environment, Orientation creates a mental model of the situation, Decision makes a decision, and Action implements the decision. The OODA loop also identifies a number of important feedbacks in the decision process that illustrates how to operate at a faster reasoning tempo. This allows the agent to operate inside the mind-time-space of an adversary which provides it with an advantage.

The concept of learning has many definitions depending on the area that it is being applied to. Learning can be thought to have key features such as obtaining new skills, altering behaviours and increasing efficiency. Symbol-based learning relies on learning algorithms that operate on a knowledge base in order to infer new knowledge. Neural networks are comprised of a large number of interconnected artificial neurons, that learn by modifying the structure of the network in order to adapt to inputs and outputs given. Genetic algorithms focus on the concept of evolution via shaping a population of agents. Reinforcement learning allows computers to learn by trial and error. It uses a value function for actions taken, that is updated according to rewards generated by a reward function. The agent learns by trying to maximize the rewards that are received during operation. Learning algorithms are used to define how and when to update the value function using the rewards.

Previous research on agents and learning has been applied to plan recognition where an agent tries to figure out which plan another agent is executing. This allows it to predict the actions of the other agent and therefore develop a strategy on how to proceed. Profit sharing is also a reinforcement learning approach that uses action-preferences instead of action-value estimates as per traditional algorithms. Finally, collaborative learning is concerned with issues encountered when two or more agents learn together.

A good understanding of how agents are able to perform reasoning has now been achieved with some potential techniques that can be used in order to realize learning. The following chapter will introduce how agents have been implemented. A detailed description of the agent development environment used for this thesis is also described.

# Chapter 4

## Intelligent Agents

This chapter examines the latest research underpinning agent technology. It gives an insight into the use and types of intelligent agents with a focus based on their internal components. Furthermore, it introduces different agent development platforms grouped as per their implementation techniques with a detailed description of the JACK agent development platform used for this research.

### 4.1 Introduction

Agents embody a new software development paradigm that attempts to merge some of the theories developed in artificial intelligence research with computer science. The power of agents comes from their intelligence and also their ability to communicate with each other. A simple mapping of agent technology compared to relevant technologies is illustrated in figure 4.1.

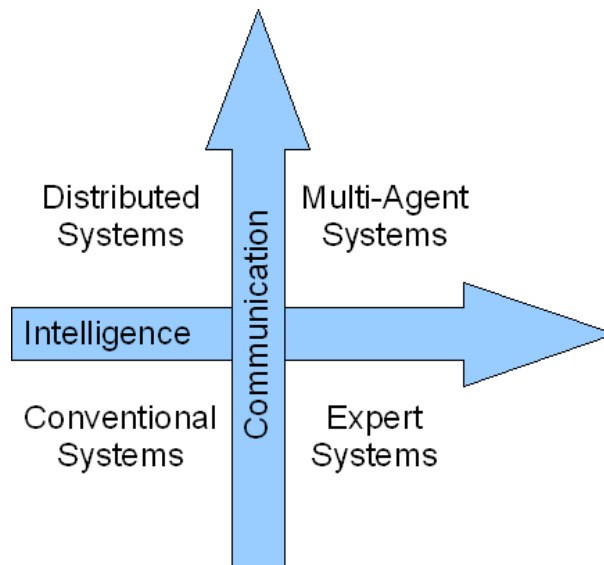


Figure 4.1: Agent Technology [101]

Agents can be considered as the successors of objects<sup>1</sup> when applied to certain problem

---

<sup>1</sup>Objects are software constructs widely used in software to reduce complexity while increasing robustness. Systems using objects are said to be developed using object-oriented techniques. Objects are able to encapsulate a set of variables and define methods for message passing between objects, they can also inherit variables and methods from parent objects and optionally override them for customization.

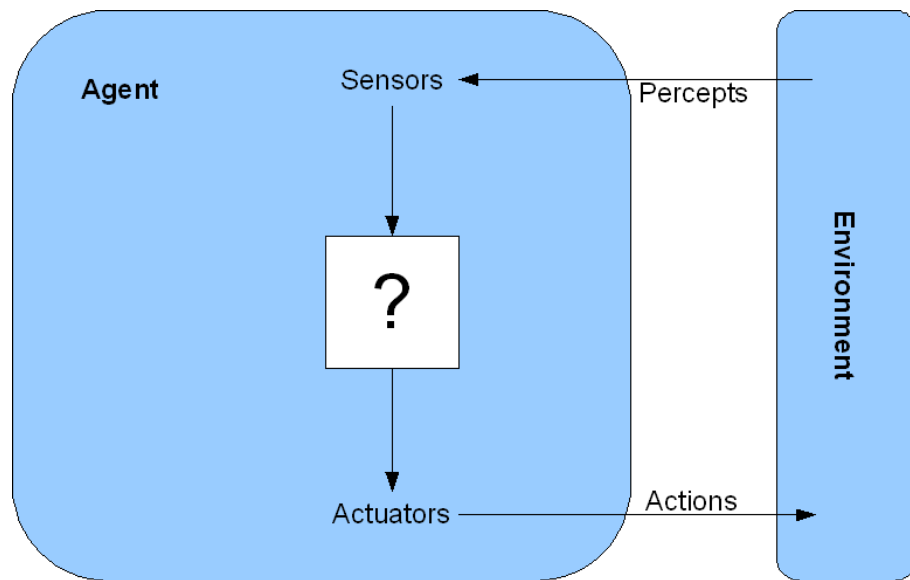


Figure 4.2: What is an Agent? [95]p33

domains. However, the additional layer of implementation in agents provides some key functionalities and deliberately creates a separation between the implementation of an agent from the application being developed. This is done in order to achieve one of the core properties of agents, autonomy. Objects are able to assert a certain amount of control over themselves via private variables and methods, and other objects via public variables and methods. Consequently, a particular object is able to directly change public variables of other objects and also execute public methods of other objects. Hence, objects have no control over the values of public variables and who and when executes their public methods. Conversely, agents are explicitly separated, and can only *request* from each other to perform a particular task. Furthermore, it cannot be assumed that after a particular agent makes a request, another agent will do it. This is because performing a particular action may not be in the best interests of the other agent, in which case it would not comply [116].

One of the reasons that agent-based technology is interesting to research is because currently there is no universal definition of an agent. This is partly because the term *agent* is widely used by many people working in closely related areas trying to perform more complex tasks with less human guidance [118]. The general problem of developing an agent is illustrated in figure 4.2, which states the agent is situated within an environment that generates sensations for the agent. The agent then performs *reasoning* using sensations about what to do next. It finally makes a decision and subsequently takes the relevant actions.

Wooldridge has provided a general definition of an agent, this is because of the danger that if no definition is given the term will be used for many different purposes and finally lose all meaning. The definition is given below.

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 requirements [116].

When defining agents, researchers mention the properties that a system should exhibit. Firstly, *autonomy* involves operating without the direct intervention of humans. Secondly,

*social ability* includes interacting with other agents. Thirdly, *reactivity* covers perceiving their environment and responding to any changes that occur in it. Finally, *pro-activeness* is concerned with exhibiting goal-directed behavior by initiating actions accordingly. Pro-activeness can be related to traditional software systems because they are implicitly built in this way. The effects of a procedure implicitly define the goal. If the preconditions are true when the procedure is started, it is expected to work correctly and terminate when the desired effect is reached.

There are also two important assumptions that must be avoided when defining agents. Firstly, it must not be assumed that the environment does not change while an agent is in the middle of reasoning, this is because if the environment does change, the agent may not be able to continue functioning. Secondly, it must not be assumed the goal of an agent will always remain valid until achieved. It may just happen that a particular goal is no longer possible to achieve. If this is the case, the agent will perform unreasonable actions unless it re-evaluates its goals. Agents are therefore required to be both reactive and deliberative. They must be able to strive to achieve goals while responding to events that affect the agent's goals and assumptions about the environment [118].

## 4.2 Using Intelligent Agents

In order to describe how to implement the reasoning of an agent, it must first be understood how to use agents in the real world. Generally, any system that can exhibit some control can be viewed as an agent. A trivial example of this analogy is represented by a light switch. A switch is integrated into an environment and can produce two different outputs **on** and **off**. The output is determined by the physical position of the switch. With this analogy in mind, it can be concluded that the agent can perform two actions which are to **turn on** and **turn off** the light.

When viewed as software entities, agents inhabit a software environment, they obtain information from the environment by invoking software functions and can perform software actions such as sending messages on a computer network [116]. There are however, a number of important pitfalls with regards to using agents in real world systems. They can be broken up into six different categories: [116].

1. Political pitfalls refer to people overselling agents for use in real world applications due to their emphasis on automation. Unfamiliar people therefore mistakenly believe that agents are currently capable of advanced reasoning and acting even though the technology has not reached that level yet. Consequently, agents are being used in applications where conventional software development approaches are more appropriate [116].
2. Management pitfalls refer to agent projects starting without having a clear idea on why the agents are needed or how they can be used to enhance or generate new products. This results in having no real goals in mind and therefore no criteria for assessing the performance of the project. This is a problem because the full potential of what the agents can do is not achieved. Other types of management pitfalls include devising new

architectures for developing agents while trying to solve a specific application. Custom built solutions will be easier to develop and more likely to satisfy the requirements [116].

3. Conceptual pitfalls refer to believing that agents are the *silver bullet* in development even though there is no scientific evidence yet to support a claim that the agents offer any advance in software development. In general, agents are simply powerful programming abstractions and development of any agent system involves a certain amount of experimentation. The adverse effect of this is that it encourages developers to forget that they are developing software and place less emphasis on software engineering processes such as requirements analysis, design, verification and testing. In reality however, agent-based systems are distributed systems which have been recognized as one of the most complex classes of computer software to design and implement. Some relevant distributed system problems include synchronization, mutual exclusion for shared resources and deadlocks [116].
4. Analysis and Design pitfalls refer to not exploiting related mature technologies such as CORBA [45] for distributed implementation, database software to handle large information processing and expert systems to handle complex reasoning tasks. Additionally, some poor multi-agent designs include little support for concurrency and in some extreme cases it is non-existent. Multi-Agent systems should be able to handle multiple objectives, while reacting to the environment concurrently [116].
5. Agent-Level pitfalls refer to the development of new architectures. New architectures are often created when starting an agent-based project and there is no existing architecture that exactly meets the project's requirements. After creating a new architecture, it is important to understand that the architecture was created to satisfy the problem at hand and hence should only be applied to similar problems. In some cases, when building applications there is a tendency to focus on the agent-specific aspects of the application. The system is then developed with experimental AI techniques making it unusable. A better approach is to use a minimum of stable AI techniques that are slowly evolved with future development as required. On the other hand, some straight-forward distributed applications that exhibit no artificial intelligence properties, are called agent systems with no justification [116].
6. Society level pitfalls are concerned with multi-agent technologies. In some cases people view everything as an agent even though a simple object would be more appropriate. A number of systems also follow the approach that developing a single complex agent is harder than developing many simpler agents to do the same job. In some cases this is true, however if there are too many agents, it becomes difficult to understand the behavior since the complexity increases exponentially as more agents are introduced into the mix. Conversely, is it possible to ignore the power of the multi-agent approach completely, developing huge monolithic agents that are able to perform a whole range of tasks. It is also important to understand the difference between real and simulated parallelism, this is to avoid problems encountered when a system is being developed using simulated concurrency in a development machine. It may not be able to scale well up

to a real distribution where agents are residing on different hardware. Finally, it is also important to consider emerging standards when working with agent systems, this is to ensure that even if agents are required to be implemented from scratch, they are able to communicate with agents from other systems, an example of such a standard is the KQML [72] language for agent communication [116].

## 4.3 Types of Intelligent Agents

This section provides a general overview of different types of agents and groups them into several intuitive categories based on the method that they perform their reasoning. A brief description of relevant software implementations for the different types of agents is also given.

### 4.3.1 Deliberate Agents

Deliberate agents are agents that perform rational reasoning, they take actions that are rational after deliberating using their KB, carefully considering the possible effects of different actions available to them. There are two subtypes of deliberate agents. Deductive reasoning agents perform all reasoning using an internal KB that attempts to provide a limited model of the environment. Blackboard systems are an extension to this approach whereby the KB is called a blackboard and is shared among a number of concurrent processes.

#### 4.3.1.1 Deductive Reasoning Agents

Deductive reasoning agents are built using expert systems theory (see section 3.2.1.2), they operate using an internal symbolic KB of the environment. Desired behavior is achieved by manipulating the environment and updating the KB accordingly. A utility function is implemented that provides an indication on how good a particular state is compared on what the agent should achieve. An example of the idea behind these type of agents is an agent that explores a building. It has the ability to move around and it uses a video camera, the video signal is processed and translated to some symbolic representation. As the agent explores the world it maintains a data structure of what it has explored.

The internal structure of deductive reasoning agents is illustrated in figure 4.3. There are two key problems encountered when trying to build deductive reasoning agents. Firstly, the transduction problem is the problem of translating the real world into an accurate, symbolic description in time for it to be useful. Secondly, the representation or reasoning problem is the problem of representing acquired information symbolically and getting agents to manipulate/reason with it [116].

CLIPS[43] is an example of a development environment for the construction of rule and/or object based expert systems. It supports a variety of KB formats using rule-based, object-oriented or procedural paradigms. Rule-based knowledge is represented as heuristics that specify a set of actions to be performed for a given situation. CLIPS can be ported to any system that has an ANSI compliant C or C++ compiler and the source code is released as public domain software, therefore it can be modified or tailored to meet specific requirements

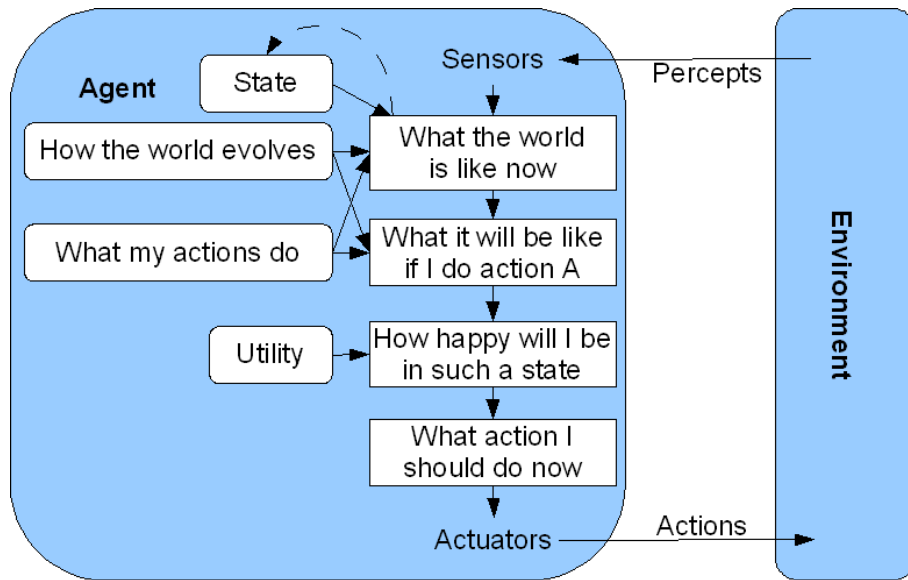


Figure 4.3: Deductive Reasoning Agents [95]p52

[43]. Jess is also an expert system development environment based on CLIPS but it is written entirely the Java language [61].

#### 4.3.1.2 Shared Data Agents

Blackboard systems operate on a problem solving model that organizes steps and domain knowledge to construct a solution to the problem. They were designed in order to eliminate some of the inherent problems encountered with traditional expert system techniques. They work by segmenting the knowledge base into modules and providing a separate inference engine for each module. The communication between modules is limited to reading and writing the shared knowledge base.

The shared knowledge base takes the form of a blackboard where a number of modules are concurrently making changes while working towards a goal. Therefore, blackboard systems contain two major components. The first component is knowledge sources encapsulated within separate and independent modules, each of which brings a particular advantage to the system. The second component is the blackboard data structure which is the global database that each of the knowledge sources incrementally operate on, in order to lead to a solution.

A final important issue to mention regarding blackboard systems is the issue of control. A control process needs to monitor the changes on the blackboard and decide on what should happen next, this includes deciding on which part of the blackboard should be operated on next, and also which of the modules should be able to access the blackboard [34].

Cougaar[109] is an example of an agent development environment where the agents use a blackboard architecture. It provides a workflow engine built on a component-based, distributed agent architecture where agents communicate via an asynchronous message-passing protocol and cooperate in order to solve a particular problem. Agents are able to concurrently and continuously rework the solution as the problem parameters, constraints, or execution environment change. Agents consist of two major components, a partitioned distributed blackboard, and plug-ins. Plug-ins are software components that provide the agent's behaviors and operate by working with contents on the blackboard [109].



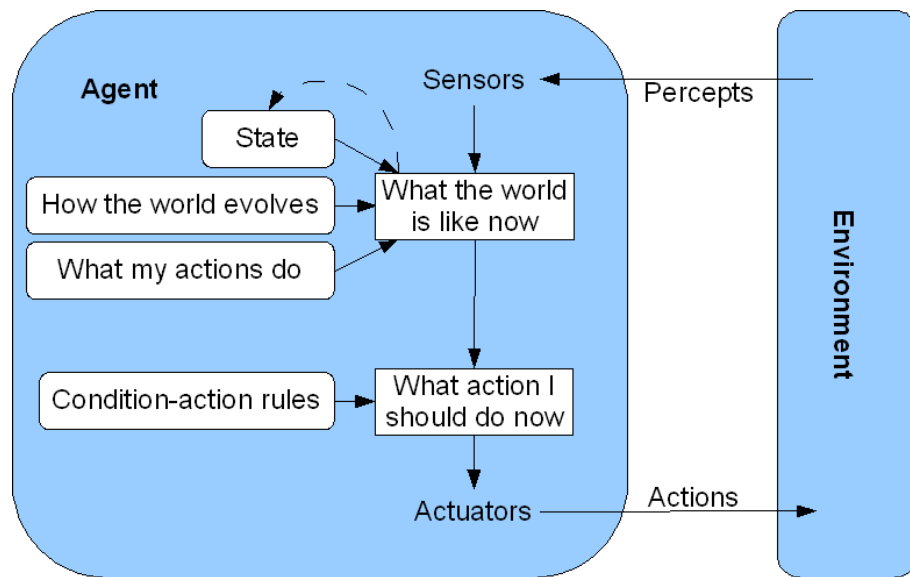


Figure 4.4: Production Agents [95]p49

#### 4.3.1.3 Production Systems

Production systems are also an extension of expert systems (see section 3.2.1.3), however they place more emphasis on how decisions are made based on the state of the KB. The general structure of production system agents is illustrated in figure 4.4. The KB is called *working memory* and is aimed to resemble short term memory. They also allow a designer to create a large set of condition-action rules called *productions* that resemble long term memory. When a production is executed it is able to cause changes to the environment or directly change the working memory. This in turn possibly activates other productions. Production systems typically contain a small working memory, and a large number of rules that can be executed so fast that production systems are able to operate in real time with thousands of rules [95].

An example of a production-rule agent development environment is called SOAR (State, Operator And Result). SOAR uses a KB as a problem space and production rules to look for solutions in a problem. IT has a powerful problem solving mechanism whereby every time that it is faced with more than one choice of productions (via a lack of knowledge about what is the best way to proceed) it creates an *impasse* that results in branching of the paths that it takes through the problem space. The *impasse* asserts *subgoals* that force the creation of *substates* of problem solving behavior with the aim to resolve the super-state *impasse* [62].

#### 4.3.2 Reactive Agents

Deliberate agents were originally developed using traditional software engineering techniques. Such techniques define pre-conditions required for operation and post-conditions that define the required output after operation. Some agents however, cannot be easily developed using this method because they maintain a constant interaction with a dynamic environment, hence they are called reactive agents. Reactive agents are especially suited for real-time applications where there are strict time constraints (ie. milliseconds) on choosing actions.

Reactive systems are studied by behavioural means where researchers have tried to use entirely new approaches that reject any symbolic representation and decision making. Instead

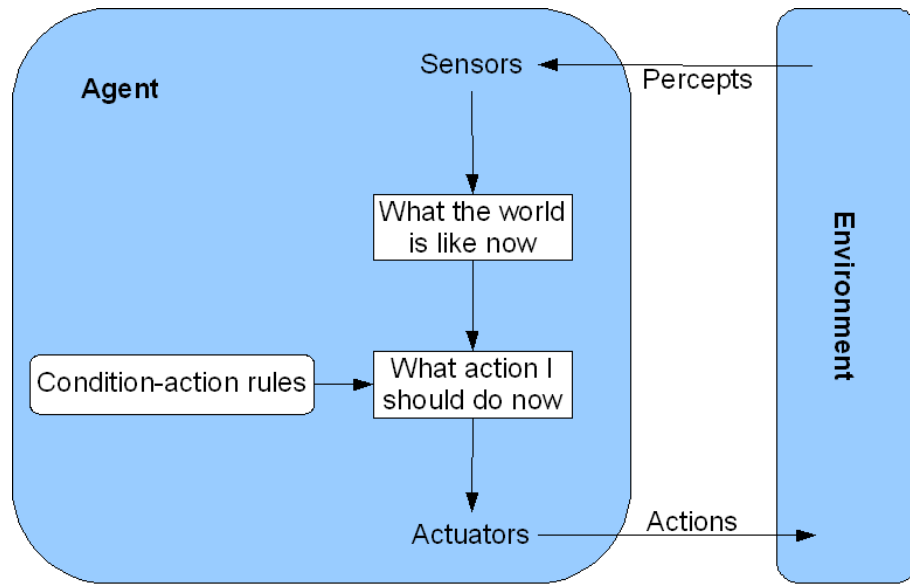


Figure 4.5: Reactive Agent [95]p47

Priority	Behavior rule	Description
1	<b>IF</b> detect an obstacle <b>THEN</b> change direction	Ensures that the agent avoids obstacles when moving
2	<b>IF</b> carrying samples and at the base <b>THEN</b> drop samples	Allows agent to drop samples in the mother-ship.
3	<b>IF</b> carrying samples and not at the base <b>THEN</b> drop 2 crumbs and travel up signal strength	Either reinforces a previous trail or creates a new one.
4	<b>IF</b> detect a sample <b>THEN</b> pick sample up	Collects samples.
5	<b>IF</b> sense crumbs <b>THEN</b> pick up 1 crumb and travel away from signal strength	Follows a crumb trail that should end at a mineral deposit. Crumbs are picked up to weaken the trail such that it disappears when the mineral deposit has depleted.
6	<b>IF</b> true <b>THEN</b> move randomly	Explores the area until it stumbles upon a mineral deposit or a crumb trail.

Table 4.1: Behavior Architecture of a Robot on Mars [116]

they argue that intelligent and rational behaviour emerges from the interaction of various simpler behaviours and is directly linked to the environment that the agent occupies [116]. The general structure of reactive agents is illustrated in figure 4.5

The main contributor of reactive agent research is Brooks with his *subsumption architecture* where decision making is realized through a set of task-accomplishing behaviors. Behaviors are arranged into layers where lower layers have a higher priority and are able to inhibit higher layers that represent more abstract behaviors [16].

A simple example of the subsumption architecture is a multi-agent system used to collect a specific type of rock scattered in a particular area on a distant planet. Agents are able to move around, collect rocks and return to the mother-ship. Due to obstacles on the surface of the planet, agents are not able to communicate directly, however they can carry special radioactive crumb that they drop on the ground for other agents to detect. The crumbs are used to leave a trail for other agents to follow. Additionally, a powerful locator signal is transmitted from the mother-ship, agents can find the ship by moving towards a stronger signal. A possible behavior architecture for this scenario is given in table 4.1.

Another reactive agent development environment is called Cybelle. It provides agent management as well as abstracting complex issues such as distributed programming, threading and time synchronization. Cybele treats anything that the agent is required to do as a reactive

activity. Activities can be delegated to different agents at run time while agents are also able to dynamically switch between different activities. Even agent interaction is modeled as the activity of one agent interacting with the activity of another agent [57].

Although reactive systems promise a relatively easy design they have some important limitations. Agents need to have sufficient information from the local environment to determine an appropriate action. Also, decisions cannot be made using previous information and inherently the performance is based on a short term view. The fact that intelligence is emergent means that behaviors are not fully understood, engineering such systems increases the difficulty of development. Finally, as more layers are introduced, the complexity of behaviors increases and the dynamics of the interactions between layers becomes hard to understand [116].

### 4.3.3 Communication Agents

Communication-centric agents tend to be simple compared to other agents described in this chapter. This is because they rely on the idea that intelligence is an emergent property of complex interactions between many simple agents. Four agent development platforms focusing on communication are described in this section.

*MadKit* is a modular and scalable multi-agent platform written in Java and built upon the an organizational model called *Agent/Group/Role* where agents are situated in groups and perform roles within the group. Communication is based on a peer to peer mechanism, and agents may be developed in many third party languages [37].

The *Open Agent Architecture (OAA)* can be used for building distributed communities of agents, where an agent is defined as any software process that meets the conventions of the OAA society. Communication and cooperation between agents is managed by facilitators, which are responsible for matching requests with the capabilities of different agents. An important feature of OAA is that when making a request the system does not need for the requester to specify (or even know of) a particular agent or agents to handle the request. Supported agents may be loosely categorized as: Application agents, that provide a collection of services. Meta-agents, used to assist the facilitators in coordinating activities of other agents. User interface agents, used to interact with humans that use the system, via a graphical user interface, handwriting, or speech.

*Aglets* are Java objects that can move from one host on the network to another, hence called *mobile agents*. This means that an agent can halt execution, dispatch to a remote host, and start executing again. When the agent moves, it takes along its program code as well as the information of objects it is carrying. A naming mechanism automatically assigns unique identities to new agent and classifies them into two categories: trusted and untrusted. The aglets built-in security manager checks whether the aglet is allowed to access different resources on a host PC as well as communicate with other agents, the decision to trust an agent is entirely up to the host [63].

*Swarm* is a software package for multi-agent simulation of complex systems, it comprises of a set of libraries which enable an agent to be written in the Objective-C or Java languages. Swarm also provides a hierarchical structure that defines a top level *observer swarm* with linked screen displays for the user. A number of *model swarm* are then created in the level below it,

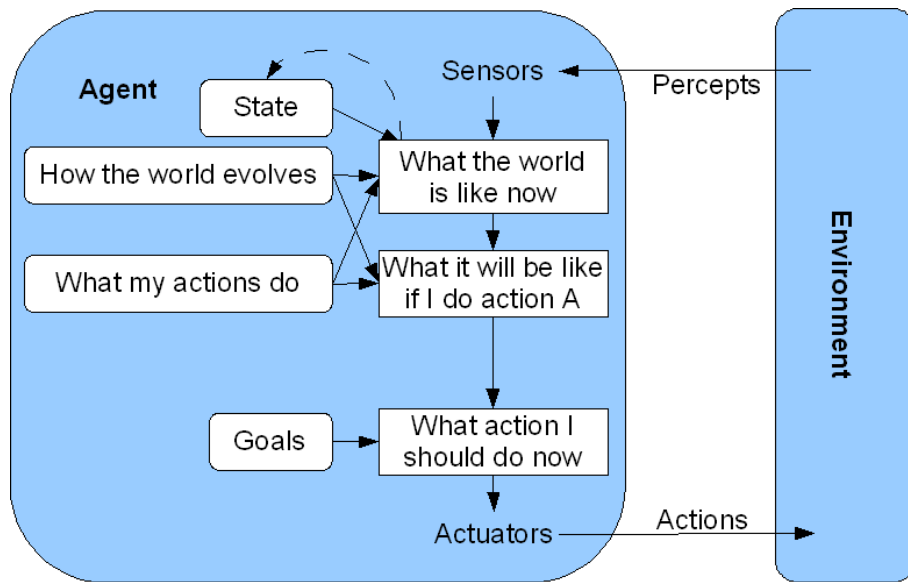


Figure 4.6: Goal Based Agents [95]p50

which in turn, create individual agents, schedule their activities, collect information and relay it to the observer swarm if required. In addition, a number of tools are provided to facilitate with the management of specific swarm properties [46].

#### 4.3.4 Hybrid Agents

Hybrid agents are capable of expressing both reactive and pro-active behavior. They do this by breaking reactive and proactive behavior into different subsystems called layers. The lowest layer is the *reactive layer* and it provides immediate responses to changes for the environment, similarly to the subsumption architecture. The middle layer is the *planning layer* that is responsible for telling the agent what to do by reviewing internal plans, and selecting a particular plan that would be suitable for achieving a goal. The highest layer is the *modeling layer* that manages goals. A major issue encountered when developing solutions with hybrid reasoning agents is that agents must be able to balance the time spent between thinking and acting. This includes being able to stop planning at some point and commit to goal, even if that goal is not optimal [116]. The general structure of hybrid agents is illustrated in figure 4.6

Due to the fact that hybrid agents attempt to exploit the best of both pro-active and reactive approaches, hybrid agent development platforms are more powerful. Consequently, a number of these have been developed, six are briefly described in this section and a seventh (the one chosen for this research) is described in greater detail in the end of this chapter.

RETSINA is a C++ based agent development platform that supports four basic agent types: Interface agents, that interact with users. Task agents, that help users perform tasks. Information agents, that provide access to information sources, and middle agents, that match agents that request services with agents that provide services. RETSINA has been applied to applications like financial portfolio management, web information management and logistics planning in military operations. Each agent consists of four reusable modules. The *communication and coordination* module accepts and interprets messages and requests from other agents. The *planning* module takes as input a set of goals and produces a plan that satisfies the goals. The *scheduling* module uses the task structure created by the planning module to

order the tasks. The *execution* module monitors this process and ensures that actions are carried out in accordance with computational and other constraints [105].

The Agent Building and Learning Environment (ABLE) developed by IBM provides a library for building intelligent agents using Java. The ABLE framework is essentially a set of Java interfaces and classes used to build JavaBeans<sup>2</sup> called AbleBeans. There are AbleBeans provided for accessing files and databases, as well as more complex operations such as rule-based inferencing, fuzzy logic, neural networks and decision trees. Developers can extend the provided AbleBeans to suite their requirements. AbleBeans may also be combined to create AbleAgents, some of the included AbleBeans and AbleAgents are:

- Data beans that read data from files and databases as well as provide some simple operations that may be performed to the data such as filtering and transforming data [12].
- Learning beans that provide encapsulated learning algorithms for neural network classification and prediction, decision trees and naive Bayes learning, self-organizing maps and temporal difference learning [12].
- Rules beans that allow the creation of inferencing engines using different algorithms like backward chaining, forward chaining, fuzzy logic and others [12].

The Procedural Reasoning System (PRS) is a set of tools aimed to implement agents based on the BDI model in the *lisp* language. It preserves control information in *plans* while managing the plan execution using a *database*. The database contains facts representing the system's view of the world and is constantly and automatically updated as new sensations are received. Plans describe a sequence of actions to be performed in order to achieve particular goals or to react to certain situations. PRS however is not able to distinguish individual actions within plans and hence, it cannot generate alternative plans. Consequently, a designer must create an extensive library of plans containing everything the agent will need in order to achieve its goals.

Plans are executed within *tasks*, which are runtime structures that monitor the execution of plans and are able to cancel plans from executing if one of their actions does not achieve the desired effect or the relevant goal is no longer active [41]. PRS has also been ported to other languages, for example UMPRS [55] is a C++ implementation and JAM [54] is a Java implementation of PRS. Both of these implementations involve developing agents in the same lisp-based format as PRS, but are also able to link to other software components from their respective language. The agent development platform used for this project is called JACK (see section 4.5), it was also inspired by PRS, however it allows for the development of agents without any prior knowledge of lisp language.

---

<sup>2</sup>JavaBeans is a portable, platform-independent software component model that enables developers to write reusable binary modules

## 4.4 Agent-Oriented Development

Agent-oriented development is concerned with the techniques of software development that are specifically suited for developing agent systems. This is an important issue because existing software development techniques are unsuitable for agents. This is because of a fundamental mismatch between traditional software engineering concepts and agents. Traditional techniques fail to adequately capture an agent's autonomous problem-solving behaviour as well as the complex issues involved in multi-agent interaction.

The first agent-oriented methodology was proposed by Wooldridge and is called *Gaia* [117]. Gaia is deemed appropriate for agent systems with the following characteristics:

- Agents are smart enough to require significant computational resources.
- Agents may be implemented using different programming languages, architectures or techniques.
- The system has a static organization structure such that inter-agent relationships do not change during operation.
- The abilities of agents and the services they provide do not change during operation.
- The system requires only small amount of agents.

Gaia splits the development process into three phases: Requirements, Analysis and Design. The requirements phase is treated in the same way as traditional systems. The analysis phase is concerned with the roles that agents play in the system as well as the interactions required between agents. The design phase is concerned with the agent types that will make up the system. The main services that are required to realize the agent's roles, and finally, the lines of communication between the different agents. The Gaia methodology was the inspiration for the more detailed methodology described in the next section [117].

### 4.4.1 The Prometheus Methodology

The *Prometheus* methodology suggests maximizing the power of BDI using subgoals whenever possible, and writing many simple alternative plans rather than more complex comprehensive plans. In other words, designing and coding for alternative ways to do things, even if only one way can initially be implemented. This requires an agent execution engine to provide some important features, in particular, if a plan fails then the agent still maintains the goal and tries to achieve it using other plans. The authors of Prometheus argue that using pre-compiled plans is much faster than on-line planning and is more sensitive to environmental changes as the engine chooses each plan as needed. If it fails due to the environment changing while executing, it then tries a new plan for the environment.

The Prometheus methodology has three stages. In the *system specification*, the designer firstly determines external interfaces such as actions, percepts and permanent data and then determines system goals, functionalities and scenarios. In the *architectural design* the designer defines the agents, incidents, interactions and also shared data structures. Finally, in the *detailed design* the designer defines capabilities, plans, belief structures and events.

The details of each of the stages can be viewed in chapter 6 which follows the Prometheus methodology, all design elements are clearly presented in that chapter.

## 4.5 JACK Intelligent Agents

JACK Intelligent Agents is a development platform for creating practical reasoning agents in the Java language using BDI reasoning constructs. It allows the designer to use all features of Java as well as a number of specific agent extensions. Any source code written using JACK extensions is compiled into regular Java code before being executed.

Each agent has beliefs about the world, events to respond reactively, goals that it desires to achieve, and plans that define what to do. When an agent is executed, it waits until it is provided with a goal to achieve or receives an event for which it can respond reactively, it then reasons using its beliefs whether to respond. If a response is required it selects an appropriate plan to execute in order to respond. JACK agents are based on the BDI reasoning model and can exhibit:

**Goal-directed behavior:** Where the agent focuses on the objective and not the method chosen to achieve it.

**Context sensitivity:** Keeping track of which options are applicable at each given moment using beliefs.

**Validation of approach:** Ensuring that a chosen course of action is pursued only for as long as applicable.

**Concurrency:** Behaviors in the agent are executed in separate, parallel and prioritized threads.

The reasons that JACK was chosen as the development platform for implementing the research ideas presented in this thesis are:

- It is commercially supported, this means that a number of support methods are available. This includes individual email-based support and collaborative support via a web-based discussion group of developers.
- Although a commercial JACK license can be expensive, educational institutions can obtain a fully supported site license in a heavily discounted price.
- It is based on the Java language that provides an extensive library of reusable objects, allowing for a quicker and better quality implementation. Java also handles complex issues such as memory management and concurrency.
- In contrast to other BDI reasoning development environments (see section 3.2.2.1) there is no need to understand lisp syntax when creating agents.
- Detailed and clear documentation is provided on how to develop agents.
- A cross-platform graphical editing suite is provided for easily developing agents.

- It is under constant development with new features released regularly. Some of the latest features include a *design tool* for drag-and-drop development, a *plan tracing* tool that provides the ability to trace the execution of plans and *JACK sim*, a framework for building and running repeatable agent based simulations.

Finally, the most influential reason for choosing JACK for this research, is that agent constructs can be *further extended* to include *new functionality* with *supporting documentation* being available on how make such extensions. The main contribution of this thesis described in chapter 5 is an extension of the BDI model. Using JACK therefore makes sense in this case because it already provides a good implementation of the BDI layer while also being extensible.

### 4.5.1 The JACK Agent Language

JACK provides a language for developing agent based systems using agent-oriented paradigms, the language is complete with a compiler, a powerful multi-threaded runtime environment and a graphical environment to assist with development. JACK agents are based on the BDI model of agency described in section 3.2.2.1. This means that equivalent software constructs had to be selected for each of the BDI components. In JACK, beliefs have been implemented as relational databases called *beliefsets*, however developers can also use their own Java-based data structures if needed. Desires are realized through goal events that are *posted* in order to initiate reasoning. This is an important feature because it causes the agent to exhibit goal-directed behaviour rather than action-directed behaviour, meaning that the agent commits to the desired outcome and not on the method to achieve it [5]. An intention is defined as a plan to which the agent commits to after choosing from a library of pre-written plans. The agent is able to abort a plan at any time depending on its beliefs and also consider alternative plans. There is however no provision for detailed real time planning, in other words changing a predetermined plan in some way to better suit changing environmental conditions or improving its performance. Norling [79] describes this as a weakness of the BDI model, its inability to support agent learning. Through the research presented by this thesis however it becomes possible to overcome this problem. Chapter 5 describes how learning is linked to the BDI model and also how JACK can be extended in order to support learning.

#### 4.5.1.1 Defining an Agent

The **Agent** class is used to define all functionality associated an agent. Agents are defined by extending this class and adding members and methods that are applicable to the application domain as shown in code listing 4.1. Note that the agent **MyAgent** extends **Agent** (line 1), it contains a beliefset **MyBeliefs** (line 2), it posts and handles a **MyEvent** event (lines 4,5) and uses the **MyPlan** plan for handling the event.

#### 4.5.1.2 Populating Beliefs

Beliefsets are relational databases that are used to maintain beliefs about the world. There are two types of beliefsets available, **ClosedWorld** and **OpenWorld**. **ClosedWorld** beliefsets assume that every fact about the world is stored in the beliefset at all times as being either true or



```

1 agent MyAgent extends Agent{
2     #private data MyBeliefs beliefsetName (...);
3
4     #posts event MyEvent postHandle;
5     #handles event MyEvent;
6
7     #uses plan MyPlan;
8 }

```

Code Listing 4.1: Extending the JACK Agent Class

```

1 beliefset MyBeliefs extends ClosedWorld{
2     #key field Type myKeyField;
3     #value field Type myValueField;
4     #indexed query myIndexedQuery(parameters);
5
6     #linear query myLinearQuery(parameters);
7
8     #complex query myComplexQuery(parameters){
9         //statements
10    }
11
12    #function query returnType myFunctionQuery(parameters){
13        //statements
14    }
15
16    #posts event EventType handle;
17 }

```

Code Listing 4.2: Defining a Beliefset

false, hence every possible tuple is always represented in the beliefset. **OpenWorld** relations model beliefs more like people, where only some of the answers are known to the agent. Some beliefs may be known to be true, others known to be false and others unknown. Agents can also include ordinary Java data structures if required. However, beliefsets provide a number of additional features designed specifically for developing agents, they are:

- They maintain consistency between individual beliefs.
- They allow for **OpenWorld** or **ClosedWorld** semantics.
- They provide the ability to post events automatically when some changes are made to the beliefset.
- They have the ability to bind individual beliefs with JACK plans used for making various decisions.

Beliefsets are defined by extending the **ClosedWorld** or **OpenWorld** classes and defining a number of fields and queries as shown in code listing 4.2. Beliefs are highly dependent on **key** fields. This is because when a belief is inserted with a **key** field that is not already in the beliefset, then a new belief is created and *added* to the beliefset. On the other hand, when a belief is inserted that has a **key** that already exists in the beliefset but has a different **value** field, then the previous belief is *replaced* with the new belief.

JACK uses queries for searching belief sets, **indexed** queries search using a hashing algorithm, **linear** queries search in sequential order, **complex** queries allow the combination of both **indexed** and **linear** queries, and finally, **function** queries allow normal Java code to be used.

#### 4.5.1.3 Specifying Goals

Goals are used to motivate an agent to take action, they are realized using events. JACK provides a number of different **event** types for modeling:

**Internal stimuli:** Events that an agent sends to itself that allow for ongoing execution of the agent and its reasoning process.

**External stimuli:** Events that contain messages from other agents or sensations received from the environment.

**Motivations:** Long term goals that the agent is trying to achieve.

Events cause the agent to choose between **plans** that it has available and then execute the selected plan until it succeeds or fails. If the execution of the plan succeeds, then the event is said to have succeeded. However, if the execution fails then the agent needs to perform additional reasoning depending on the type of event that has failed. Normal events are said to have failed after their associated plan fails and are useful for realizing reactive behaviours. BDI events however allow an agent to exhibit goal-directed behaviour, they allow for a number of plans to be selected and each attempted in turn. The associated events fail only after all chosen plans have been exhausted without success.

#### 4.5.1.4 Developing Plans

**Plans** are used to handle events by defining a sequence of actions for the agent to perform. Each plan is capable of handling only one type of event and is selected whenever its associated event is posted. If more than one plan has been defined for handling the same type of event then the agent is faced with a choice of which plan to select. Three mechanisms have been included in plans for intelligently making this choice. Firstly, the **relevant()** method may be defined in the plan to indicate when it is relevant to be executed. This is useful when the same event is posted with different internal properties and a different plan is required to handle it depending on these properties. Secondly, the **context()** method is used to query a beliefset by binding certain *logical members* in a plan to a belief. If no binding takes place because no relevant beliefs were found then the plan is not executed. Conversely, if multiple beliefs are binded then multiple applicable instances of the same plan are generated, one for each binding. The third choice becomes available if both of the previous two mechanisms fail to cause the selection of a plan, in this case a specially crafted plan can be implemented to explicitly make the choice, this is otherwise called *meta-level* reasoning. Finally, if meta-level reasoning is not used and a choice is required, then the JACK runtime simply chooses the plan that was first declared in the agent's source code.

A plan is executed by executing statements sequentially in the **body()** method. This method is a *reasoning method* and is different to a normal Java method in that each line is treated as a logical statement which may succeed or fail. The plan succeeds by executing all statements in the **body()** successfully. If any of the statements fails then the plan immediately also fails. The **body()** method can also call other reasoning methods that are executed in the same way, they can be used to help reduce complexity and make plans simpler to understand.

```

1 plan MyPlan extends Plan{
  #handles event MyEvent myEvent;
3  #reads data MyBeliefset myBeliefset;

5  static boolean relevant (EventType reference){
    // determines whether the plan is relevant
7  }

9  #logical VariableType myVariable;
  context(){
11    // binds myVariable to myBeliefset
  }

13  body(){
15    // The plan body.
    //Every line is treated as a logical statement
17  }

19  #reasoning method methodName (parameters){
    // some additional statements that can be executed
21  }

23 }

```

Code Listing 4.3: Defining a Plan

```

1 #capability CapabilityType extends Capability{
  //Beliefsets used by the capability, they can be internal only
  //or shared with other capabilities.
3  #private data ... (internal only)
  #exports data ... (created here but shared with other capabilities)
5  #imports data ... (created outside but available for use)

7  //Plans used by the capability.
  #uses plan ...

9  //Events used by the capability. Internal events are posted and handled
  //by internal plans. External events are either posted outside and
11 //handled here or posted here and handled outside.
  #posts event ...
13  #handles event ...
  #handles external event ...
15  #posts external event ...

17  // Sub-capabilities used by the capability
  #has capability ...

19 }
21 }

```

Code Listing 4.4: Defining a Capability

Additionally, all statements in reasoning methods can include JACK specific language syntax. Code listing 4.3 shows the skeleton code of an example JACK plan illustrating all of the components described.

#### 4.5.1.5 Grouping with Capabilities

**Capabilities** are used to group events, beliefsets and plans into groups that encapsulate a particular reasoning ability. This simplifies the design of agents and allows for directly reusing code by simply linking to the required capability. Additionally, capabilities can be structured such that they are composed of a number of sub-capabilities that combine simpler behaviours to realize more complex ones. This parent capability can be added to an agent to give it the complete ability. The definition of a capability requires a number of declarations for beliefsets, events and plans that can perform the relevant functionality as shown in code listing 4.4.

#### 4.5.1.6 Multi-Agent Teams

JACK Teams is an extension to the JACK platform that provides a team-oriented modeling framework. The Teams extension introduces the **team** reasoning entity that encapsulates teaming behaviour and **roles** for defining what an agent is required to do in a team. Because Teams is an extension of JACK, agent functionality is also available within a team. Team-oriented programming allows the designer to specify:

- What functionality a team can perform.
- What roles are needed in order to form a team.
- Whether an agent can perform a particular role within a team.
- Coordination of activities between team members.
- Shared knowledge between team members.

### 4.6 Summary

Agents can be used in software development to reduce complexity while increasing robustness. When viewed as software entities, agents inhabit a software environment, they obtain information from the environment by invoking software functions and can perform software actions such as sending messages on a computer network. There are however a number of important pitfalls in regards to using agents in real world systems: political pitfalls refer to people overselling agents for use in real world applications, management pitfalls refer to using agents without having a clear idea on why they are needed, conceptual pitfalls refer to believing that agents should always be used, analysis and design pitfalls refer to not exploiting related mature technologies when developing agents, agent-level pitfalls refer to the development of new agent platforms even when it is not required, and finally, society level pitfalls are concerned with multi-agent technologies.

Deliberate agents perform rational reasoning by taking actions after deliberating using their knowledge. Conversely, reactive agents reject any symbolic representation of decision making, instead their behaviour emerges from the interaction of various more simple behaviours directly linked to the environment that they occupy. Hybrid agents are capable of expressing both reactive and deliberative behavior. A key issue encountered when developing hybrid reasoning agents is that they must be able to balance the time spent between thinking and acting. This includes being able to stop thinking at some point and commit to a goal, even if the goal is not optimal.

JACK Intelligent Agents is a development platform for creating practical reasoning agents in the Java language using BDI reasoning constructs. It allows the designer to use all features of Java as well as a number of specific BDI reasoning extensions. Each agent has beliefs about the world, events to respond reactively, goals that it desires to achieve and plans that define what to do about goals or events that may arise. JACK agents can exhibit goal-directed behavior, context sensitivity, validation of approach and concurrency.

We now have a comprehensive understanding on how agents can be used in different applications. We also have developed an appreciation for a number of different types of agent development platforms currently available and their associated features. Furthermore, an understanding of how the JACK platform is suitable for this research has been achieved. The next chapter illustrates how to combine all of the work presented until now in order to design and implement a more complete reasoning model for agents.

# Chapter 5

## Cognitive Hybrid Reasoning Intelligent Agent System

This chapter introduces a novel, hybrid, agent reasoning model developed by fusing concepts from the previous chapter. It then gives a detailed insight on an implementation framework called *CHRIS* that is based on this model, and finally provides some example learning agents in which this framework is put into practice.

### 5.1 Introduction

This chapter describes the main contribution of this thesis. It brings together all the information described in previous chapters in order to define how to integrate learning abilities into the agent's reasoning process. It is split into three major sections. Section 5.2 provides an overview of a new hybrid conceptual reasoning model that was derived from reasoning models described in chapter 3. Section 5.3 reveals how this model has been implemented as an extension to the JACK agent development. This extension can be used as a framework for creating many different agent learning applications. Finally, section 5.4 provides two such examples of the framework being used and describes in detail the implementation.

### 5.2 Conceptual Model

This section tries to put three human reasoning models into the context of agent reasoning. It describes how the Decision Ladder (section 3.2.2.2), the OODA loop (section 3.2.2.3) and the BDI model (section 3.2.2.1) are complementary and in fact overlap in their processes. Each introduces some additional details that are not present in other models and after fusing together, it is possible to yield a new, hybrid and more detailed conceptual reasoning model.

#### 5.2.1 Fusing the Reasoning Model

The structure of the OODA loop is retained, but the decision ladder is divided such that appropriate components of it are placed within each of the four stages. The relevant components of the ladder now provide some extra detail about what sub-processes that are occurring in

each of the OODA stages. Three components of the ladder are also directly applicable to each of the major components of the BDI model, and are apparent within the new hybrid model.

#### 5.2.1.1 The Observation Stage

The observation stage is shown in figure 5.1. It incorporates the activation operation, alert state, observation operation and information state of the ladder. The activation operation can occur via obtaining new information from sensors in the environment or via feedback activation originating from the action stage.

The observation stage receives an implicit guidance feedback originating from the orientation stage that is useful to re-focus the attention of the observation process. The advantage of this, is for quickly re-processing important data in greater depth in order to improve accuracy. This can apply to situations where a great deal of processing is performed by the observation stage. In such cases, the observation stage firstly performs an overview scan of the data, providing some preliminary information. If the orientation stage then detects something of interest, it requests a deeper analysis to be performed to a specific part of the sensor information.

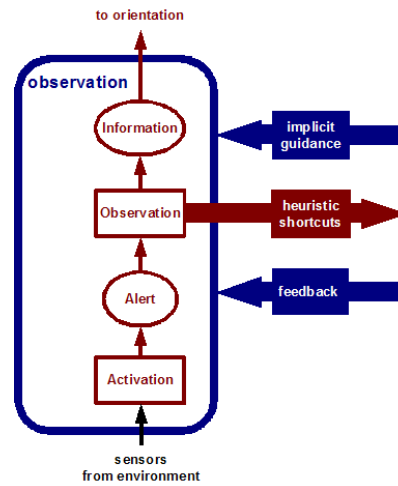


Figure 5.1: The Observation Stage

The observation operation also receives feedback from the decision stage. The aim of this feedback is to constrain its processing only to the data that is relevant to the agent's desires and intentions. This is useful for agents that are provided with large amounts of data, because there is no need to spend large amounts of time processing data that is irrelevant to the decisions that the agent needs to make. The observation stage also allows the agent to take pre-programmed heuristic shortcuts for particular situations.

#### 5.2.1.2 The Orientation Stage

The orientation stage illustrated in figure 5.2 effectively builds awareness about the environment. The identification operation uses the information produced by the observation stage and updates the agent's knowledge such that it understands the new state of the environment. Through inspecting figure 5.2 it is noticeable that identification includes components of the

OODA loop that have been slightly modified, such that they are more applicable for agent based reasoning. Each of these components influence the agent's understanding of a situation.

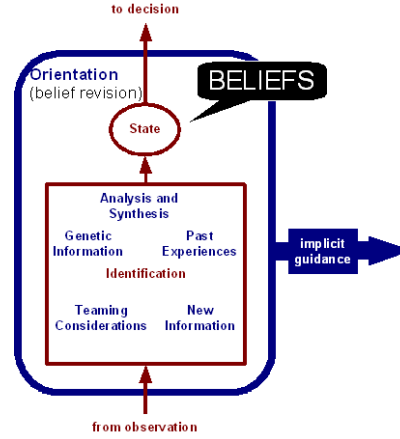


Figure 5.2: The Orientation Stage

Using genetic information, an agent can be given certain genetic traits, such as being more aggressive or conservative with its reasoning. Hence, one agent could identify a particular situation as advantageous, whereas another agent could identify it as dangerous and undesirable. Through teaming considerations the agent also understands its obligations towards the team that it is part of. This can influence its decisions such that it can take actions that are normally considered bad if the agent was acting alone, but they are good for the entire team.

At this point the agent has an updated knowledge base that includes the new information and past experiences as well as other useful information. The BDI model identifies this knowledge as Beliefs. That is, what the agent believes about the world. The output of the orientation stage therefore is the agent's updated beliefs about the entire world.

The question of how to implement beliefs is an open-ended question because it is very application specific. In the final section of this chapter an agent is described that uses a single variable for its beliefs and that is all that it needs to do its job successfully. While in chapter 6, a more complex range of beliefs is needed. An implementation of an expert system would also be applicable for a KB that is able to maintain consistency as new information arrives. The JACK agent development environment provides beliefsets that are relational databases for this purpose.

### 5.2.1.3 The Decision Stage

In the decision stage, as illustrated in figure 5.3, the agent decides what it needs to achieve depending on its beliefs, this encompasses the top-most section of the decision ladder. The agent evaluates the goal options available to it, and predicts the consequences of trying to achieve that particular goal. The output of the decision stage is a **Target**, a goal that the agent has enabled and is trying to achieve. This is also called a desire when identified through the BDI model.

The decision ladder indicates that it is possible to use heuristic shortcuts to identify desires without going through the entire decision stage. This would account for much faster responses for such situations. In addition, once a desire has been selected, it can be used as feedback to



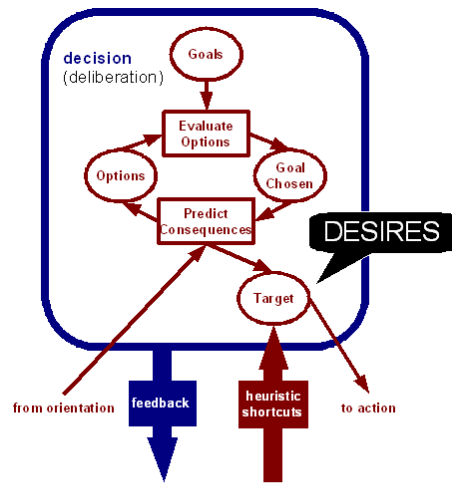


Figure 5.3: The Decision Stage

the observation stage in order to constrain further processing to specific data that is relevant for achieving that desire.

#### 5.2.1.4 The Action Stage

The final stage is the action stage. Here, the agent chooses from a range of possible tasks, which it will commit to in order to satisfy its desires. However, a task may also be selected via implicit guidance. This forces the agent to act spontaneously without any decision making. Such a shortcut, for example, could be used to evade immediate danger.

When a task has been selected, it means that the agent has committed to a specific way of achieving that goal. In the BDI model, this is identified as an intention which may in-turn be used as feedback to re-activate the decision loop.

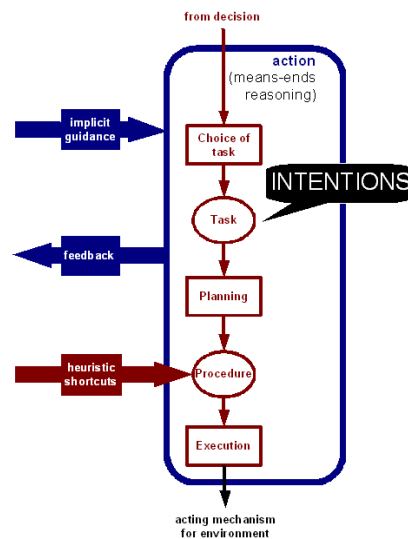


Figure 5.4: The Action Stage

## 5.2.2 Integrating Learning

For the purpose of this thesis, agent learning is the union of three properties. The first being dynamic development of new behaviours, through the combination of simpler tasks. Secondly, an improvement in the agent's performance such that it performs faster and/or more efficiently. Thirdly, being able to adapt to changes in the environment. Therefore, if an agent is placed in an unfamiliar situation it will still make, and improve upon, decisions towards achieving its desires. When the agent subsequently encounters a familiar environment it will can still use its previous knowledge for further decisions.

Urlings in his PhD thesis [110] describes that the decision ladder can be segmented into three levels of expertise. The skill level represents very fast, automated sensory-motor performance and it is illustrated in the ladder via the heuristic shortcut links. The rule level represents decision making based on rules and/or procedures that have been pre-defined, or derived empirically using experience, or communicated by others. Finally, the knowledge level represents behaviours during less-familiar situations when the agent is faced with an environment where there are no rules or skills available. In such cases, a more detailed analysis of the environment is required.

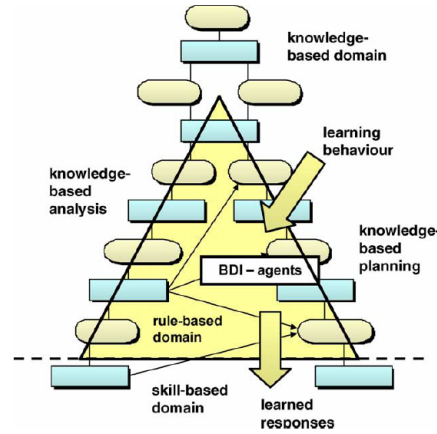


Figure 5.5: Introducing Learning in BDI Agents [110],p82

Furthermore, Urlings [110] suggests that learning involves crossing the skill levels. Specifically, a learning agent placed in an unfamiliar environment would be at first operating in the knowledge level. As the learning agent gains experience it builds a rule-base for situations presented in that particular environment. As the rule-base becomes larger, the agent uses its new rules more frequently and consequently drops to the rule level. In a similar way, as the learning agent improves the performance of its rules, it gains confidence on using them. It then subsequently selects them automatically with less or no pre-processing involved. Hence, the agent crosses to the skill level of operation. This process is illustrated graphically in figure 5.5.

The ability to learn can be achieved via appropriately interfacing an additional learning stage in parallel to the main decision loop as shown in figure 5.6. The learning stage takes an instance of the state and generalizes it into a perception.

The reinforcement learning component learns how to map perceptions to actions so as to maximize its rewards [104]. The numerical reward is provided by a reinforcement signal which is generated by a reward function. The reward is used to communicate to the RL algorithm the

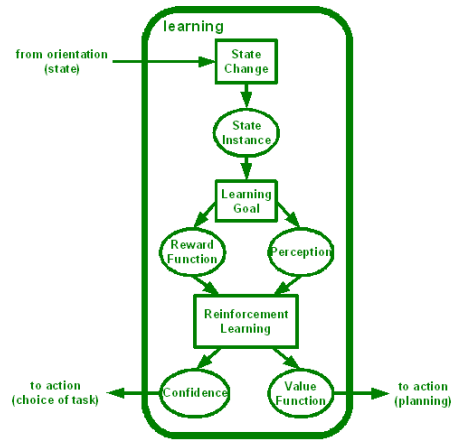


Figure 5.6: The Learning Stage

agent’s learning goals by defining how good or bad that particular state is [104]. An important concept to understand is that the reward function only communicates what the agent should achieve and not how to achieve it. This is clearly explained in the following quote by Sutton and Barto.

A chess-playing agent should be rewarded only for winning, not for achieving sub-goals such as taking its opponent’s pieces or gaining control of the center of the board. If achieving these sorts of goals were rewarded, then the agent might find a way to take the opponent’s pieces even at the cost of losing the game [104],p56.

The RL algorithm can provide five useful outputs which can be used by different operations within the new reasoning model. Currently two of these are illustrated in figure 5.6, the other two can be included with future work.

- *The Model* produced by the RL component that mimics the behaviour of the environment. Hence, it can be used to predict how the environment will respond to the agent’s actions [104]. This information can be useful when the agent is predicting consequences in the decision stage.
- *The Policy* defines the complete behaviour of the agent, the way it behaves at any given time [104]. A pure BDI reasoning agent follows a policy that is dictated by the choices emerging from its beliefs, desires and intentions. The learning component can however learn this policy at run-time by observing the actions it takes in different situations. It can also evaluate this policy (see below) and calculate another policy that would yield better rewards. This policy is presented to the agent when it is choosing a task in the action stage that may be used as an alternative to the one dictated by the BDI reasoning.
- *Policy Evaluation* can provide a numeric value for a given policy. A higher value would indicate that following this policy would yield greater rewards [104]. This information is useful when the agent is evaluating its options in the decision stage. This way the agent can be provided with the ability to choose desires that would yield higher rewards.
- *The Value Function* also provides a numerical value. However, this value is only for a particular state or state-action pair. That is, for a given state of the environment, the

value of how good it is to be in that state or how good it is to make a specific action from a specific state [104]. The agent can therefore try to avoid states and actions with a small value. This information is useful when the agent is planning in the action stage.

- *The Confidence Value* provides an indication of how well an agent has learned and is useful for choosing whether to take actions suggested by the learning component or not.

### 5.2.3 The Complete Picture

A complete picture of the new reasoning model is shown in appendix B. Two versions of the reasoning model are shown in order to illustrate how learning is integrated into the reasoning process.

The first version illustrates how all of the reasoning stages fit together. The picture is printed in colour in order to clearly illustrate which parts belong to which of the three models that were fused to create it. Components in a red colour belong to the Decision Ladder, components in blue colour belong to the OODA loop, and the black components belong to the BDI model. It can also be seen how the OODA loop stages divide and enclose the decision ladder and also how in the orientation stage a number of relevant OODA processes are included within the identification process.

The second version reveals how the learning module effectively provides the *implicit guidance* link indicated in the OODA loop between the orientation stage and the action stage. This link is severed from the original model as illustrated with a dotted line, and recreated using the learning stage. However, due to the greater detail known about how the link is achieved, it is possible to link down to the ladder level within the OODA stages.

## 5.3 Implementation Framework

The Cognitive Hybrid Reasoning Intelligent Agent System (*CHRIS*) is an implementation of the conceptual model described previously an extension to the JACK agent development environment. The new framework provides an abstract way to create learning agents without the need to worry about how the internal learning algorithms operate.

*CHRIS* equips an agent with the ability to learn from actions that it takes within its environment, however, the way that the agent behaves or how the agent learns is totally controlled by the designer. In fact, design decisions made have a great impact on the effectiveness and speed of learning achieved.

This is difficult to grasp initially because it is easy to think that while using learning there is no need for complex problem solving. This is true for the most part, however what learning does is to transform a particular problem into a (hopefully simpler to understand and code) learning problem. This is explicitly indicated in *CHRIS* through its inclusion of **LearningGoals**. In *CHRIS*, without a LearningGoal there can be no learning, in fact the learning problem is totally encapsulated within a LearningGoal. While the system is executing, the agent learns using that LearningGoal and changes its behaviour accordingly. The aim of the designer therefore becomes creating LearningGoals that will cause the agent to learn to behave in a way that

solves the original complex problem. Identifying the BDI states in the conceptual model gives reference upon which to extend JACK.

### 5.3.1 The General Architecture

*CHRIS* defines the agent reasoning process into five stages based on the new hybrid model. Each of the stages were implemented in different Java packages which need to be imported for successful compilation of source code. Two additional packages were also created, they are all listed below with a brief description for each one.

- **Agent:** Provides an extended JACK learning agent, the *CHRIS* capability and associated Java interface classes.
- **Observation:** Receives sensations and converts raw data into usable information.
- **Orientation:** Uses new information to update the agent's beliefs accordingly.
- **Decision:** Manages goals that the agent is trying to achieve.
- **Action:** Executes plans that cause the agent take different actions.
- **Learning:** Modifies the actions taken by the agent according to user specified Learning-Goals.
- **Logging:** Controls the logging functionality built into the framework.
- **Util:** Some utility classes that have been developed while creating *CHRIS*.

### 5.3.2 Extending the JACK Platform

The **Agent** package was specifically developed to allow for easy integration of JACK with the *CHRIS* framework. A number of Java classes and a JACK capabilities are used for this purpose.

The *CHRIS* capability encapsulates all plans, events and sub-capabilities that are required for the framework to operate correctly. It contains the **ObservationStage**, **OrientationStage**, **DecisionStage**, **ActionStage** and **LearningStage** capabilities. It also posts/handles required events and imports/exports required internal data. Finally it contains methods for easily saving and loading a learned value function for operation with a specific environment.

**CHRISConstants** is a Java interface that provides a number of static constants that are used throughout the framework. Table 5.1 lists the different constants available and a brief description of their use. Additionally, the **CHRISFunctions** Java interface provides function declarations that are used throughout the framework.

**LearningAgent** is an abstract Java class that extends JACK's **Agent** class. It implements the **CHRISFunctions** interface and it provides default implementations for most of its functions. Some of the functions however are application specific and do not have default implementations. These functions are required to be implemented by the designer of an agent that extends **LearningAgent**. In addition, some pre-implemented functions contain very simple default implementations, it may be preferred to override them by providing an alternative

Prefix	Postfix	Description
SUBGOAL_	PROCEDURAL	Identifies the procedural sub goal type
	PARALLEL	Identifies the parallel sub goal type
	OPTIONAL	Identifies the optional sub goal type
	RESIDENT	Identifies the resident sub goal type
	EITHER	Identifies the either sub goal type
LEARNING_	ACTIVE	Agent learns and changes its behaviour to improve its performance
	PASSIVE	Agent learns a pre-written behaviour, learning has no control over the actions taken
RL_	SARSA	Learn using the Sarsa learning algorithm
	QLEARNING	Learn using the QLearning learning algorithm
POLICY_	EGREEDY	Use EGreedy action selection for active learning
	SOFTMAX	Use SoftMax action selection for active learning

Table 5.1: The CHRISConstants

implementation in the agent. Required methods are very important, they are the ones that *hook* the architecture code with code from the specific application into one complete working system. Implementing a JACK agent using `LearningAgent` is shown in code listing 5.1.

### 5.3.3 The Hybrid Reasoning Layer

#### 5.3.3.1 Observation Package

The observation package is intended to receive sensations from the environment, alert the agent depending on the sensation received and finally convert any raw data contained within the sensation into usable information. Note that events and plans have been fully implemented and are encapsulated within the `ObservationStage` capability, which is itself encapsulated in the `CHRIS` capability. However, these may be easily changed by defining some specified methods in the agent.

The process begins by handling a `Sensation` event. This event is posted by the agent when some new data is received from the environment using the `[public activateDecisionProcess( Object data )]` method within the agent. The method was specifically included so that it may be called by a middleware program that interfaces the agent with its environment.

The event is handled by the `Activation` plan, the default behaviour is to post only a single `Alert` event that contains the data from the sensation. This may be changed by overriding `[public List generateAlertData( Object sensationData )]` within the agent. Notice that the function returns a `List`, this is important because a separate alert event will be posted for each item in the list. This provides the agent with the ability to spawn multiple decision processes concurrently for single sensations.

The `Observation` plan handles the `alert` event and is intended to convert data within the alert event into useful information that may be subsequently used by the rest of the decision process. The default behaviour is to assume that the data contained within the alert event does not require any further conversion, and it posts an `Information` event using that data directly. This behaviour may be changed by overriding `[public Object generateInformationData( Object alertData )]` within the agent.

```

1 import edu.unisa.chris.agent.*;
import edu.unisa.chris.observation.*;
3 import edu.unisa.chris.orientation.*;
import edu.unisa.chris.action.*;
5 import edu.unisa.chris.decision.*;

7 public agent MyLearningAgent extends LearningAgent{
    #has capability CHRIS chris;
9
    public MyLearningAgent(){
11        super("MyLearningAgent");
    }
13
    public List getActionsAvailable(StateInstance state){
15        //code that returns a List of actions
    }
17
    public void updateWorldState(Object info){
19        //code to update the agent's state
    }
21
    public void activateDecisionProcess(Object data){
23        //post a Sensation event with data object
    }
25
    public void loadLearning(String file){
27        chris.loadLearning(file);
    }
29
    public void saveLearning(String file){
31        chris.saveLearning(file);
    }
33 }

```

Code Listing 5.1: Implementing a Learning Agent

### 5.3.3.2 Orientation Package

The function of the orientation stage is to take information received from the environment and update the agent's knowledge about the world accordingly. This function is primarily performed by the **Identification** plan which handles the information event posted by the observation stage. The identification plan requires that the [**public void updateWorldState(Object informationValues )**] method is implemented within the agent. This method is responsible for updating the agent's knowledge base with the new information. It needs to be implemented because it is application specific.

**State.java** is a Java interface that may be used to encapsulate the agent's knowledge about the world, it declares a number of functions that are required to be implemented by a class that extends it. For example, the [**public StateInstance getInstance()**] method is required to generate a *snapshot* of the entire state and return it as a **StateInstance**.

State is normally a continuously updated repository of the agent's knowledge of the environment, as such it always represents the agent's *current* view of the world. As soon as the state is updated with new information, the previous information is lost in place of a new state that contains the new information. The reason, therefore, for having **StateInstances** in the framework is to provide the ability to capture changes in the agent's state. This is a required feature in order to realize learning.

The [**public void stateChange()**] method provides the designer with a hint that something needs to happen when the state becomes updated. Specifically, the **CurrentStateInstance** beliefset needs to be updated with the current **StateInstance**. This is very important be-

```

package edu.unisa.chris.examples.bandit;
2 import edu.unisa.chris.orientation.*;
import java.util.*;
4
public beliefset BanditState extends ClosedWorld implements State{
6     #value field double currentValue;
    #indexed query get(logical double currentValue);
8     #function query double getCurrentValue(){
        logical double currentValue;
10         Cursor c = get(currentValue);
        if(c.next())
12             return currentValue.as_double();
        return 0.0;
14     }

    public void moddb(){
16         stateChange();
18     }

    #posts event StateChange stateChange;
    public void stateChange(){
20         postEvent(stateChange.stateChange(getInstance()));
22     }

    public StateInstance getInstance(){
24         try{
26             return new BanditStateInstance(getCurrentValue());
28         }
        catch(BeliefSetException e){}
30         return new BanditStateInstance(0.0);
32     }
}

```

Code Listing 5.2: Defining a State Using a Beliefset

```

public view TTTState implements State{
2     private int[] cells = new int[9];
    private int wins;
4     private int losses;
    private int draws;

6     #uses data CurrentStateInstance currentStateInstance;

8     public void stateChange(){
10         try{
            currentStateInstance.add(getInstance());
12         }
        catch(Exception e){
14             System.out.println(e);
16         }
    }

18     public StateInstance getInstance(){
        int[] newCells = {cells[0], cells[1], cells[2],
20                        cells[3], cells[4], cells[5],
                        cells[6], cells[7], cells[8]};
22     return new TTTStateInstance(newCells, wins, losses, draws);
    }

24

26     public void updateState(String update){
        //code to update the cells, wins, losses and draws
28     }

30     /** some code omitted for clarity */
}

```

Code Listing 5.3: Implementing a State Manually



Type	Posting Method	Parent Succeeds	Parent Fails	SubGoal Succeeds	SubGoal Fails
Procedural	Sequential in order	All subgoals succeeded	Any subgoal fails	Post next subgoal	Parent fails
Optional	Sequential as chosen	Any choice succeeds	All choices failed	Parent succeeds	Choose next goal
Parallel	Parallel	All subgoals succeeded	Any subgoal fails	No effect	Parent fails
Either	Parallel	Any subgoal succeeds	Any subgoal fails	Parent succeeds	Parent fails
Resident	Parallel	External source	External source	Repost sub goal	Repost sub goal

Table 5.2: Decision Stage Subgoal Behaviour Types

cause learning algorithms monitor the **CurrentStateInstance** beliefset during processing, if this beliefset is not updated, the learning algorithms will not function correctly.

State is intended to be implemented either with a JACK view or a JACK beliefset. There is an important difference on how the two are implemented. A JACK beliefset is able to post events, therefore the agent can post a **StateChange** event to update **currentStateInstance** as shown in code listing 5.2. Conversely, a JACK view is not able to post events, however it can directly obtain a handle to the **CurrentStateInstance** beliefset and update it manually as shown in code listing 5.3.

### 5.3.3.3 Decision Package

The decision stage provides a very useful layer of high level goal management with pre-defined behaviours separated into categories. Using the decision stage a designer needs only to populate the **Goals** beliefset with appropriate information to instigate complex high-level goal behaviours that would otherwise require multiple definitions of JACK events and plans. An additional advantage of using the decision stage is the ability to use learning when choosing between different high-level goals.

Using the decision stage, the agent’s goals are viewed as a hierarchy, starting with primary goals and branching down to lower level subgoals. The behaviour of a goal is defined firstly by the composition of its subgoals, and secondly by the type of its sub goals. Goals positioned at the bottom of the hierarchy, do not have any subgoals. They are posted as targets, using the **Target** event and are subsequently handled by the action stage for means-ends reasoning.

The current implementation of the decision stage provides five subgoal categories which post subgoals in different ways. It is possible to also add more categories in the future. The different categories are now described and are summarized in table 5.2.

**Procedural** sub goals are activated in sequence in the order defined by their **proceduralIndex**. They are only activated after their preceding subgoal has completed successfully. When all procedural subgoals are completed successfully, the parent goal also succeeds. If any of the subgoals however fail, the parent goal will also fail and no subsequent subgoals will activate.

**Parallel** subgoals are activated concurrently resulting in parallel threads of reasoning. When all parallel subgoals succeed then the parent goal succeeds. If any of the parallel subgoals fail then the parent goal will also fail. An additional feature provided is that when any subgoal fails then all other subgoals are instantly stopped but are not treated as failed.

**Resident** subgoals are activated concurrently like parallel goals. However, they have no effect on the success or failure of their parent goal. In fact, resident subgoals are always automatically reactivated regardless if they succeed or fail as long as their parent goal is active. The primary reason for including resident goals, is for the agent having the ability to

```

1 #key field String name;
  #key field String parent;
3 #value field int type; //0=procedural, 1=parallel, 2=optional, 3=resident, 4=either
  #value field boolean active;
5 #value field int priority;
  #value field int successCount;
7 #value field int failureCount;
  #value field int proceduralIndex;

```

Code Listing 5.4: Fields of the Goals Beliefset

always have particular goals activated. Hence, causing it to take actions regardless on any other goals that are active. Resident goals are automatically stopped when their parent goal is deactivated.

**Either** subgoals are also executed in parallel. However, if one of them fails, then the parent goal fails and other subgoals are automatically stopped. An important difference with **parallel** is the way that the parent goal succeeds. If any subgoals succeed, then the parent subgoal succeeds and all other subgoals are stopped.

**Optional** subgoals allow the agent to choose between the different subgoals. If the chosen subgoal fails, then another subgoal is attempted, until all subgoals have been tried and failed, in which case the parent goal fails. If any of the chosen subgoals succeed then the parent goal also succeeds. An important feature of optional subgoals is that their implementation allows an agent to learn which of its subgoals gives better results and choose that instead of other choices.

The **Goals** beliefset contains the definition of the goal hierarchy for a particular application. The fields in the Goals beliefset are shown in code listing 5.4. A designer is therefore only required to populate the Goals beliefset with appropriate information to define a goal hierarchy for the particular application. The goals beliefset may also be initialized using a text file in JACOB<sup>1</sup> format.

The *GoalsGenerator* program was created and included within the *CHRIS* framework, it provides a graphical user interface in order to simplify the development of the goal hierarchy. A screen capture of the GoalsGenerator is shown in figure 5.7. After defining the required goals, GoalsGenerator saves them into text file in the JACOB format, which that can then be used to directly populate the **Goals** beliefset. An additional feature of GoalsGenerator is the ability to export a hierarchy into a special text format used by the open source GraphViz [44] software. Using GraphViz, it is possible to generate a graph of the hierarchy with colour coded nodes, and different arcs corresponding to the different types of goals as shown in table 5.3. Figure 5.8 is an example hierarchy showing all different types of goals. Notice that the priority of goals is indicated in brackets within nodes, the **proceduralIndex** is indicated as a number appearing next to arcs, and targets (leaf nodes) are explicitly indicated with a border around the nodes.

---

<sup>1</sup>JACOB is a format for structuring information in text files such that they can be used to populate JACK beliefsets

Type	Node Colour	Arc Shape
Procedural	Yellow	Filled arrow
Optional	Light blue	Diamond
Parallel	Brown	Three pronged fork
Either	Dark blue	Reverse filled arrow
Resident	Green	Filled circle

Table 5.3: GoalGenerator Graph Codes

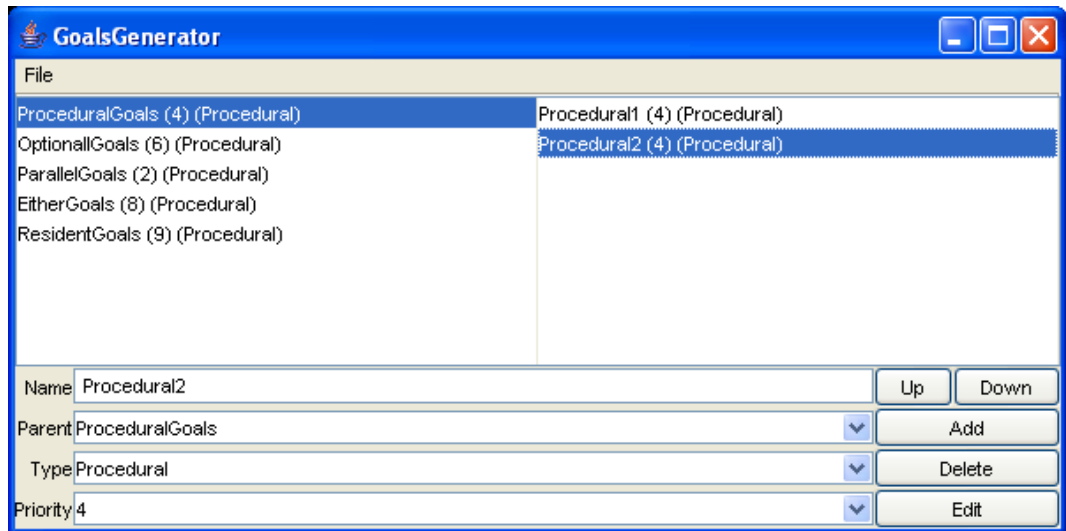


Figure 5.7: A Screen Capture of GoalsGenerator

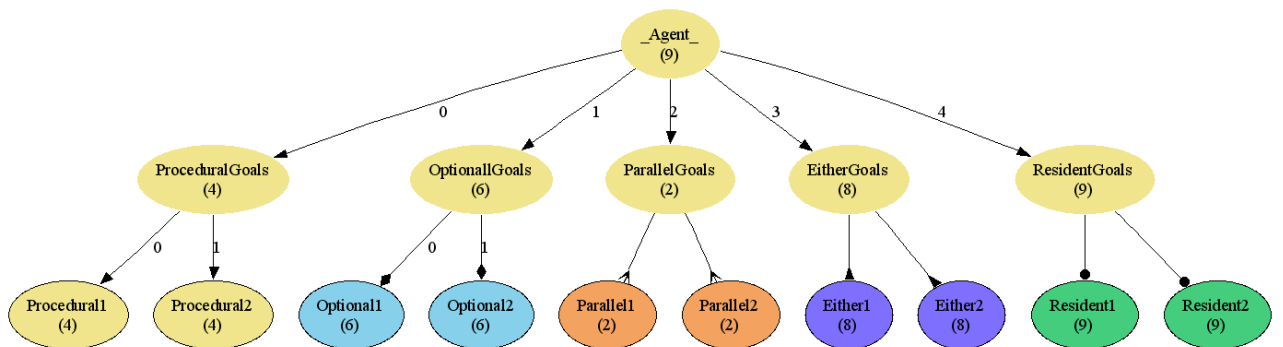


Figure 5.8: Example Goal Hierarchy Graph Generated by GoalsGenerator

```

2  public class MyAction extends Action{
    //class members required for the action
4     int example1;
    String example2;

6     public MyAction(int e1, String e2) {
        super(" ActionNamePrefix"+e1," ActionCategory");
8         example1 = e1;
        example2 = e2;
10    }

12    public boolean equivalent(Action other) {
        MyAction otherAction = (MyAction) other;
14        if(example1 == otherAction.example1 &&
            example2.equals(otherAction.example2))
16            return true;
        return false;
18    }
20 }

```

Code Listing 5.5: Extending the Action Class

#### 5.3.3.4 Action Package

A JACK-based agent system uses events and plans to define different behaviours that an agent can exhibit. A behaviour is executed by posting an event which is handled by a plan which contains any relevant code that causes the agent to exhibit the desired behaviour. A plan is also able to post other events and the JACK language provides a number of ways in which events can be posted and handled. This makes JACK a powerful platform for building complex agent behaviours. The *CHRIS* framework also introduces **Actions**. They are required to be explicitly defined for all lowest-level, simplest behaviours of an agent and are defined in two parts:

1. A class that extends the **Action** abstract class.
2. A plan that handles **ActionEvent** event with a definition of the [**static boolean relevant()**] function that returns **true** if **ActionEvent** contains the specific type of action handled by this plan.

The advantage of this method is that all low level behaviours are initiated by a *single* event type, **ActionEvent**, that contains an **Action** object. Complex behaviours can now be defined by a collection of **Action** type objects that are used to post **ActionEvents**. More importantly, modifying the collection of **Actions** during execution can yield changes in the agent behaviour *on-line*. **Actions** are fully exploited to realize agent learning in the *CHRIS* framework. A collection of actions is maintained and learning is achieved using standard reinforcement learning algorithms that post different **ActionEvents** accordingly.

**Actions** are defined as the smallest unit of behaviour that an agent can execute to influence its environment. Any complex agent behaviours should therefore be comprised of different combinations of actions. Different **Action** objects should at-minimum have a unique name but may also define other variables that are required for executing the action, as shown in code listing 5.5. **MyAction** is an example of how to define an action that contains two class members, code listing 5.6 shows the definition of a plan that can be used to handle action **MyAction**.

```

import edu.unisa.chris.action.*;
2 public plan MyActionPlan extends Plan{
    #handles event ActionEvent ev;
    #uses data Actions actions;

    public boolean relevant(ActionEvent ev){
        return ev.action instanceof MyAction;
    }

    body(){
        actions.add(ev.action,true);
        MyAction action = (MyAction) ev.action;
        //code that executes the behaviour of the action
        ...
        actions.add(ev.action,false);
    }
}

```

Code Listing 5.6: Implementing an Action Plan

```

1 public class BanditLearningGoal extends LearningGoal{
    //required class members
3
    public MyLearningGoal(parameters) {
        super("LearnSomething", CHRISConstants.RL_QLEARNING, CHRISConstants.POLICY_EGREEDY);
        //other initialization
7    }

    public Perception getPerception(edu.unisa.chris.orientation.StateInstance state) {
        //returns a perception that is applicable to this learning goal
11    }

    public double rewardFunction(Perception before, Action actionTaken, Perception after) {
        //returns a reward for the action taken
15    }

    public int getGoalStatus(edu.unisa.chris.orientation.StateInstance state) {
        //returns the status of the goal this is used to stop
        //the learning algorithm when the goal has succeeded
        return CHRISConstants.GOAL_ACHIEVED;
21    }
}

```

Code Listing 5.7: Defining a LearningGoal

### 5.3.4 The Learning Layer

The learning stage uses the flexibility provided by the **Action** package to allow an agent to change its behaviour on-line by learning from its experiences. The learning implemented is based on standard RL algorithms. Although it is possible to expand the system by implementing additional (more complex) RL algorithms that could provide better learning performance for particular applications.

#### 5.3.4.1 Defining a learning problem

The **LearningGoal** class is the heart of the learning package. Without a **LearningGoal** there can be no learning. Learning is activated by attaching a **LearningGoal** to a **Target** goal. This is done by adding the name of the **LearningGoal** and the name of its associated **Target** as a belief in the **LearningGoals** beliefset provided by the framework.

Defining a **LearningGoal** involves extending the **LearningGoal** class and providing implementations for some of its methods, a sample skeleton code is shown in code listing 5.7. The constructor requires a name for the **LearningGoal** as well as specifying the algorithm and policy type used for learning. Currently the framework supports **Sarsa** and **QLearning** for the

Name	Default Value	Description
equiprobability	1	The value $\tau$ for SoftMax policy
stepSizeParameter	0.1	The value of $\alpha$ for RL algorithms
discountFactor	0.9	The value of $\gamma$ for RL algorithms
variableStepSizeParameter	false	Causes the stepSizeParameter to be gradually reduced as more actions are taken
exploration	0.1	The value of $\epsilon$ the EGreedy policy
skillControl	false	Enables the SkillControl extension
learningTypeControl	true	Allows SkillControl to switch learning types based on confidence
explorationControl	true	Allows SkillControl to change exploration based on confidence
explorationChangeExponential	true	Causes exploration to be changed exponentially based on confidence
stepSizeControl	false	Allows SkillControl to change the stepSizeParameter based on confidence
stepSizeChangeExponential	true	Causes stepSizeParameter to be changed exponentially based on confidence
activeThreshold	0.8	The threshold of confidence where learning switches to active
passiveThreshold	0.4	The threshold of confidence where learning switches to passive
explorationMinimum	0.0	The minimum exploration when confidence is maximum
explorationMaximum	1.0	The maximum exploration when confidence is minimum
stepSizeMinimum	0	the minimum stepSizeParameter when confidence is maximum
stepSizeMaximum	0.1	The maximum stepSizeParameter when confidence is minimum
passiveTarget	null	The name of the target that should be posted for passive learning observation
replaceDuplicatePerception	false	Causes the algorithm to replace a duplicate perception in the value function, otherwise a duplicate perception is discarded
waitForActionFinish	true	Causes RL algorithms to wait for an action to finish before a new iteration
waitForStateInstanceUpdate	false	Causes RL algorithms to wait for a new StateInstance update before a new iteration
waitOrderActionState	true	If waiting for both action to finish and StateInstance update, this controls the order in which RL algorithms wait for them

Table 5.4: LearningGoal Parameters

learning algorithm and **EGreedy** and **SoftMax** for the policy. A **LearningGoal** also contains a number of variables that can be changed for modifying the operation of the learning package. The different variables and their effects are described in table 5.4.

The [public **Perception** **getPerception(StateInstance state)**] method is required to generate perceptions for the learning algorithms to use. **Perceptions** are used such that only a subset of the whole state is used for learning purposes. This is because, if the entire state is used, then redundant information in the state would hinder the learning performance.

The reward function [public double **rewardFunction(Perception before, Action action, Perception after)**] informs the agent with what is required to learn. Choosing a correct reward function is very important as it has a great effect on learning. Sometimes, it is also required to try different reward functions, and find the function that yields best results.

The [public int **getGoalStatus(StateInstance state)**] method returns a value that indicates the status of the **LearningGoal**. It returns **ACTIVE** while the **LearningGoal** is active and **SUCCEEDED**, **FAILED** or **IMPOSSIBLE** as needed. If any of the latter three values are returned, the learning algorithm finishes operating for that **LearningGoal**.

#### 5.3.4.2 Active Learning: Starting from Scratch

RL algorithms split up time into learning episodes, and evaluate the steps taken in the episode. In the *CHRIS* framework the steps within an episode are defined as the actions taken. The episode begins when the target is activated and it ends when the target finishes.

Active learning is realized by traditional RL algorithms. It involves handling a **Target** that has an attached **LearningGoal**, the agent then activates the learning algorithm specified in the **LearningGoal**, and executes different actions while observing the results of the actions.

```

1 public final int LOG_AGENT = 0;
2 public final int LOG_PLANS = 1;
3 public final int LOG_BELIEFS = 2;
4 public final int LOG_OBSERVATION = 3;
5 public final int LOG_ORIENTATION = 4;
6 public final int LOG_DECISION = 5;
7 public final int LOG_ACTION = 6;
8 public final int LOG_LEARNING = 7;

```

Code Listing 5.8: Logging Levels of CHRIS Framework

Learning is achieved by the algorithm finding an optimal value function. That is, a function that translates a particular state to the best action that is to be taken in that state.

#### 5.3.4.3 Passive Learning: Observing Pre-Defined Behaviours

Passive learning is a feature that has been implemented specifically in the *CHRIS* framework but was inspired by Dixon [32]. It allows a policy to be replaced by a plan and hence, learn the behaviour of that plan with respect to the given LearningGoal.

**SkillControl** is another feature that allows the agent to switch from passive learning to active learning. It is enabled by setting the **skillControl=true** in the LearningGoal. SkillControl uses two thresholds for changing learning algorithms. The **activeThreshold** causes an agent to switch from passive learning to active learning. The default value of the threshold is 80%. In other words, when the confidence reaches 80% during passive learning it switches to active learning. When this happens the agent's behaviour changes as it explores new actions selected when the RL algorithm explores.

#### 5.3.5 Logging How Decisions Are Made

The logging package was introduced to manage logging performed by the framework. The main purpose of this package is to turn on and off logging information produced by other packages in order to aid with debugging.

Customization of logging is achieved with only a single **String** of ones and zeros. Each character in the string corresponds to a specific package (also called logging level). The index of the character identifies the logging level while the character itself is used to turn logging on and off. If the character is a '1' logging is turned on, if the character is anything else, logging is turned off. The index of logging levels is defined as shown in code listing 5.8.

**DebugLog** provides the default implementation of logging. It simply prints out all entries to the screen. This may however, be changed by defining a class that extends **DebugLog**. Some possible implementations are writing the log to a file on the disk, or sending messages through a network connection. **DebugLog** may be extended as shown in code listing 5.9.

### 5.4 Example Agents

This section briefly describes two example learning agents developed using the *CHRIS* framework.

```

2  public class MyDebugLog extends DebugLog{
    /** constructor */
4     public MyDebugLog(String strLogLevels){
        super(strLogLevels);
    }
6
    /** Overwrites default logging behaviour */
8     public void writeToLog(String logEntry){
        //code that writes logEntry to the log
10    }
}

```

Code Listing 5.9: Extending the CHRIS Framework DebugLog

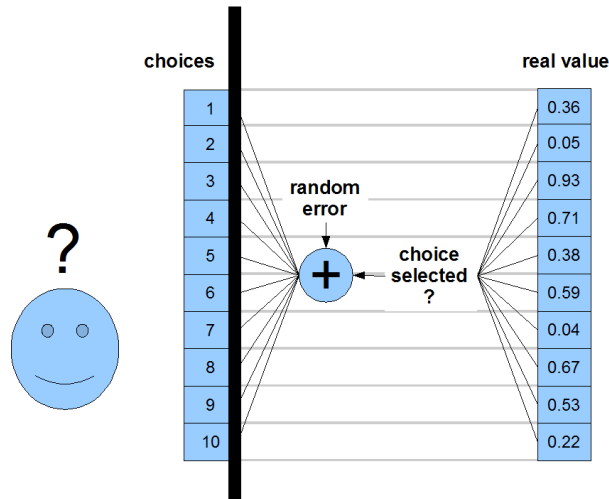


Figure 5.9: 10 Armed Bandit RL Problem

### 5.4.1 The 10-Armed Bandit

This section describes a classic RL problem that is solved using the CHRIS framework. This particular problem was chosen because it is able to demonstrate the *CHRIS* framework with a relatively small amount of code required. The source code for this problem is short enough to entirely include in appendix C of this thesis and is described step-by-step.

The problem presented is the *10-armed bandit* problem. It involves the agent being faced repeatedly with a choice of 10 distinct actions. After selecting the action (called a play), the agent receives a numerical reward. The *objective* for the agent is to *maximize* its rewards over 10,000 plays. If the agent could directly measure the value of each action, this would be trivial problem because it would only need to sample each action once, and then always select the action with the highest value. This problem however incorporates a *gaussian random error* to the value returned to the agent. Hence, if the agent performed the same action twice, it would obtain two different values for that action.

The only way for the agent to maximize its rewards, is by focusing its plays on the best actions available to it. Initially, it is not possible for the agent to know which are the best actions. Hence, it can only choose actions randomly. However, after all actions are tried enough times, it becomes possible for the agent to learn how good each action really is. The 10-armed bandit problem is illustrated in figure 5.9.



#### 5.4.1.1 The TenChoiceTestbed Environment

The `TenChoiceTestbed` environment is very simple, the source code is shown in code listing C.9. It contains two variables, firstly, a reference to the `Bandit` agent (line 3) and secondly an array of 10 integers called `qStar`, that represent the real values of the ten actions (line 4). The array is initialized to random decimal values in the range of 0.0 to 2.0 in the constructor (lines 6,10). The `step()` method is the one that is called by the agent when it makes a choice. Line 13 calculates the value given to the agent, by adding the real value with a Gaussian random error. Line 14 then notifies the agent using the `activateDecisionProcess()` method. Finally, the `init()` method (lines 17-19) simply initializes the internal reference of a `LearningAgent` with an instance of `Bandit`. This method is *expected* to be executed *before* attempting any choices. Otherwise an error will occur as the `step()` method assumes that the `LearningAgent` reference has been initialized.

The beliefs of the bandit have been designed to reflect the latest value that the agent has received from its environment. They are defined using a single beliefset called `BanditState`, as shown in code listing C.5. It implements the `State` interface, this forces it to define the methods that are required by the framework in order to operate correctly. The beliefset contains a single `#value` declaration and an associated query to access the value (Lines 3,4). The fact that there is no `#key` field means that there can only ever be a single belief in this beliefset. The function `getCurrentValue()` (lines 6-12) is a standard function query that simply allows normal Java objects to query the value of the beliefset.

The framework also requires two additional methods to be implemented. Firstly, the `getInstance()` method (lines 23-25) creates a `BanditStateInstance` object using the value of the state. Secondly, the `stateChange()` method (lines 19-21) defines how to notify the framework when the state has been updated with new information. In this case, it is done by posting a `StateChange` event, that is automatically handled by the framework. Finally, the `moddb()` method (lines 14-16) is a definition of a beliefset call-back method. It is automatically executed when the beliefset is updated, the method simply executes the `stateChange()` method.

The `BanditStateInstance` is defined in code listing C.6, is meant to capture an instance of the bandit's state. Therefore, it simply contains a single decimal (called `double` in Java) value. The constructor (lines 3-7) simply copies the value received as a parameter to the internal variable. The framework requires the definition of two methods when it comes to `StateInstances`. Firstly, the `equivalent()` method (lines 9-12) is meant to test whether another `BanditStateInstance` is equal to this one. Secondly, the `hashCode()` method (lines 14-16) generates a unique integer value for this `StateInstance` that is based on internal variables.

The `BanditPerception` definition is very similar to `BanditStateInstance` and is defined in code listing C.7. There is however one key difference that relates to the implementation of the `equivalent()` method (lines 9-11). The `StateInstance` implementation returns `true` only if the other `StateInstance` has the exact same value. Conversely, this implementation *always* returns `true`. The reason for this lies with the fact that the agent always has the same actions available to it, and the environment has no influence of the actions themselves.

Therefore, it can be assumed that the state is the same both before and after making a choice. This is done in order for the framework to *generalize* all states as the same perception.

The ability to act in the environment is defined in two parts. The **BanditAction** is shown in code listing C.3. It contains the information needed by the agent in order to execute the action. In this case it is the choice made by the agent (line 3). The constructor assigns a unique name to the action as required by the framework (line 6). The **equivalent()** and **hashCode()** methods are basically implemented in the same way as with **BanditStateInstance**. The **BanditPlan** is a plan that is executed in order to perform the action. It handles an **ActionEvent** (line 4) but it declares that it is only relevant for execution if the **ActionEvent** contains a **BanditAction**. The body of the plan extracts the value of the choice from the **BanditAction** (line 13) and executes the action (line 14). There are also two additional statements (lines 12 & 15) that notify the framework when the action has started and completed execution. This is performed in order for the framework to be able to operate with actions that take long periods of time to complete, in which time several updates to the state may have occurred.

#### 5.4.1.2 The Learning Problem

A key point to understand about the bandit problem is that rewards are directly proportional to the real value of the action. Conceivably, it makes sense that if all different actions are sampled enough times, then overall, the actions with a higher hidden value would yield better rewards. The question now arises as to why to use learning for this problem as it is simple enough to create an agent that keeps an average of rewards received, and then influence its actions accordingly. In-fact, this problem is only a special case of the full RL problem because the agent always finds itself in the same state after a choice. In addition, it is simple to implement and describe, in terms that are relevant to larger problems.

The *CHRIS* framework is able to encapsulate an entire learning problem into a **LearningGoal**. The **BanditLearningGoal** is shown in code listing C.2. **LearningGoals** are designed to be extended in order to define all required information about the learning problem. The constructor of **BanditLearningGoal** declares that the framework should use **QLearning** for the RL algorithm and the **EGreedy** action selection policy (line 5). Lines 6-12 are optional, but they have been included in order to illustrate how different aspects of the **LearningGoal** have been changed in the results presented further on in this section.

The **getPerception()** method (lines 15-18) is required by the framework because it is used by the RL algorithm to generate perceptions. It converts a **StateInstance** that is globally available, to a **Perception** that is only applicable for this **LearningGoal**. This is done in order to filter out all information that may confuse the learning process. The **rewardFunction()** method (lines 22-25) generates a reward for the agent based the perception of the world before it took the action, and the perception of the world after the action. For the bandit, the implementation is very simple because the reward is the same as the value returned by the environment.

The **RandomSelectionPlan** plan is defined specifically for passive learning, it is shown in code listing C.8. It provides a default plan for the agent to execute during passive learning. The plan begins using JACK keywords (lines 3-9) to obtain references to components used in the body of the plan. The body begins by obtaining a reference of the **LearningGoal** that is

observing this plan (line 16), it then uses the agent to obtain the actions available to it (line 17). It then chooses a random number (line 19), and notifies the learning algorithm of the chosen action (line 20). Finally, it waits for the learning algorithm to take the selected action (lines 21 & 22) before proceeding to post an **ActionEvent** with the chosen **BanditAction** (line 23).

#### 5.4.1.3 The Bandit Agent

The **BanditAgent** brings all of the previously described components together. The definition of the agent also begins with a number of JACK keywords that are required for the agent to function correctly. The constructor of **BanditAgent** has a parameter that defines the number of choices that it is required to perform (line 17). It also initializes the testbed (line 19), it creates a list of **BanditActions** that provide the choices available to the agent (lines 21-23), initializes the agent's state (line 24) and finally the agent's **LearningGoal** (line 25).

The **activateDecisionProcess()** method (lines 28-30) simply posts a **Sensation** with the value received from the environment. The **updateWorldState()** method (lines 32-34) updates the agent's state with the new value. The **getActionsAvailable()** method (lines 36-38) simply returns the static list of actions that was initialized in the constructor. The **choose()** method (lines 45-57) is called by **BanditPlan** in order to execute the required choice. Finally, the **findBestAction()** method (lines 40-43) is the one that is executed by the parent program that created the agent (see code listing C.10) in order to get it to start playing.

#### 5.4.1.4 Simulation Results

The Bandit agent was executed for a small number of runs. The learning algorithm and the action selection policy were not changed for the different runs. Additionally, an initial random seed was entered for all random variables. This was done for two reasons, firstly, to allow the results to be easily repeatable and secondly, to allow the comparison of the performance of different learning options without having to worry about differences caused by the randomness of different runs.

Three specific variables were changed in order to investigate the differences in performance. The first variable is **skillControl** which effectively acts as a master switch for the following two variables. The second variable is **learningTypeControl** which when enabled allows switching between active and passive learning. The third variable is **explorationControl** which when enabled causes the exploration rate of the EGreedy policy to be variable based on the confidence of the **LearningGoal**.

The first two runs were both executed with **learningType** being active. However, the first run had **skillControl** disabled, this means that the entire run was executed in active mode, and the exploration rate was constant at 0.1. The second run was executed with **skillControl** enabled, but only **explorationControl** enabled which means that the exploration rate was variable between 0.05 and 0.5.

The last three runs were executed with learning type being passive. The third run was executed with **skillControl** disabled. This means that the learning stage could only observe but not influence the agent's behaviour. The forth and fifth runs were both executed with

Run	learningType	skillControl	learningTypeControl	explorationControl
1	LEARNING_ACTIVE	false	N/A	N/A
2	LEARNING_ACTIVE	true	false	true
3	LEARNING_PASSIVE	false	N/A	N/A
4	LEARNING_PASSIVE	true	true	true
5	LEARNING_PASSIVE	true	true	false

Table 5.5: Learning Options for Bandit Runs

`skillControl` enabled. Except, the latter had `explorationControl` disabled. The options for each run is summarized in table 5.5.

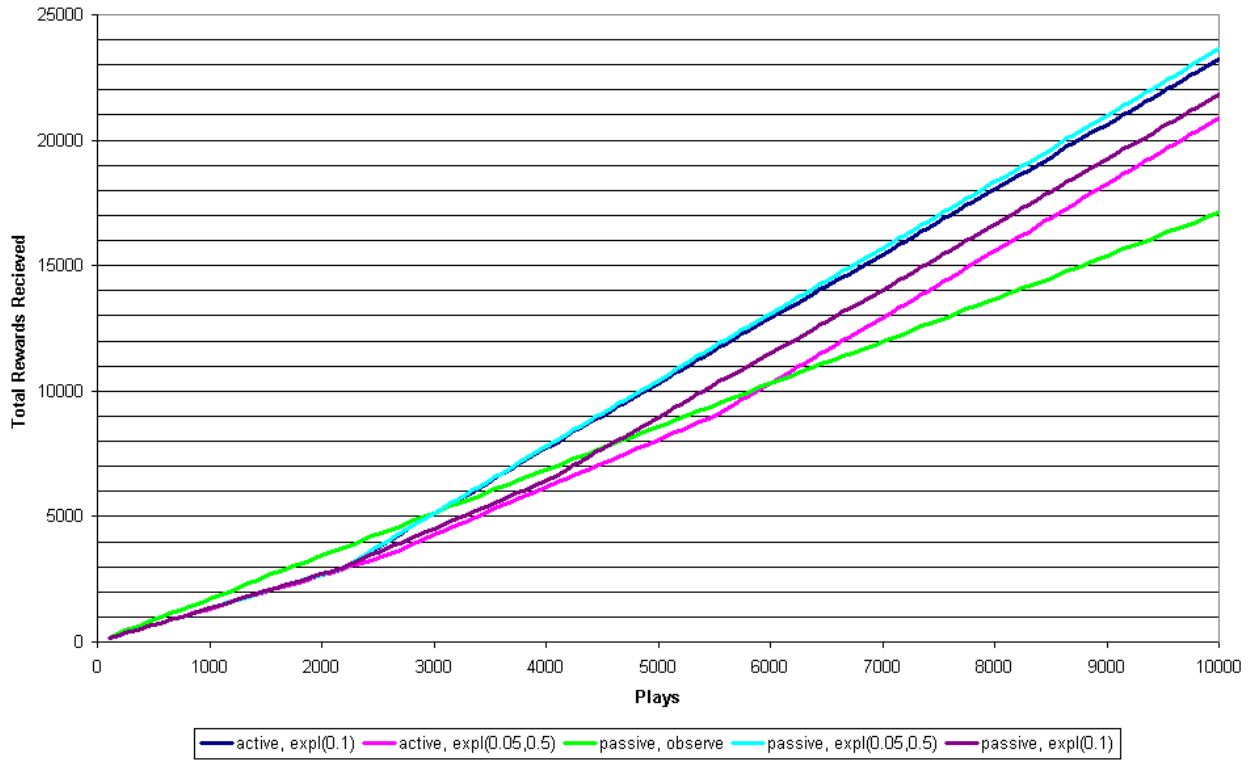
The rewards received by Bandit for the different runs are shown in figure 5.10a. It indicates that active learning with a constant exploration rate and passive learning with a variable exploration rate obtain very similar rewards throughout their runs. Passive learning with a constant exploration rate also performs better than active learning with a variable exploration rate. The worst performance is indicated as passive learning with `skillControl` disabled. This is to be expected because, in that case, the learning stage only observes the agent it does not influence its actions. The default JACK plan also obtains rewards because all actions are given a reward. The total rewards are however nowhere near as when the learning stage is used.

The second plot shown in figure 5.10b is much more interesting because it illustrates the performance of each run based on the number of times that the optimal action was selected. The optimal action is the action with the best value that is hidden from the agent. The run with passive learning and variable exploration performs the best and it had selected the optimal action 71% at the end of the run. The run with active learning and static exploration performs at 60%. Both runs with a variable exploration take a lot longer to find the optimal action (due to the larger initial exploration or 0.5). Predictably, the run with passive observation performs no better than 10%. This is expected because in that run the agent chooses actions totally randomly and should not choose one of the ten actions more than 10% of the time.

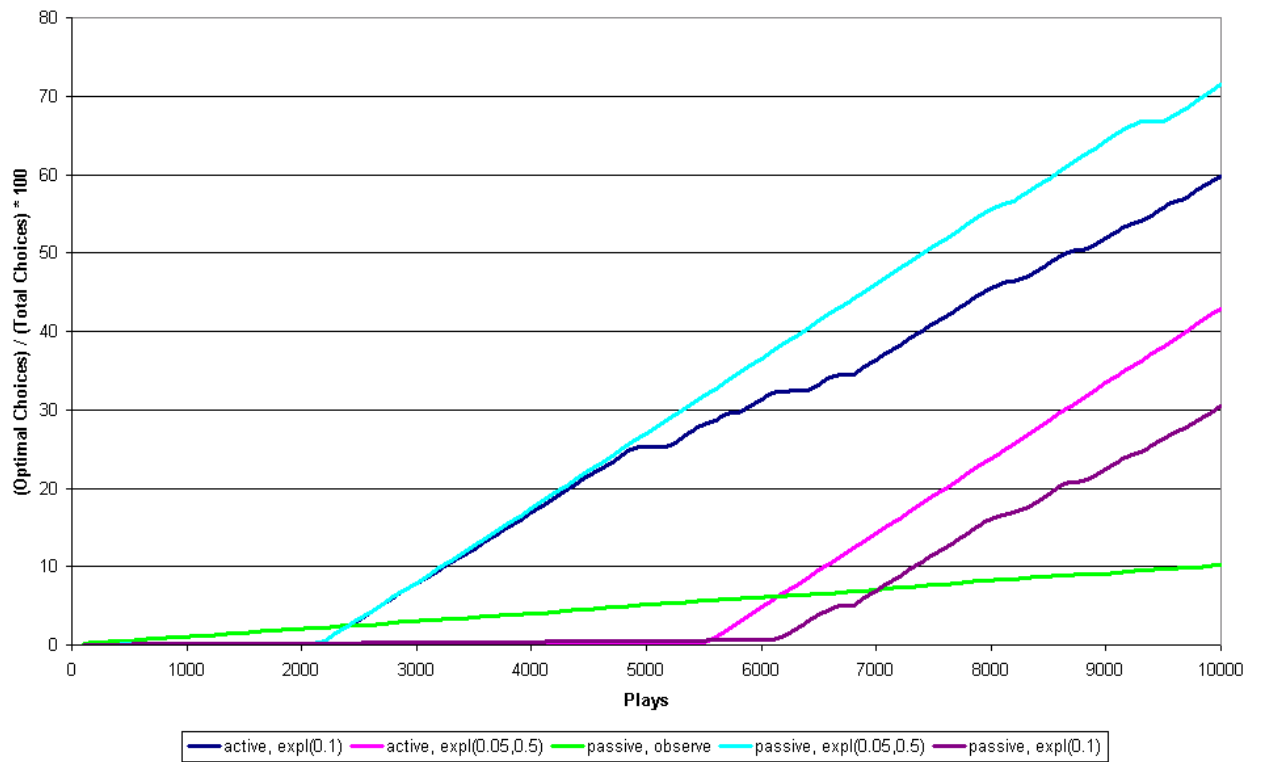
## 5.4.2 The Tic-Tac-Toe Player

This section describes a slightly more complex example where the agent learns how to play Tic-Tac-Toe and win against a computerized opponent. When playing Tic-Tac-Toe if both players are experts the game always ends in a draw. In other cases the first player has a greater chance of winning because it can make 5 moves as compared to the second player only having 4 moves [81]. For this reason the system was designed such that the agent always makes the first move, which is hard-coded as random.

The actions available to the agent are variable and are based upon the moves available to the agent for a particular state. The Sarsa RL algorithm and the EGreedy action selection policy were selected. The reward function is based on what happens after the agent takes an action. It returns -50 if the action results in a loss, +50 if the action results in a win, -20 if the action results in a draw, and -1 if the game is not finished. This instructs the agent to avoid losing and drawing while and trying to win in the quickest way possible. A simple plan was also implemented for operation during passive learning, it contains the following algorithm:



(a) Total Rewards Received



(b) Percent Optimal Choices

Figure 5.10: Results Obtained With Bandit

1. If an action exists that will cause a win, take it.
2. If in danger of losing, block it.
3. Otherwise take random action.

The required components for both active and passive learning were implemented, and a run of 10000 plays was made for each of them. The EGreedy exploration percentage was manually decreased from 50% at 0 plays down to 0% at 5000 plays. This was done to slowly converge the value function to a solution. Hence, after the 5000 plays the performance observed is caused by pure exploitation of learned knowledge, no exploration. This provides a indication of how good the learned solution is.

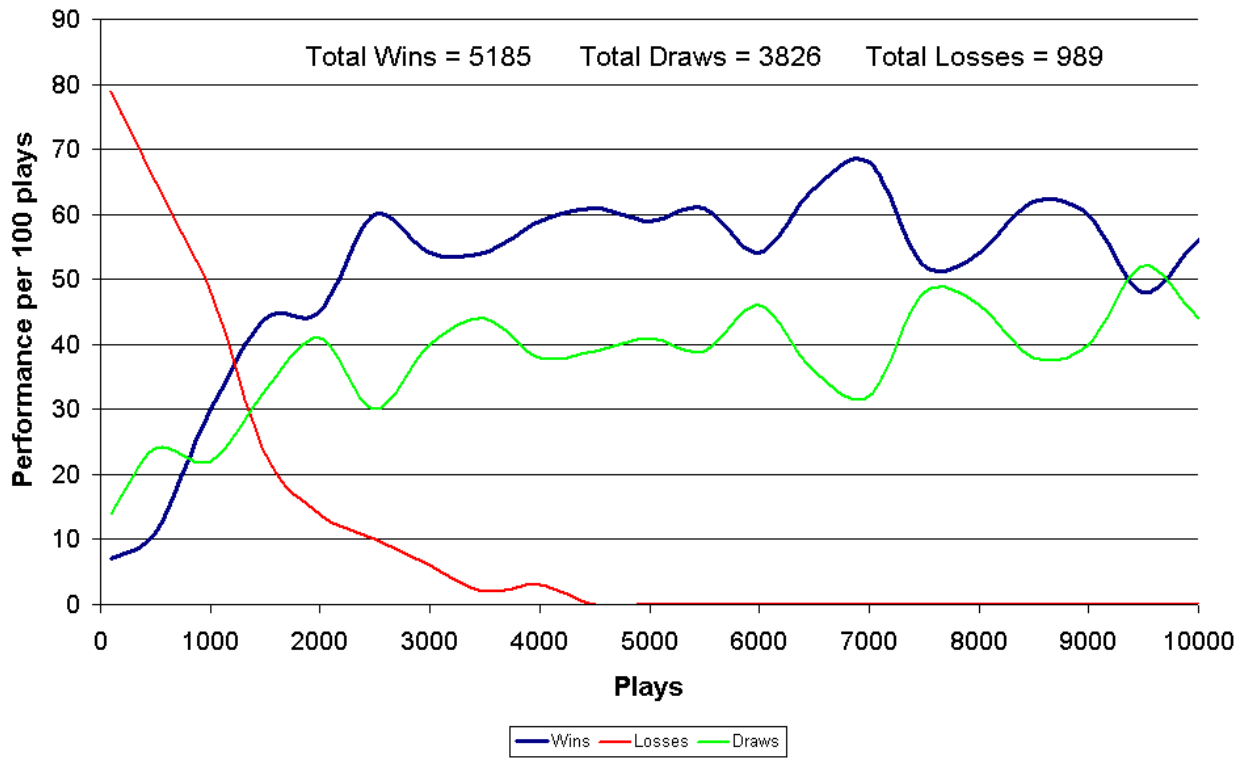
Figure 5.11a shows the performance of the agent for active learning. The performance is an indication of how many games the agent has won/lost/drawn in blocks of 100 plays. It begins with a very low performance as it initially has no knowledge about how to play. It quickly improves and wins soon outperform the losses at 1300 plays, losses subsequently drop to less than 10 at 2500 plays. Finally, at 4500 plays losses drop to zero and remain there until the end. The wins fluctuate at around 55 while draws fluctuate at around 45. This fluctuation is caused by the forced randomness of the agent's first move in each game. This was proven by disabling the random first move, the agent learned to win 100% of the time. However, in this case the agent always played the same initial move, which resulted in exactly the same game being repeated over and over.

Figure 5.11b shows the performance of the agent for passive learning. The initial performance is much better with less than 10 losses, approximately 20 wins and 70 draws. It is clear that the behavior dictated by the plan favors drawing. At 1000 plays learning is switched from passive to active at which point the agent starts exploring new actions, subsequently losses and wins increase while draws decrease. The value function finally converges to a similar performance than the active learning. An important point is that wins are always more than losses (albeit by a fraction at 1300 plays) and losses never reach 40. This illustrates that passive learning allows for better initial behavior while converging to similar solutions in the end. The down point is the effort required to create the additional pre-defined plan which is not required for active learning.

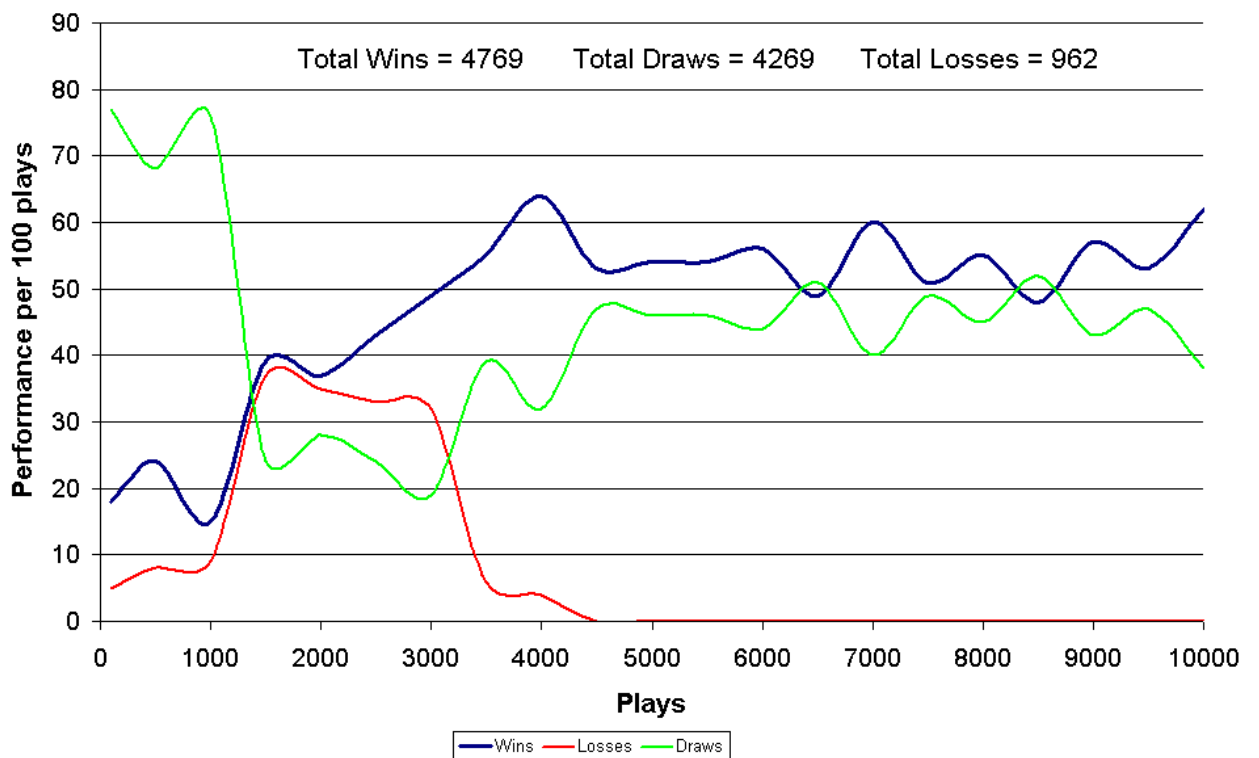
## 5.5 Discussion

The decision ladder, the OODA loop and the BDI model described in chapter 3 are complementary and can be fused together to yield a new, hybrid and more detailed conceptual reasoning model. The general structure of the OODA loop is retained, by dividing the decision ladder and placing appropriate components in each of the four stages. The key components of BDI reasoning are also apparent within the new hybrid model.

The ability to learn can be achieved via appropriately interfacing an additional learning stage in parallel to the main decision loop. It captures an instance of the state of the environment, and generalizes it into a perception. It then uses reinforcement learning to learn how to map perceptions to actions such that rewards are maximized. The learning stage effectively replaces the “implicit guidance” link indicated by the OODA loop between the orientation



(a) Active Learning, exploration(0.5,0)



(b) Active Learning, exploration(0.5,0)

Figure 5.11: Learning Agent Performance when Playing Tic-Tac-Toe

stage and the action stage.

The Cognitive Hybrid Reasoning Intelligent Agent System is an implementation of the conceptual model as an extension to the JACK agent development platform. It provides a framework to create learning agents without the need to worry about how the internal learning algorithms operate. Two simple learning agents have also been developed in order to demonstrate the *CHRIS* framework in operation. The first agent solves the classic 10-armed bandit problem. The second agent is a slightly more complex example, where it learns how to play Tic-Tac-Toe and win against a computerized opponent.

This chapter has presented the primary contribution of this thesis, it reveals a new agent hybrid reasoning model and the associated implementation framework. The following chapter illustrates how these new concepts apply to an agent-oriented development design methodology.



# Chapter 6

## Agent-Oriented and Learning Design

This chapter extends a well-known agent-oriented design methodology in order to include additional features introduced by the *CHRIS* framework. It then provides a design for a learning-agent system which strives to win a game of Capture The Flag in Unreal Tournament.

### 6.1 Introduction

The previous chapter described a new conceptual model of how humans perform reasoning, the advantage of the new model lies in the extra detail available in the reasoning process. A software framework called *CHRIS* was then described that implements this model as an extension to the JACK agent development platform. This chapter describes the design methodology used to develop agents with the *CHRIS* framework, it follows the *Prometheus* agent-oriented design methodology but introduces a number of additional components to accommodate the *CHRIS* framework. The design of a multi-agent system that operates in a complex simulation environment is then presented.

Section 6.1.1 firstly provides an overview of the Prometheus agent-oriented design methodology and illustrates what new design artifacts are required in order to accommodate the *CHRIS* framework. Section 6.2 describes the general specification of the system being designed. Section 6.3 defines the agents in the system and how they interact. Section 6.4 describes a detailed internal structure of one of the agent types identified. A summary and final remarks are finally provided in section 6.6.

#### 6.1.1 Extending the Prometheus Methodology

The Prometheus agent-oriented methodology was developed by Padgham and Winikoff [83] at RMIT University, it is summarized in figure D.1. It extends the work originally performed by Wooldridge's Gaia methodology [117]. The agent design is divided into three major parts: System Specification, Architectural Design and Detailed Design. The advantages of using the Prometheus methodology for agent-oriented design are threefold.

1. Prometheus describes the methodology of the entire design life cycle.
2. Detailed and comprehensive documentation is provided for each section in the associated book [83].

3. A software program called the Prometheus Design Tool (PDT) [6] is under constant development in order to assist with different design components described in the methodology.

The Prometheus methodology is intended to be generic, this allows it to be used on design problems while also being applicable for different agent development platforms. Researchers can use the Prometheus methodology and PDT for the first two stages and then develop agents using a platform of their choice. The Prometheus authors however wanted to provide examples of development for the detailed design stage, and chose the JACK platform. The detailed design presented in the Prometheus book is therefore based on Beliefsets, Events and Plans.

The *CHRIS* framework introduces a number of new concepts that must also be taken into account in the design methodology. They are illustrated in the extension of the Prometheus diagram shown in figure D.2. The main differences are:

- The inclusion of learning problems in the agent specification stage.
- The inclusion of behaviour adaptation descriptors in the architectural stage.
- The inclusion of all *CHRIS* components in the detailed design stage.

By comparing figure D.2 to the original D.1, it can be seen that the extensions to the methodology not only take into account learning, but provide a lot more detail on the structure of the agent's reasoning process.

The following sections describe a partial design for an agent-based system required to control characters for a team of players in UT. The design deliberately does not delve into much depth, in order to keep within the scope of this thesis, the integration of learning in agent-oriented design.

## 6.2 System Specification

This section describes the artifacts and processes in the system specification phase of the Prometheus methodology. The aim of this section is to understand the problem of developing an agent system for playing UT and develop a clear and precise understanding of the system to be built. This borrows concepts heavily from the area of requirements engineering. Systems specification is broken down into four sub-components, description of the interface between the agent and the environment, specification of the system's goals and subgoals, development of scenarios, and finally, describing the learning problems to be solved by the agent system.

### 6.2.1 Interface Description

The agent-environment interface is realized using software created in the initial stages of this research called *UtJackInterface* (UtJI) [100]. It was implemented with the idea that the Gamebots-Javabots combination described in section 2.4.3 potentially allows any Java-based software to interact with UT. JACK is built on top of Java, and according to Agent Oriented Software [5] it provides all the functionality of Java along with agent-oriented extensions. It was therefore assumed that it would be relatively easy to have a JACK agent interact with UT.

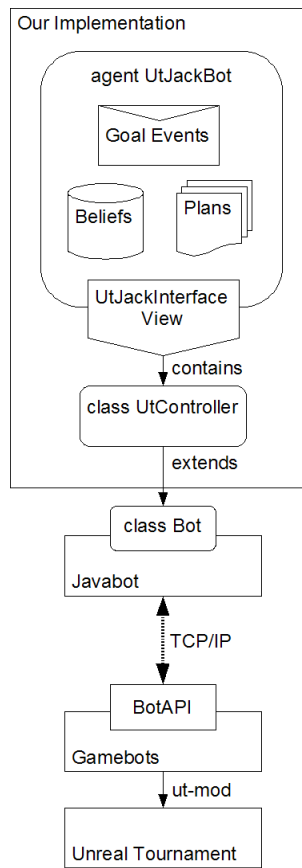


Figure 6.1: The Architecture of UtJackInterface [100]p746

```

1 //Gamebots Message
  GAM {PlayerScore {playerID val} {playerID val} ... } ...
3 //Parsed Key-Value pairs.
  key = "PlayerScore" + "@" + "playerID"
5 value = val

7 //Gamebots Message
  PATH {1 {NodeID location}} {2 {NodeID location}} ...
9 //Parsed Key-Value pairs.
  key = PathIndex
11 value = "NodeID" + "&" + "location"

```

Code Listing 6.1: Enhancing the Javabots Parser

The general structure of the UtJI is segmented into a Java extension and a JACK component as shown in figure 6.1.

The Java extension is based primarily on the **UtController** class, which is a subclass of the Javabot's **Bot** class. This means that it provides all the functionality of the Javabots package with a few extra features. Additionally, it fixes a message parsing error that exists in the Javabots message parser that does not parse GAM and PATH messages correctly. This is because they contain as a value a sub list of attribute-value pairs. Therefore the assumptions outlined for the parser (see section 2.4.3) do not hold for these messages. The new parser, parses these two message types as shown in code listing 6.1.

After parsing a message, a number of classes are used to convert data into usable information. For example, a NAV message has three attributes: **Id**, **Location** and **Reachable**. The corresponding **UtNav** class is used to capture NAV messages, it uses a **String** to capture the **Id**, a **UtCoordinate** object to capture the **Location** and a **boolean** to capture **Reachable**.

The heart of the JACK component is the `UtJackInterface view`<sup>1</sup> which wraps all functionality of the interface within a single construct. The view performs all bi-directional communication with UT. This involves updating relevant beliefsets, posting events for sensations and also providing methods for taking actions by sending messages back to the server.

### 6.2.1.1 Sensations

An agent senses the environment through messages received from the server, there are two types of messages. Firstly, synchronous messages correspond to what the agent *sees* in the environment, they provide the following information for a specific instant in time:

- Information about the state of player that the agent is controlling (e.g. health).
- Important game information such as the current score of the game.
- Other players that are in the field of view.
- Navigation points in the field of view.
- Movers (doors, elevators etc.) in the field of view.
- Domination control points in the field of view.
- Flags in the field of view.
- Inventory items in the field of view.

Each of the entities that an agent can see, have an associated beliefset that is continuously updated with information received in synchronous messages. For example, when another player is in field-of-sight, the agent's **players** beliefset is updated with information about this player. In other words, the agent believes that it is looking at another player when its beliefset says so. Synchronous messages are transmitted in blocks on a configurable interval, the default is every 10 milliseconds. The associated beliefsets can also be configured to generate events under some conditions. For example, a **FindHealthPack** goal event can be fired when the value of **Health** in the **Self** beliefset drops below 10%.

Asynchronous messages are received when some important events occur in UT. When an asynchronous message is received, the interface generates an associated JACK event with relevant data contained in the message. To understand this better, an agent “hears” when it receives an a **HEAR** message from the UT server, this causes an **UtJackAsyncHearNoise** event to be posted. Furthermore, it “feels” that it has been shot when it receives a **DAM** (damage) message from the UT server that causes a **UtJackAsyncDamage** event. In most cases sensory events require the agent to behave reactively. The types of asynchronous percepts that can be received are:

---

<sup>1</sup>Views are central concepts to JACK's data modeling capability. They provide the means to integrate different data sources such as JACK beliefsets, Java data structures and other software components into agents.

- Obtained an inventory item.
- Received a text message that was sent globally.
- Received a text message that was sent only to team members.
- Received a predefined UT command message.
- The agent's feet have entered another zone in UT.
- The agent entirely entered another zone.
- Changed the weapon type.
- Collided with a wall.
- In danger of falling off a ledge.
- Fell off a ledge.
- Bumped into someone.
- Heard someone pick up a weapon.
- Heard something making noise nearby.
- In danger of being hit by a projectile weapon.
- Another player has been killed.
- The player itself was killed.
- Received damage (was shot by someone).
- The agent successfully shot another player.
- A list of navigation nodes describing a path that leads to a requested location.
- A boolean reply that indicated whether a requested location is directly reachable.

The sheer number of sensations available to the agent as described in this section leads to the necessity of the agent having a very complex internal state constantly modeling the outside environment. Consequently an instance that captures the state in a particular moment in time is usually unique. This concept is easily understood by considering a single player walking around the UT world by itself. As it walks around, its state is always updated with its absolute (x,y,z) location represented in a decimal format of UT units. This means that an agent rarely occupies *exactly* the same location twice within an extended period of time. Even if the agent does end up occupying the exact same location, chances are that other variables in the state are different. For example, the agent may be looking in a different direction or it may be carrying a different weapon. In addition, when including other players in the world, it becomes extremely unlikely that two or more players in visual range happen to occupy exactly the same location. This seemingly infinite number of states that an agent may find itself in makes it very difficult to create behaviours that can accommodate every possible combination that may occur.

### 6.2.1.2 Actions

Agents take action in the world by transmitting commands back to the server. Most of the commands are instantaneous like `Jump`, some commands however operate over relatively long periods of time and are called persistent. For example, when a `RunTo` command is issued, the player will keep on running until it reaches its destination. Persistent commands can only be interrupted when some important events occur (e.g. being killed). Although persistence is desirable, it requires the agent to monitor the execution of persistent commands. For example,

if an agent has issued a command to **RunTo** from location A to location B, it cannot be assumed that it actually arrives to that location. This is because it may encounter a wall between where it started and where its destination is. In this example, the agent would keep pushing itself against the wall causing **WAL** sensations to be posted. The designer must therefore take into account the possibility of receiving such **WAL** messages, and include behaviours that cause the agent to try walking to a different destination. The commands available to the agent are:

- Telling the player whether to walk to run when moving between locations.
- Stop all movement (cancels persistent commands).
- Tell the player to Jump.
- Walk or run to a given destination. A destination may be specified by the ID of a navigation point or an absolute location coordinate.
- Perform a strafe by moving towards a destination while facing another location.
- Turn towards a particular target (like another player) or a specific location.
- Rotate the player's yaw, roll or pitch a specified number of degrees.
- Shoot a target.
- Stop shooting.
- Change the weapon held by the player.
- Ask the UT server whether a particular location is directly reachable by the player (i.e. there are no obstructions in between).
- Ask the UT server for a list of navigation nodes (a path) leading to particular location.
- Send a text message to team mates or all players present in the game.

By inspecting the above list of available actions, specifically the ability to walk, strafe, turn and rotate to any desired location, it is obvious that the agent again faces an infinite amount of actions at any moment in time.

## 6.2.2 System Goals

This section provides a listing of the hierarchy of goals that are required in order to achieve the desired system functionality. The top level goal is called **WinCTF** which corresponds to the overall desire to win unreal tournament in a CTF game. The subgoals of **WinCTF** divide the hierarchy into three main categories.

The **Survive** subgoal tries to keep agents alive as much as possible in the game through the use of their weapons and monitoring their health status. Additional reasoning is also required in order to balance between agents surviving longer, or sacrificing themselves for the benefit of gaining an advantage towards achieving **WinCTF**. The **Achieve** subgoal is directly aimed to achieving **WinCTF**. It involves the agents scoring points against the enemy, while also preventing the enemy from scoring. The **Explore** subgoal was introduced for two reasons. First, it is specifically required at the start of a game in order for the agents to search and locate the home and enemy flags. This information is required to instigate reasoning about the **Survive** and **Achieve** subgoals. Second, it is used for **Reconnaissance**, this means continuously processing

information received from the environment in order gain a better understanding of the simulated world. The improved understanding allows agents to better plan how to achieve their given goals. This goal hierarchy is illustrated graphically in figure 6.2 which was created using the PDT.

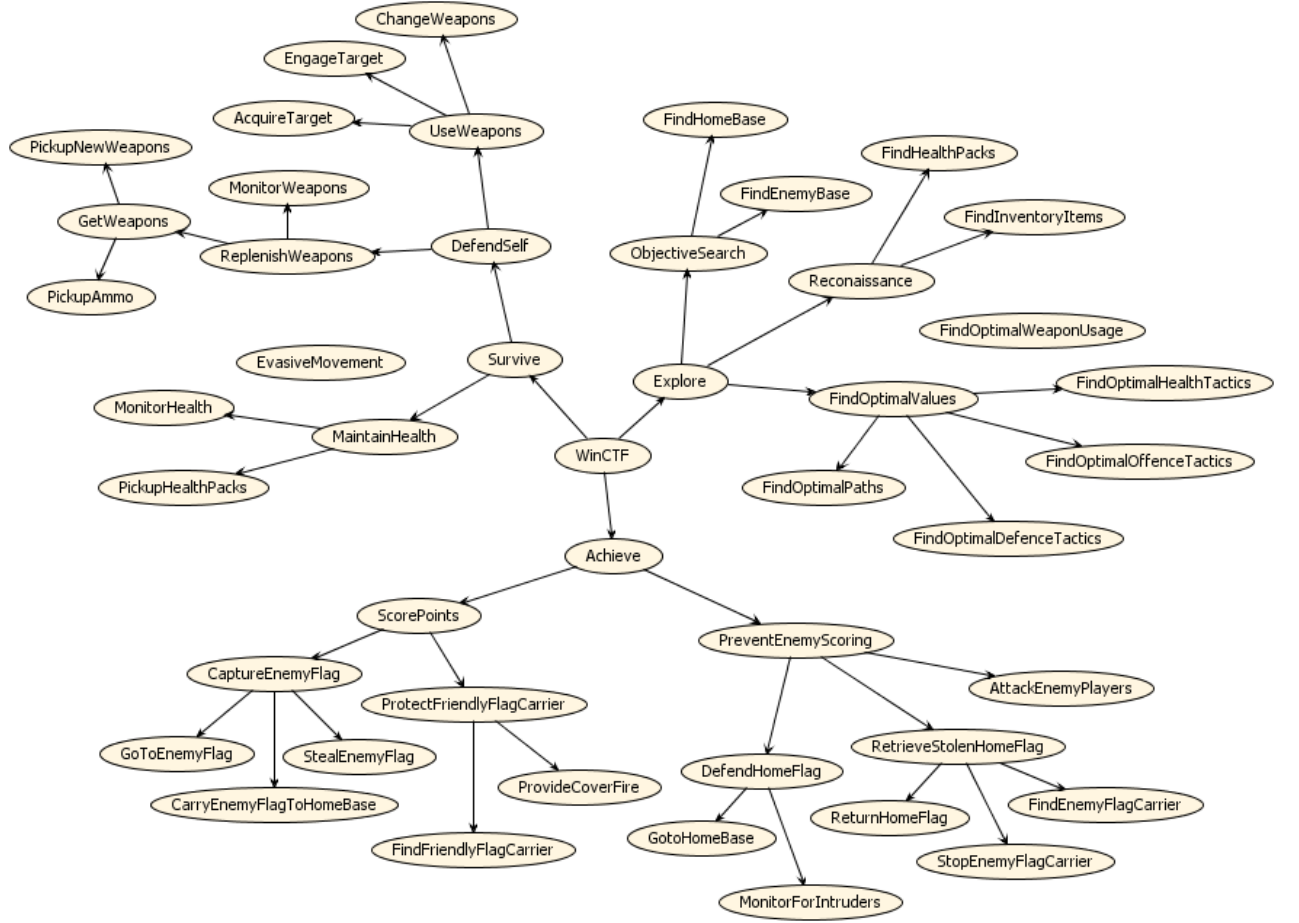


Figure 6.2: UnrealAgents System Goals

### 6.2.3 Functionalities

This section describes the different functionalities that are required in order to produce a successful solution. They are divided into three categories depending on their use. The first category encompasses *teaming* functionalities that involve tasks such as team formation and management. The second category encompasses *commanding* functionalities that require issuing orders to subordinate agents. These have the most responsibility because they are required to make tactical decisions and order their troops what to do as situations unfold. The third category encompasses a *troop* functionality that creates a link with UT and controls a player in the UT environment. Figure 6.4 shows the relevant functionality descriptors in the format defined by the Prometheus methodology.

The system uses four teaming functionalities. The first functionality is called **UnrealAgents** and is responsible for achieving the overall goal of winning a game of CTF. UnrealAgents encloses three subteams for realizing different required functionalities. The **UADefence** subteam is responsible for preventing the enemy team from scoring. This is mainly performed

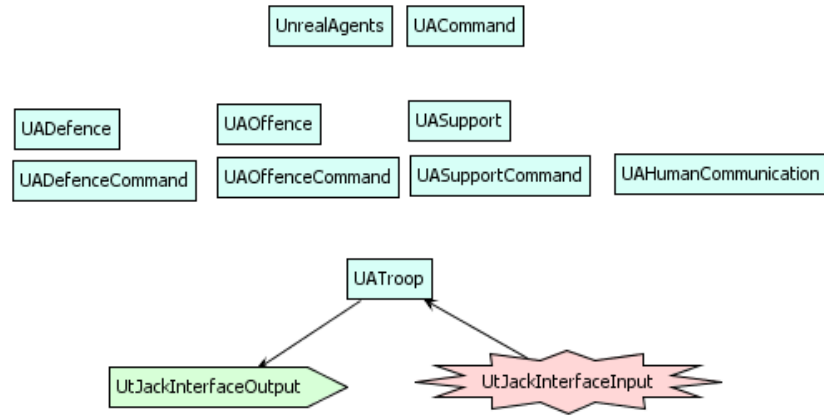


Figure 6.3: Graphical Summary of Functionalities

by defending the team flag. However, it also includes trying to recover the flag, when it has been captured before the enemy scores. The **UAOffence** subteam instead tries to score points against the other team. Finally, the **UASupport** team performs general reconnaissance and provides support to other teams when requested.

There are four commanding functionalities. **UACommand** is responsible for making tactical decisions about the whole team by managing subteams accordingly. **UADefenceCommand** is responsible for making tactical decisions about the deployment of the defence troops. **UAOffenceCommand** is responsible for scoring points against the other team. Finally, **UASupportCommand** provides support, and other useful information to the team.

There is only one type of troop functionality. It requires a much bigger number of capabilities as troops need to interface with UT and control a player within the game. Although the same troop functionality is used for all agents connecting to UT, troops belonging to different subteams are expected to follow different orders as per the objectives of their team.

## 6.2.4 Scenarios

Scenarios are used to describe sequences of events that the system will need to execute. They are primarily used to indicate the events, that instigate goals, as well as their corresponding subgoals. Some scenarios may also involve messages being sent between agents.

### 6.2.4.1 Win Scenario

The **Win** scenario is a system-level scenario for the entire **UnrealAgents** team. It describes what happens when the highest level goal **WinCTF** is activated by an external influence to the system (i.e. the User). The **WinCTF** goal simply activates the **Survive**, **Explore** and **Achieve** subgoals for the **UnrealAgents** team.



Name	UnrealAgents (UA)
Goals	2
Description	A team consisted of agents and possibly humans that work together towards winning a game of CTF in UT.
Percepts	N/A
Actions	TeamCreation
Messages	N/A
Information Used	TeamStatus, GameStatus
Information Produced	RoleChange
Interactions	N/A

Name	UAGoalCommand
Goals	2, 4.2, 3.1
Description	A team of players that assist the UA team when needed.
Percepts	None
Actions	None
Messages	Survive, ObjectiveSearch, Score-Points
Information Used	TeamStatus
Information Produced	N/A
Interactions	N/A

Name	UADefence
Goals	2, 3.2
Description	A team that is responsible for defending assets in UT.
Percepts	Gamebots VizClient
Actions	N/A
Messages	Survive, DefendAssets, Retrieve-HomeFlag
Information Used	TeamStatus
Information Produced	N/A
Interactions	N/A

Name	UASupportCommand
Goals	2, 4
Description	A team of players that assist the UA team when needed.
Percepts	None
Actions	None
Messages	Survive, ObjectiveSearch, InformationSearch, DefendAssets
Information Used	TeamStatus
Information Produced	N/A
Interactions	N/A

Name	UAGoalOffence
Goals	2, 4.2, 3.1
Description	A team of players that assist the UA team when needed.
Percepts	None
Actions	None
Messages	Survive, ObjectiveSearch, Score-Points
Information Used	TeamStatus
Information Produced	N/A
Interactions	N/A

Name	UADefenceTroop
Goals	2, 3.2
Description	A team that is responsible for defending assets in UT.
Percepts	Gamebots VizClient
Actions	N/A
Messages	Survive, DefendAssets, Retrieve-HomeFlag
Information Used	TeamStatus
Information Produced	N/A
Interactions	N/A

Name	UASupport
Goals	2, 4
Description	A team of players that assist the UA team when needed.
Percepts	None
Actions	None
Messages	Survive, ObjectiveSearch, InformationSearch, DefendAssets
Information Used	TeamStatus
Information Produced	N/A
Interactions	N/A

Name	UAGoalOffenceTroop
Goals	2, 4.2, 3.1
Description	A team of players that assist the UA team when needed.
Percepts	None
Actions	None
Messages	Survive, ObjectiveSearch, Score-Points
Information Used	TeamStatus
Information Produced	N/A
Interactions	N/A

Name	UACommand
Goals	1
Description	Responsible for tactical decisions for winning a game of CTF in UT.
Percepts	N/A
Actions	N/A
Messages	Survive, Explore, Achieve
Information Used	TeamStatus, GameStatus
Information Produced	N/A
Interactions	N/A

Name	UADefenceCommand
Goals	2, 3.2
Description	A team that is responsible for defending assets in UT.
Percepts	Gamebots VizClient
Actions	N/A
Messages	Survive, DefendAssets, Retrieve-HomeFlag
Information Used	TeamStatus
Information Produced	N/A
Interactions	N/A

Name	UASupportTroop
Goals	2, 4
Description	A team of players that assist the UA team when needed.
Percepts	None
Actions	None
Messages	Survive, ObjectiveSearch, InformationSearch, DefendAssets
Information Used	TeamStatus
Information Produced	N/A
Interactions	N/A

Figure 6.4: Functionally Descriptors

**INCIDENT:** WinCTF (*External*  $\rightarrow$  *UnrealAgents*)

**PARALLEL:** Execute in Parallel

- Survive (*UACommand*  $\rightarrow$  *UADefence*)
- Survive (*UACommand*  $\rightarrow$  *UAOffence*)
- Survive (*UACommand*  $\rightarrow$  *UASupport*)
- Explore (*UACommand*  $\rightarrow$  *UADefence*)
- Explore (*UACommand*  $\rightarrow$  *UAOffence*)
- Explore (*UACommand*  $\rightarrow$  *UASupport*)
- Achieve (*UACommand*  $\rightarrow$  *UADefence*)
- Achieve (*UACommand*  $\rightarrow$  *UAOffence*)
- Achieve (*UACommand*  $\rightarrow$  *UASupport*)

#### 6.2.4.2 Surviving Scenarios

Surviving scenarios are concerned with the agent being able to reasonably survive when operating against an enemy in UT. The higher level **Survive** goal simply activates the **DefendSelf** and **MaintainHealth** subgoals.

**INCIDENT:** Survive (*UACommand*  $\rightarrow$  *UnrealAgents*)

**PARALLEL:** Execute in Parallel

- DefendSelf (*UACommand*  $\rightarrow$  *UnrealAgents*)
- MaintainHealth (*UACommand*  $\rightarrow$  *UnrealAgents*)

The **DefenceSelf** scenario is concerned with the agent being able to defend itself using weapons against other enemies. This is done by posting the **UseWeapons** and **ReplenishWeapons** goals in parallel. This way agents are always ready to use their weapons while at the same time always monitoring the status of their weapons.

**INCIDENT:** DefendSelf (*UACommand*  $\rightarrow$  *UnrealAgents*)

**PARALLEL:** Execute in Parallel

- UseWeapons (*UACommand*  $\rightarrow$  *UnrealAgents*)
- ReplenishWeapons (*UACommand*  $\rightarrow$  *UnrealAgents*)

The **UseWeapons** scenario describes when weapons should be used. It has four subgoals, the first subgoal is activated most of the time as it monitors the agent's field of view for enemy targets. When an enemy is seen, the agent changes its weapons depending on the situation and engages the target. The entire scenario is wrapped in a loop, indicating that after an engagement has finished, the agent returns back to **AcquireTarget** mode.

**INCIDENT:** UseWeapons (*UACommand*  $\rightarrow$  *UnrealAgents*)

**LOOP:** Forever

- AcquireTarget (*UATroop*)
- PERCEPT: Enemy player in field of view
- ChangeWeapons (*UATroop*)
- EngageTarget (*UATroop*)

The **ReplenishWeapons** scenario describes when and how an agent tries to replenish its weapons. It begins with a goal that monitors the status of the weapons. When ammo is running low, or there is insufficient firepower to defeat the selected target, the agent tries looking for more weapons and ammo. This scenario is also wrapped in a loop that indicates that it is always executing while the agent is operating.

**INCIDENT:** ReplenishWeapons ( $UACommand \rightarrow UnrealAgents$ )

**LOOP:** Forever

- MonitorWeapons ( $UATroop$ )
- PERCEPT: Weapon ammo Low or Insufficient firepower for target
- GetWeapons ( $UACommand \rightarrow UATroop$ )

**PARALLEL:** Execute in Parallel.

- PickupNewWeapons ( $UATroop$ )
- PickupAmmo ( $UATroop$ )

The **MaintainHealth** scenario is concerned with keeping the agent's health to acceptable levels, such that it can provide more resistance when required. It involves monitoring the health of the agent, and looking for health packs when required.

**INCIDENT:** MaintainHealth ( $UACommand \rightarrow UnrealAgents$ )

**LOOP:** Forever

- MonitorHealth ( $UATroop$ )
- PERCEPT: Health Low
- PermissionLookForHealth ( $UATroop \rightarrow UACommand$ )
- LookForHealth ( $UACommand \rightarrow UATroop$ )
- PickUpHealthPacks ( $UATroop$ )

#### 6.2.4.3 Exploring Scenarios

Exploring scenarios are concerned with the ability to effectively explore the UT world. There are three subgoals for exploring. Firstly, **ObjectiveSearch** is concerned with looking for important entities in the UT world, such as the home and the enemy flags. This is because without this information the **Achieve** subgoals have no basis for execution. Secondly, the **Reconnaissance** subgoal is concerned with trying to find as much information about the world as possible. This is useful for uncovering the locations of health packs, powerups, powerful weapons and ammo, such that there is no need to search for them when they are needed.

The scenario shown below illustrates that **ObjectiveSearch** is posted separately to each of the three subteams. This is because each team has a different scenario for **ObjectiveSearch**. After **ObjectiveSearch** has completed for all three subteams, then **Reconnaissance** is posted to **UASupport** for exploring all unknown areas. Followed by, **FindOptimalValues** to activate learning goals.

**INCIDENT:** Explore ( $UACommand \rightarrow UnrealAgents$ )

**PARALLEL:** Execute in Parallel

- ObjectiveSearch ( $UACommand \rightarrow UAOffence$ )
- ObjectiveSearch ( $UACommand \rightarrow UADefence$ )
- ObjectiveSearch ( $UACommand \rightarrow UASupport$ )
- Reconnaissance ( $UACommand \rightarrow UASupport$ )
- FindOptimalValues ( $UACommand \rightarrow UASupport$ )

The following three scenarios describe what happens when the different subteams are tasked with **ObjectiveSearch**, they each posted a different subgoal.

**INCIDENT:** **ObjectiveSearch** ( $UACommand \rightarrow UAOffence$ )

- **FindEnemyBase** ( $UAOffenceCommand \rightarrow UAOffenceTroop$ )
- **POST MESSAGE:** **EnemyFlagFound**

**INCIDENT:** **ObjectiveSearch** ( $UACommand \rightarrow UADefence$ )

- **FindHomeBase** ( $UADefenceCommand \rightarrow UADefenceTroop$ )
- **POST MESSAGE:** **HomeFlagFound**

**INCIDENT:** **ObjectiveSearch** ( $UACommand \rightarrow UASupport$ )

- **FindHomeBase** ( $UASupportCommand \rightarrow UASupportTroop$ )
- **FindEnemyBase** ( $UAOffenceCommand \rightarrow UASupportTroop$ )

The **Reconnaissance** subgoal is used for searching the UT world in order to find all health packs and inventory items, this is used for agents to make better decisions.

**INCIDENT:** **Reconnaissance** ( $UACommand \rightarrow UASupport$ )

**PARALLEL:** Execute in Parallel

- **FindHealthPacks** ( $UASupportCommand \rightarrow UASupportTroop$ )
- **FindInventoryItems** ( $UASupportCommand \rightarrow UASupportTroop$ )

The **FindOptimalValues** scenario captures the learning required to be performed by the system. Agents are required to learn such that they improve their playing performance and adapt to other team tactics. Five parallel learning processes have been identified: improving movement, finding health, using weapons, defence and offence deployments and strategies.

**INCIDENT:** **FindOptimalValues** ( $UACommand \rightarrow UASupport$ )

**PARALLEL:** Execute in Parallel

- **FindOptimalWeaponTactics** ( $UASupportCommand \rightarrow UASupportTroop$ )
- **FindOptimalHealthTactics** ( $UASupportCommand \rightarrow UASupportTroop$ )
- **FindOptimalOffenceTactics** ( $UASupportCommand \rightarrow UASupportTroop$ )
- **FindOptimalDefenceTactics** ( $UASupportCommand \rightarrow UASupportTroop$ )
- **FindOptimalMovementTactics** ( $UASupportCommand \rightarrow UASupportTroop$ )

#### 6.2.4.4 Achieving Scenarios

**Achieve** is a goal that is posted to the entire **UnrealAgents** team, however only the defence and offence teams are involved in trying to satisfy it. The offence team satisfies it by scoring points, while the defence team satisfies it by preventing the enemy from scoring points.

**INCIDENT:** **Achieve** ( $UACommand \rightarrow UnrealAgents$ )

- **ScorePoints** ( $UACommand \rightarrow UAOffence$ )
- **PreventEnemyScoring** ( $UACommand \rightarrow UADefence$ )

The **ScorePoints** scenario is only applicable for the **UAOffence** team. It is concerned with scoring points in the game, this can only be done by capturing the enemy flag. After the enemy flag is captured by a teammate, then all other offensive troops protect the flag carrier from being killed by the enemy.

**INCIDENT:** ScorePoints ( $UACommand \rightarrow UAOffence$ )

- CaptureEnemyFlag ( $UACommand \rightarrow UAOffenceTroop$ )
- PERCEPT: Enemy Flag Captured.
- ProtectFriendlyFlagCarrier ( $UACommand \rightarrow UAOffenceTroop$ )

The **PreventEnemyScoring** scenario is applicable to the **UADefence** team because it is concerned with preventing the enemy from scoring. This is achieved by protecting the home flag from being captured. If however, the home flag does become captured, then the defence team's priority becomes going after the enemy flag carrier, and returning the home flag.

**INCIDENT:** PreventEnemyScoring ( $UACommand \rightarrow UADefence$ )

**PARALLEL:** Execute in Parallel.

- DefendHomeFlag ( $UACommand \rightarrow UADefence$ )
  - GoToHomeBase ( $UADefence \rightarrow UADefenceTroop$ )
  - MonitorForIntruders ( $UADefence \rightarrow UADefenceTroop$ )
  - AttackIntruders ( $UADefence \rightarrow UADefenceTroop$ )

**PERCEPT:** HomeFlagTaken ( $UADefence$ )

- RetrieveStolenHomeFlag ( $UADefence \rightarrow UADefenceTroop$ )
  - FindEnemyFlagCarrier ( $UADefence \rightarrow UADefenceTroop$ )
  - StopEnemyFlagCarrier ( $UADefence \rightarrow UADefenceTroop$ )
  - ReturnHomeFlag ( $UADefence \rightarrow UADefenceTroop$ )

## 6.2.5 Learning Problems

This section describes the additional component that involves describing what learning abilities the system should exhibit. At this stage only a general description of what is expected to be learned is required. For this system, four learning problems have been identified. They are concerned with improving the performance of the system after playing many times.

### 6.2.5.1 Weapons Learning

This learning problem is concerned with using weapons against other agents. It involves learning which weapons are the best ones to use in different situations. This is an important feature because there are many factors to consider in order to choose the correct weapon.

The different characteristics of weapons are described in appendix F. Some weapons only afflict small damage, while other weapons can kill an enemy instantly. Weapon power may also require the agent to consider its proximity to the enemy player. This is because it may also receive damage or even die from their own weapon if the enemy is too close.

The number of enemy and friendly players in close proximity can also be considered. For example, if only one enemy player is seen, then lower powered weapons can be used. If two or more enemy players are in close proximity, then higher powered weapons are required in order to stay alive.

The general location of the agent also plays a role in choosing the weapon. For example, if an agent is closer to the home base, it can afford to be killed as it is immediately respawned nearby. This means it can be more aggressive and use powerful weapons even though it will also sustain damage from its own weapons.

#### **6.2.5.2 Health Learning**

This learning problem is concerned with the issue of when an agent should stop what it is doing and attempt to find more health. This includes deciding on which health pack it should attempt to obtain.

There are again many factors that may lead to the decision of where to go, ranging from the agent's assessment of its chances that it can get to a health pack in time, to the commitment it has towards team objectives. Sometimes it may be required for an agent to sacrifice itself for the good of the team. An example scenario is when the agent is defending a friendly player that is carrying the enemy flag. The agent in this case is required to try to take the cross fire of enemy players trying to stop the team mate from scoring a point. The decision of which health pack to try to get depends of the agent's chances of successfully reaching that health pack without being attacked. It therefore requires the agent to learn which areas of the world are usually occupied by enemy, and try to stay clear from such areas.

#### **6.2.5.3 Movement Learning**

This learning problem is concerned with improving the efficiency of moving in UT. This is mainly performed by learning which is the more efficient path to take within the UT world in order to reach a particular destination. The choice of the path to take can be affected by different factors, like the time it takes to reach the destination and the probability of successfully reaching that destination without being killed by the enemy. This information may be used for general movement like trying to reach specified destinations, for example the home and enemy flags, as well as intercepting an enemy player carrying the home flag.

#### **6.2.5.4 Defence Learning**

This learning problem is concerned with improving the efficiency of defending the home flag. Issues that are applicable for this problem include where is the best location around the flag, and which weapon to use in order to successfully prevent an enemy intruder from taking the flag. Other issues include when to order more defence troops to specifically return to the flag, and also when it would be beneficial to request support from the `UASupport` team.

#### **6.2.5.5 Offence Learning**

This learning problem is concerned with finding the most efficient ways to successfully score a point against the enemy. This includes learning which are the best routes to take with the greatest chance of capturing the flag. After the flag has been captured, agents must learn which is the best path to take home in order to successfully score a point. Additionally, the offence team must be able to adapt its tactics based on the movements of the enemy. For example, if the enemy has successfully figured out a defence against a particular offence, then another tactic should be used.

### 6.2.5.6 Human Learning

Human learning is concerned with the agent system being able to adapt to changes in the humans behaviour. The reason for this is that a human cannot be tightly controlled like agents, and therefore must be closely monitored such that the system understands the intentions of the human. The rest of the agents then re-organize their structure in order to accommodate the human. For example, if the human is observed to be defending the flag, then the system can assign the human the role of a `UADefenceTroop`. If the human then becomes bored and tries to capture the flag, the system would automatically reassign the human as an offence team member.

Additionally, the system should be able to adapt according to the human's perceived skill level. If the human is a novice, then the agents should compensate by assigning critical tasks such as defending the home flag and capturing the enemy flag to agents. As the human's skill improves, the agents should adapt accordingly, and shift their behaviour to providing support to the human, rather than completing the tasks themselves.

## 6.2.6 Data Stores

When scenarios and functionalities were being developed in the previous section of this chapter, a number of data stores were included as part of them. This section now specifically identifies all data stores that are part of the design of the system. At this stage it is not necessary to provide details on how these data stores will be implemented. Rather, they are simply identified and a brief description is provided on how they are used. The data stores have been separated into three categories with respect to their usage, and are described in the following three subsections.

### 6.2.6.1 Interfacing Data Stores

A number of JACK beliefsets are used to store information that was received using synchronous messages. The collection of these beliefsets represents the agent's KB about the world. Every time a synchronous message block is received, all relevant beliefs are updated accordingly. The different beliefsets are listed below:

<b>Self:</b> Contains information regarding the agent's state at a given instant. Items include the current health, location, direction, velocity etc.	<b>Invs:</b> Contains information about any inventory items that have been seen in UT up to now.
<b>Players:</b> Contains the last known information about a player seen in UT.	<b>Game:</b> Contains important game information, such as the score.
<b>Navs:</b> Contains information about any navigation points that have been seen in UT up to the instant received.	<b>Flags:</b> Contains information about any flags that have been seen in UT up to now.
<b>Movs:</b> Contains information about any moving items (such as elevators and doors) that have been seen in UT up to the instant received.	<b>Doms:</b> Contains information about any domination control points that have been seen in UT up to now.
	<b>AsyncMsgs:</b> Contains the last known text message that was received. This is used for communications between bots.

#### 6.2.6.2 Learning Data Stores

The following data stores contain information that agents have learned through experiences. Examples include which weapons to use in different situations or which parts of the environment are safer to travel in:

<b>UALearning:</b> Contains learned information about tactical deployments of the entire team.	<b>MovementLearning:</b> Contains learned information about moving around in the UT world, such as which are the quickest and safest paths to follow.
<b>WeaponsLearning:</b> Contains learned information about the use of weapons, such as which weapons to use in which situations.	<b>HealthLearning:</b> Contains learned information about team members managing their health.
<b>DefenceLearning:</b> Contains learned information used by the Defence team.	<b>HumanLearning:</b> Contains learned information regarding the behaviour of a human in the team.
<b>OffenceLearning:</b> Contains learned information used by the Offence team.	

#### 6.2.6.3 Human Management Data Stores

The following data stores contain information about how to monitor and communicate with the human. These are specifically useful when agent players are required to be in the same team as human controlled players.

<b>HumanCommunication:</b> Contains information regarding communicating with	human players.
--	----------------



### 6.3 Architectural Design

The architectural design phase focuses on deciding what agent types will be implemented, and how they interact with other components and agents. The overall static design structure is illustrated through a system overview diagram, and the dynamic behaviour of the system is illustrated using interaction diagrams.

### 6.3.1 Agent Types

This section uses the information presented in the system specification to produce the agent types that are required for this system.

#### 6.3.1.1 Data Coupling

The data coupling diagram identifies which functionalities should be grouped together with which data stores. There are three reasons for grouping them together, firstly, if the functionalities are intuitively related, secondly, if they use the same data, and thirdly if the functionalities interact a lot. The data coupling diagram as generated by the PDT is shown in figure 6.5. The groups are identified within thick black lines.

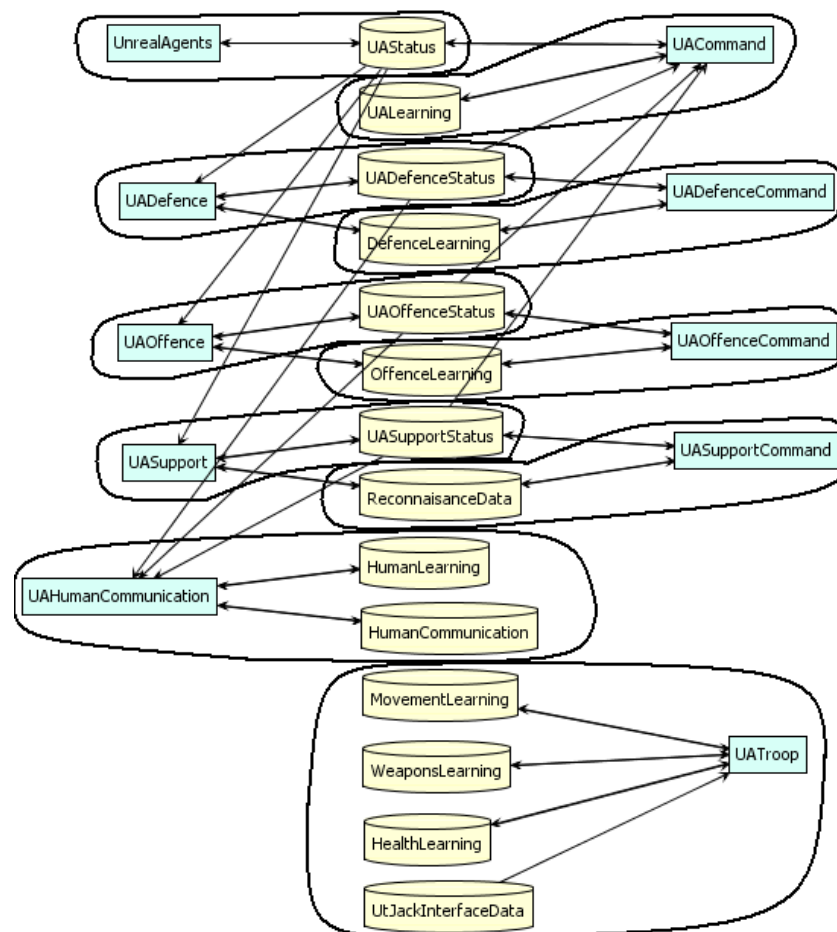


Figure 6.5: Functionality Data Coupling for UnrealAgents

### 6.3.1.2 Agent-Functionality Coupling

Agent-functionality coupling is concerned with the identification of agent types that can perform the functionalities previously described. In total, fourteen agents have been identified for use by this system. There are five commander agents and nine troop agents. Each of the commander agents is responsible for performing two functionalities, the management of its associated team, as well as making command decisions for that team. There is one commander agent for each of the defence, offence and support team, as well as a commander for the entire **UnrealAgents** team. The nine troops are the agents that directly control players in UT, they are split equally, and assigned to the defence, offence and support teams. This arrangement is illustrated graphically in figure 6.6.

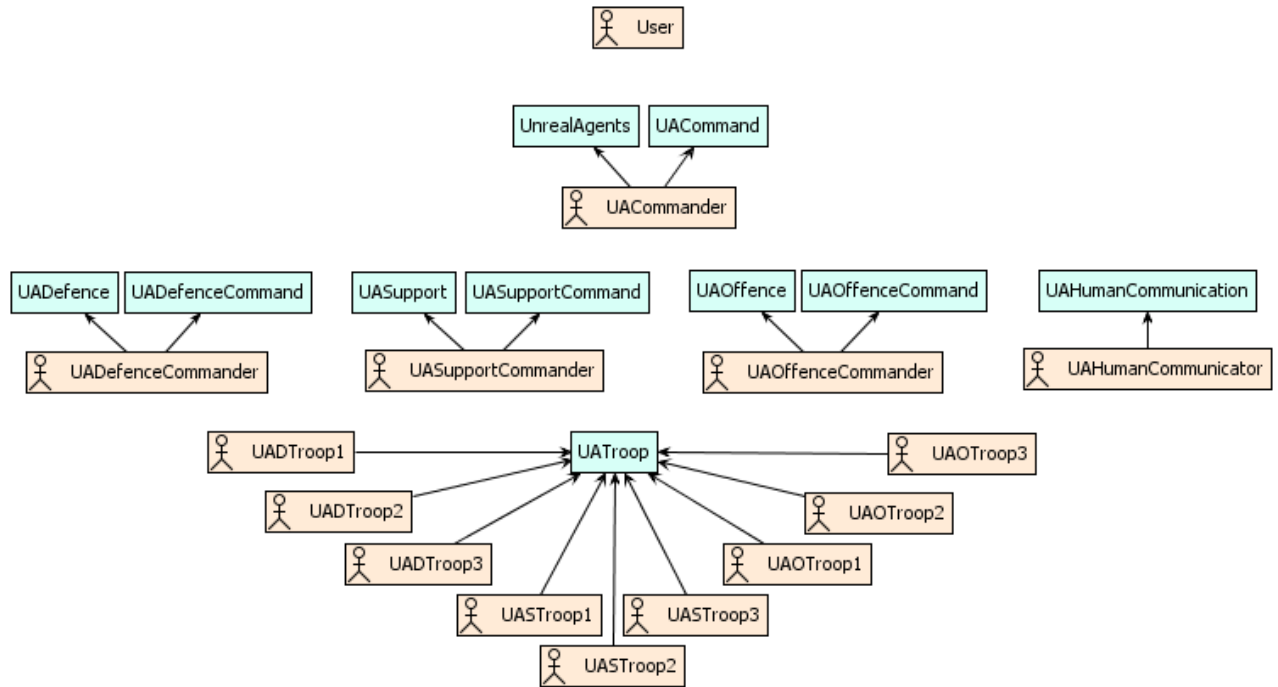


Figure 6.6: Agent-Functionality Coupling for UnrealAgents

### 6.3.1.3 Agent Acquaintance

The agent acquaintance section describes the communication being performed between the different agents. As illustrated in figure 6.7, a command style communication model has been adopted. Hence, troop agents can only communicate with their team members and their corresponding commander agent. Similarly, the commander agents can only communicate with their superior. If they wish to coordinate with another team, they must first communicate their intentions to their commander, who decides whether to relay their request or not.

### 6.3.1.4 Agent Descriptors

Agent descriptors are very similar to the functionality descriptors, this is because agents themselves perform the functionality defined by their descriptors. Agent descriptors however, provide more detailed and more specific information which leads to the implementation of the

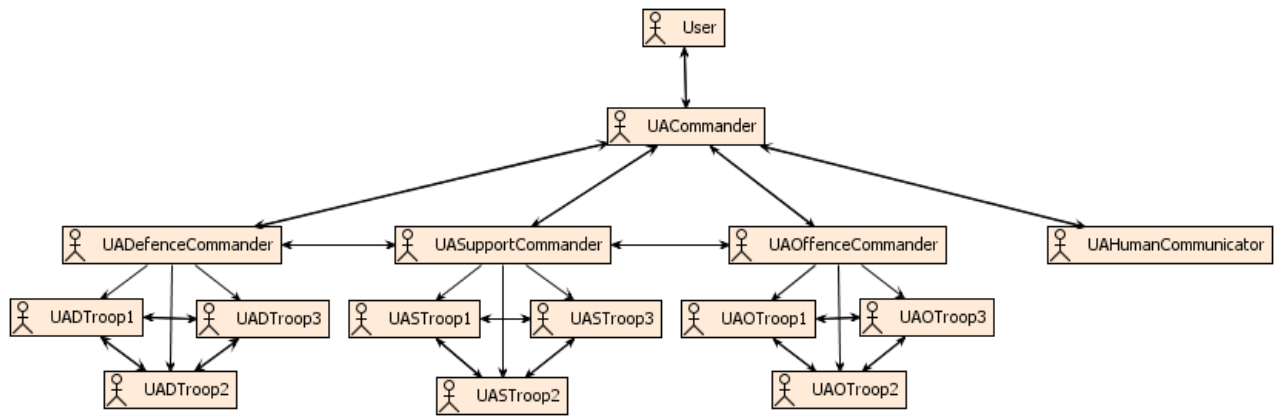


Figure 6.7: Agent Acquaintance

agents. Six agent descriptors are shown in this section, one for each of the agent types previously identified.

Name	UACommander
Description	Commands the entire UnrealAgents team. Is responsible for making tactical decisions regarding the activities performed by subteams.
Cardinality	One/User, One/UADefenceCommander, One/UAOffenceCommander, One/UASupportCommander.
Lifetime	Instantiated at the start of a game and destroyed at the end of the game.
Initialization	Analyze the UT world and enemy, load known data about that world. Initialize commander agents for subteams.
Demise	Write all known info into a file associated with the UT world.
Functionalities	UnrealAgents and UTCommand
Uses data	UAStatus and UALearning
Produces data	UAStatus and UALearning
Goals	WinCTF
Percepts	Latest known situation of the UT world
Actions	Assign commander agents for subteams. Instigate global team tactics. Assign global team learning. Facilitate requests for support.
Interactions	ProvideDefence, ProvideSupport, ProvideOffence, ManageHuman.
Name	UADefenceCommander
Description	Commands the defensive team. Is responsible for making tactical decisions regarding troops defending the flag.
Cardinality	One/UACommander, Three/UATroop
Lifetime	Instantiated at the start of a game and destroyed at the end of the game.
Initialization	Analyze previously known data. Initialize troop agents.
Demise	Release Troops
Functionalities	UADefence and UADefenceCommand
Uses data	UADefenceStatus and UADefenceLearning
Produces data	UADefenceStatus and UADefenceLearning
Goals	Achieve, PreventEnemyScoring
Percepts	Latest known situation of the UT world
Actions	Create troop agents. Instigate defence tactics and learning. Request support when required.
Interactions	ProvideDefence, DefenceOrders, DefenceSupport.

Name	UAGOffenceCommander
Description	Commands the defensive team. Is responsible for making tactical decisions regarding scoring points against the enemy.
Cardinality	One/UAGCommander, Three/UATroop
Lifetime	Instantiated at the start of a game and destroyed at the end of the game.
Initialization	Analyze previously known data in UT world. Initialize troop agents of the offence team.
Demise	Release Troops
Functionalities	UAGOffence and UAGOffenceCommand
Uses data	UAGOffenceStatus and UAGOffenceLearning
Produces data	UAGOffenceStatus and UAGOffenceLearning
Goals	Achieve, PreventEnemyScoring
Percepts	Latest known situation of the UT world
Actions	Create troop agents. Initiate offence tactics and learning. Request support when required.
Interactions	ProvideOffence, OffenceOrders, OffenceSupport.

Name	UAGSupportCommander
Description	Commands the offensive team. Is responsible for making tactical decisions on where and how to perform reconnaissance. Decide on learning activities performed.
Cardinality	One/UAGCommander, Three/UATroop
Lifetime	Instantiated at the start of a game and destroyed at the end of the game.
Initialization	Analyze previously known data in UT world. Initialize troop agents of the support team.
Demise	Release Troops
Functionalities	UAGSupport and UAGSupportCommand
Uses data	UAGSupportStatus and UAGSupportLearning
Produces data	UAGSupportStatus and UAGSupportLearning
Goals	Explore
Percepts	Latest known situation of the UT world
Actions	Create troop agents. Initiate support tactics and learning. Provide support when required.
Interactions	ProvideSupport, SupportOrders, DefenceSupport, OffenceSupport.

Name	UATroop
Description	Controls a player within UT. Can perform any of the roles required by the defence/offence/support teams. Can also change its team during operation.
Cardinality	One/UAGDefenceCommander or One/UAGOffenceCommander or One/UAGSupportCommander
Lifetime	Instantiated at the start of a game and destroyed at the end of the game.
Initialization	Initialize connection with UT, obtain initial sensor information, join team, wait for orders.
Demise	Disconnect from UT server
Functionalities	UATroop
Uses data	Sensor data within UtJackInterface.
Produces data	MovementLearning, HealthLearning, WeaponsLearning.
Goals	All leaf goals from diagram 6.2
Percepts	UtJackInterface synchronous and asynchronous senses.
Actions	UtJackInterface actions.
Interactions	DefenceOrders, OffenceOrders, SupportOrders.

Name	UAGHumanCommunicator
Description	Monitors a human player and communicates all activities to the UAGCommander. Can also provide estimates of what the human is doing.
Cardinality	One/UAGCommander
Lifetime	Instantiated when a human join the team. Destroyed when the human exits the team.
Initialization	Obtain information about human, associate temporary team functionality, negotiate to change functionalities when human exhibits different behaviour.
Demise	Say GoodBye to human when exit detected. Negotiate for agent replacement with UAGCommander.
Functionalities	UAGHumanCommunicator
Uses data	UtHumanMonitor.
Produces data	UAGStatus.
Goals	None.
Percepts	Information provided by UtHumanMonitor human interface software.
Actions	UtJackInterface actions.
Interactions	Actions provided by the UtHumanUI human interface software.

## 6.3.2 Behaviour Adaptation

This section defines a number of learning descriptors that encapsulate specific parts of the learning problems described in section 6.2.5. They are separated into sections according to the learning problem that they are based on.

### 6.3.2.1 Learning About Weapons

**Learning Goal:** UseBestWeaponLearningDescriptor

**RL Algorithm:** QLearning

**Policy:** EGreedy

**Perception:** WeaponPerception

- Location of agent
- Proximity to closest enemy player
- Number of enemy players nearby
- Number of friendly players nearby
- Weapon being used

**Actions Available:** Choice very depending on the weapon types available to the agent.

**Goal Status:** Learning episode starts when enemy player is range and finishes when no enemy player is in range.

**Reward Function:** Return -1 if agent dies at any time this learning goal is active.

**Learning Parameters:** None.

**Translation:** None

**Description:** This learning goal learns which weapon to use in different situations.

### 6.3.2.2 Learning About Health

**Learning Goal:** GetHealthLearningDescriptor

**RL Algorithm:** QLearning

**Policy:** EGreedy

**Perception:** HealthPerception

- General area of location of agent
- Current level of health
- Number of enemy players nearby
- Number of friendly players nearby

**Actions Available:** Request permission to look for health, or continue on.

**Goal Status:** Learning episode starts when player enters the game or respawns after it gets killed and finishes when player is killed.

**Reward Function:** Return -1 if agent dies.

**Learning Parameters:** None.

**Translation:** None.

**Description:** This learning goal makes a decision when the agent should stop what its doing and request permission to look for a health pack.

**Learning Goal:** FindHealthLearningDescriptor

**RL Algorithm:** QLearning

**Policy:** EGreedy

**Perception:** FindHealthPerception

- Location of agent

**Actions Available:** Choices are locations of nearby health packs. If no health packs close enough, choices are nearby navigation points.

**Goal Status:** Learning episode starts when player is given permission to look for health and finishes when it finds a health pack or when it is killed instead.

**Reward Function:** Return -1 if agent dies and +1 if successfully obtained a health pack.

**Learning Parameters:** None.

**Translation:** None.

**Description:** This learning goal tries to find safest path for the agent to follow in order to obtain a health pack.

### 6.3.2.3 Learning About Movement in UT

**Learning Goal:** FindOptimalPathsLearningDescriptor

**RL Algorithm:** QLearning

**Policy:** EGreedy

**Perception:** LocationPerception

- Location of agent

**Actions Available:** Choices are locations nearby navigation points.

**Goal Status:** Learning episode starts when player is given a destination to reach and finishes when it reaches that destination or when it is killed instead.

**Reward Function:** Return +100 if the player reaches the destination. Return -1 for all navigation points reached before the destination is reached.

**Learning Parameters:** None.

**Translation:** None.

**Description:** This learning goal tries to find the quickest path to follow to a particular destination.

### 6.3.2.4 Learning how to Defend Better

**Learning Goal:** DefenceEfficiencyLearningDescriptor

**RL Algorithm:** QLearning

**Policy:** EGreedy

**Perception:** DefenceStatusPerception

- Location of all defence players
- Location of known enemy players
- Status of home flag

**Actions Available:** Choices are locations that defence players should be defending from.

**Goal Status:** Learning episode starts at the beginning of the game and ends at the end of the game.

**Reward Function:** Return +1 for every enemy player killed. Return -100 if home flag is taken.

**Learning Parameters:** None.

**Translation:** None.

**Description:** This learning goal tries to find the locations where defence players should be placed in order to effectively defend the home flag.

**Learning Goal:** StopEnemyFlagCarrierLearningDescriptor

**RL Algorithm:** Sarsa

**Policy:** EGreedy

**Perception:** DefenceStatusPerception

- Location of all defence players
- Location of known enemy players
- Status of home flag

**Actions Available:** Choices are locations that defence players should go to next in order to catch up to the enemy flag carrier.

**Goal Status:** Learning episode starts when home flag is taken and finishes when home flag is returned or enemy scores.

**Reward Function:** Return -100 if enemy scores. Return +100 if home flag is returned.

**Learning Parameters:** None.

**Translation:** None.

**Description:** This learning goal tries to find the best locations that defence players should go to in order to successfully catch up to an enemy player carrying the home flag.

### 6.3.2.5 Learning how to Score Points Quicker

**Learning Goal:** CaptureEnemyFlagLearningDescriptor

**RL Algorithm:** Sarsa

**Policy:** EGreedy

**Perception:** OffenceStatusPerception

- Location of all offence players
- Location of known enemy players
- Status of enemy flag

**Actions Available:** Choices are locations that defence players should go to next in order successfully reach the enemy flag.

**Goal Status:** Learning episode starts when enemy player enters the game or respawns and finishes when the player successfully reaches the enemy flag or is killed in the process.

**Reward Function:** Return +100 if flag reached.

**Learning Parameters:** None.

**Translation:** None.

**Description:** This learning goal tries to find the best locations that offence players should go to in order to successfully reach the enemy flag.

**Learning Goal:** ScoreLearningDescriptor

**RL Algorithm:** Sarsa

**Policy:** EGreedy

**Perception:** OffenceStatusPerception

- Location of all offence players
- Location of known enemy players
- Status of enemy flag

**Actions Available:** Choices are locations that defence players should go to next in order successfully reach the home base while carrying the flag.

**Goal Status:** Learning episode starts when an offence player captures the flag and finishes when the player scores a point or is killed in the process.

**Reward Function:** Return +100 if a point is successfully scored.

**Learning Parameters:** None.

**Translation:** None.

**Description:** This learning goal tries to find the best locations that offence players should go to in order to successfully reach the home base and score a point while carrying the enemy flag.

### 6.3.2.6 Learning About a Human Team Member

**Learning Goal:** HumanRoleLearningDescriptor

**RL Algorithm:** QLearning

**Policy:** EGreedy

**Perception:** HumanStatusPerception

- Location of human player

**Actions Available:** Re-assign a defence or offence or support role to human. Alternatively keep the role the same as before.

**Goal Status:** Learning episode starts a human joins the team and finishes when the human leaves the team or the game finishes.

**Reward Function:** Return +10 whenever the human achieves a goal of its associated team. For example, score a point while in offence or kill an enemy player close to the flag while in defence.

**Learning Parameters:** None.

**Translation:** None.

**Description:** This learning goal attempts to assign a human player an appropriate role by observing the player's movements.

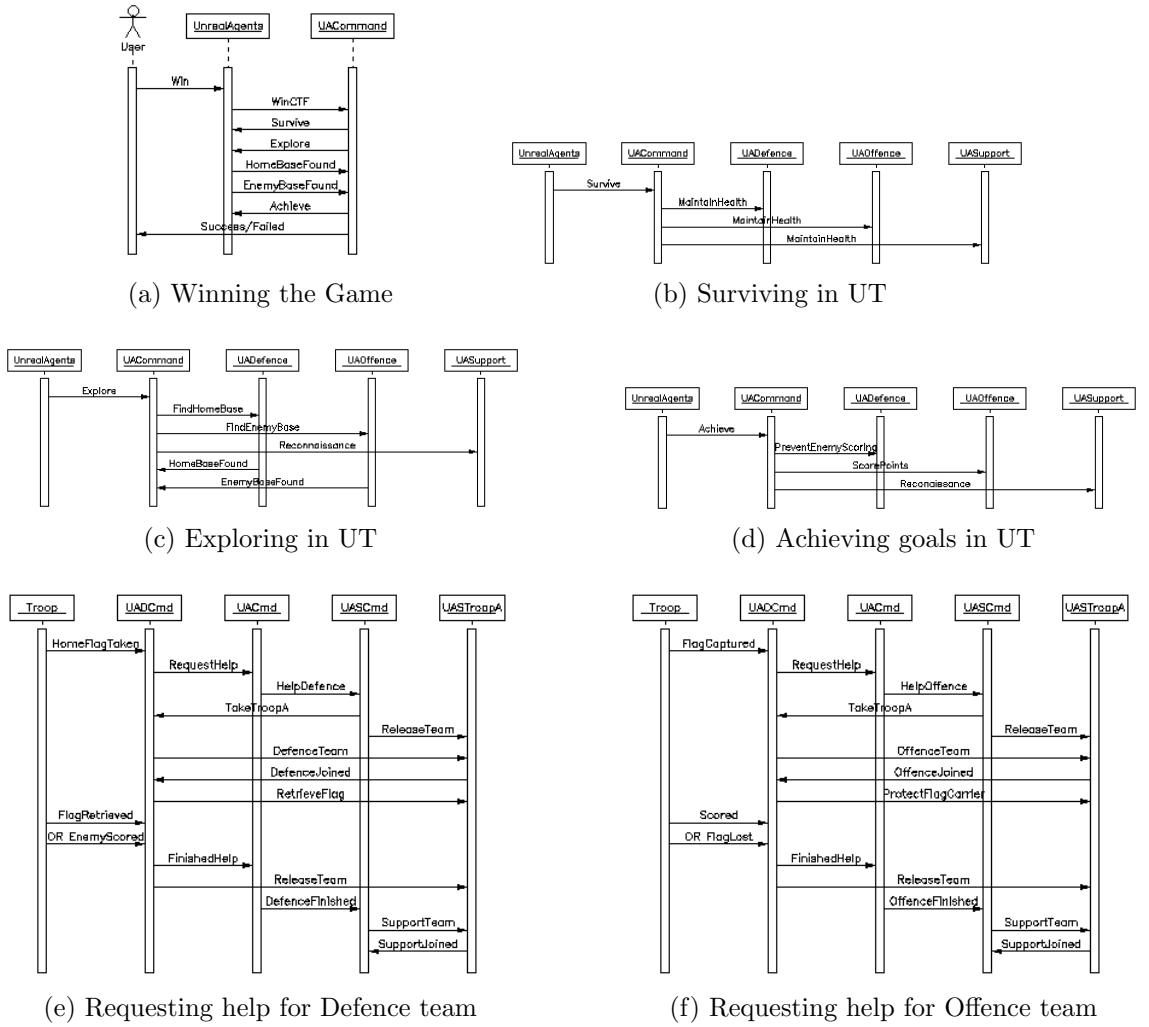


Figure 6.8: Interaction Diagrams

### 6.3.3 Agent Interactions

Interaction diagrams attempt to capture how the agents interact with each other. The interaction diagrams lead to the development of protocols of acceptable communication between agents. Figure 6.8a shows the interaction of `UnrealAgents` with the `UACCommand` subteam when a `Win` message is received. Figure 6.8b shows how the agents are instructed to survive in the game. Figure 6.8c shows how agents set out to explore the world. Defence agents look for the home base, offence agents look for the enemy base and support agents look for weapons and inventory items. Figure 6.8d shows how teams are ordered to achieve their corresponding goals. Finally, figure 6.8e and figure 6.8f show the interaction of the offence and defence teams requesting help from the support team.

#### 6.3.3.1 Protocol Descriptors

Protocols are elaborate interaction diagrams that are specifically concerned with messages that are required in situations where an error has occurred. With robust protocols, agents can easily reinitialize their communication in order to continue co-operating toward achieving their goals. Protocols can be very detailed, they are not included as part of this design because they are beyond the scope of this thesis.



### 6.3.3.2 Message Descriptors

This section lists different events that can occur in the agent system.

#### Events generated by monitoring the agent's beliefs.

- A team has scored
- A flag has been taken.
- An enemy player is in close proximity
- An inventory item has been obtained.
- Weapon ammo is running low.
- Weapon ammo has depleted.
- Health is below 50%.
- Danger, health has dropped below 10%.

#### Events generated by asynchronous UT messages.

- Received a global text message.
- Received a team-only text message.
- Received a predefined UT command message.
- Entered a different zone in the world.
- Collided with a wall.
- Danger of falling off a ledge.
- Fell off the ledge.
- Bumped into another player.
- Heard someone pick up a weapon.
- Heard something making noise nearby.
- An enemy is firing weapons nearby.
- Another friendly or enemy player has been killed.
- The agent itself was killed.
- The agent itself has received damage.
- The agent successfully shot another player.

#### Events originating from agent team members.

- A mission objective is in danger of failing.
- Defence perimeter has been breached.
- Team member flag-carrier is in danger of
- being killed.
- Request for defence support.
- Request for offence support.
- Agent has changed roles in the team.

#### Events originating from human team members

- Human player has sent message
- Human player has issued a command
- Human player has acknowledged orders.
- Human player has assumed a new team role.
- Human player is in danger of being killed.

### 6.3.4 Overall System Architecture

This section integrates all information previously presented in order to gain a good perception of the entire system.

#### 6.3.4.1 Data Flow

The diagram shown below describes how the percepts are organized in the UA system. Agent troops and human players are the only ones that directly receive percepts from UT. The rest of the agents receive events and data passed to them by these agents.

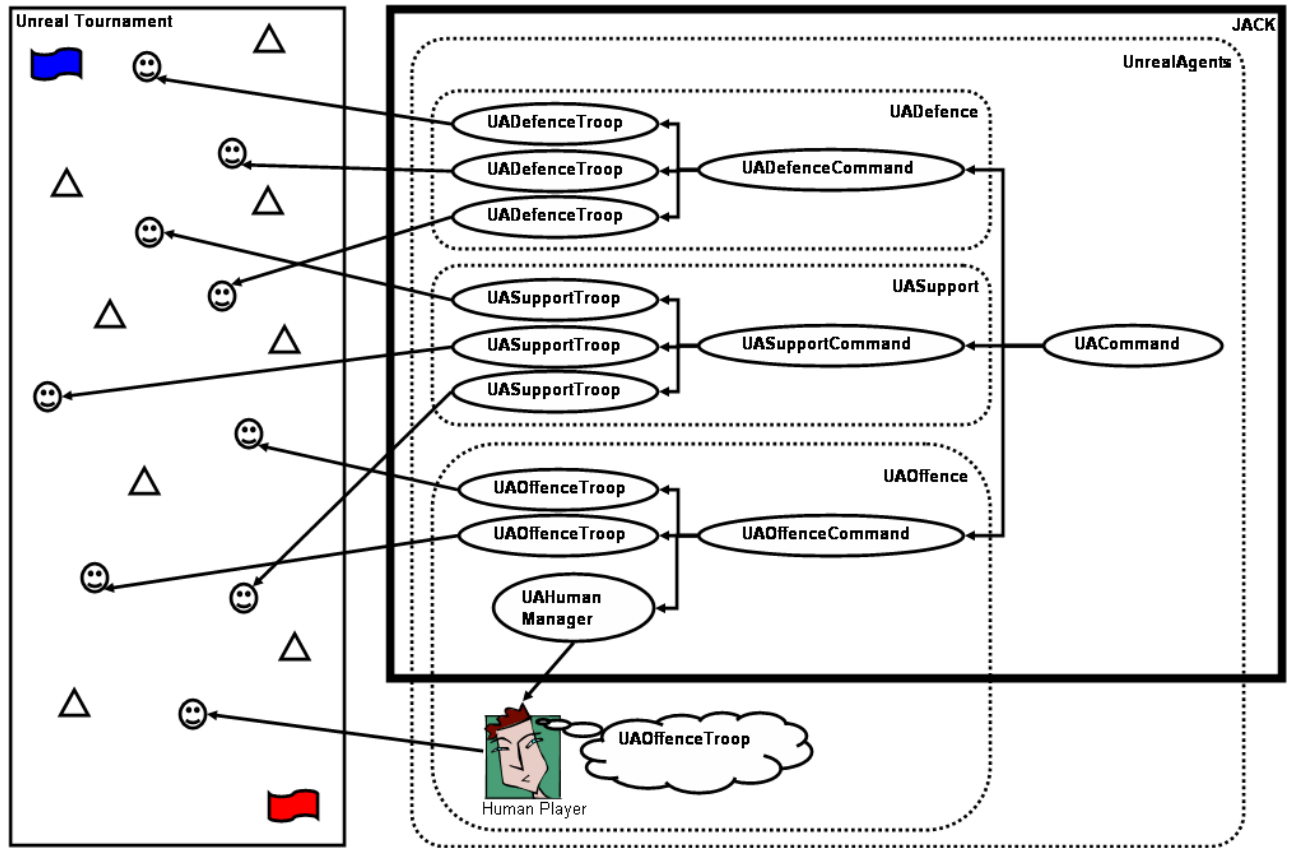


Figure 6.9: Percept Linking for UnrealAgents

#### 6.3.4.2 Shared Data

The following data is shared by two or more agents in the system.

**UAStatus:** UnrealAgents and UADefence

**UADefenceStatus:** UADefence, HumanCommunicator and UADefenceTroop

**UAOffenceStatus:** UAOffence, HumanCommunicator and UAOffenceTroop

**UASupportStatus:** UASupport, HumanCommunicator and UASupportTroop

**UATroopStatus:** UATroop, UADefence, UAOffence and UASupport

#### 6.3.4.3 System Overview

The system overview diagram provides an overall picture of the components that make up the system. All the agents are shown as well as the data that they use, the communications they perform with other agents, and also any percepts and actions that they have. The system overview diagram for UnrealAgents is shown in figure 6.10, which was generated by the PDT.

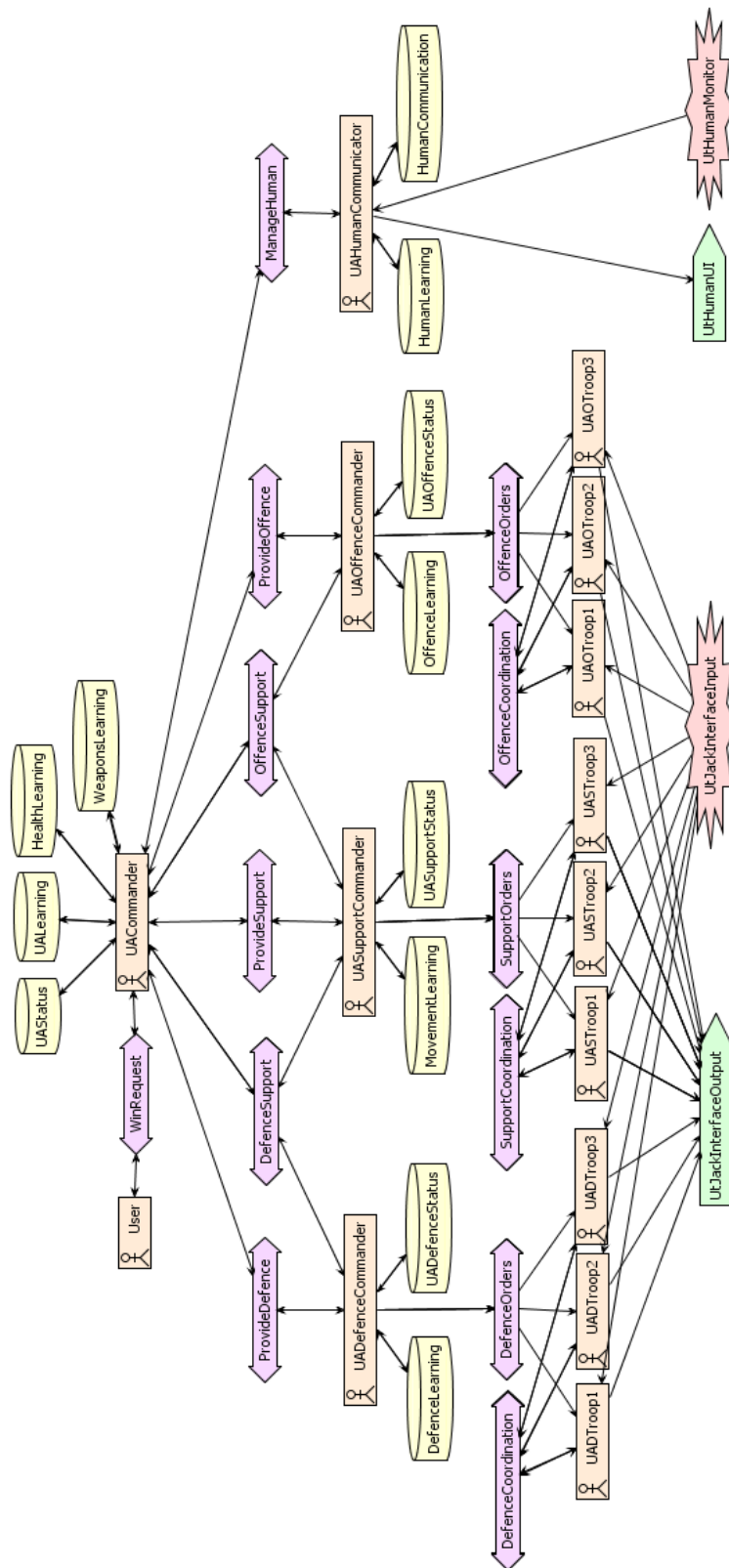


Figure 6.10: System Overview for UnrealAgents

## 6.4 Detailed Design

The third and final part of the Prometheus agent oriented methodology describes the internal design of agents. It can be divided into three major parts. The first part involves using the previously developed interaction diagrams to define system processes that illustrate the flow of activities that the system will need to execute. The second and third parts are agent and capability descriptors respectively that describe the internal structure of agents. In Prometheus these two parts are based on the JACK agent development environment. All agent behaviours are implemented by specifically creating events, plans and data for them. This means that the complete detailed design of this system would be a lengthy document. Considering the aim of this chapter is the illustration of how learning is included within agent-oriented design, only a small subset of the whole detailed design is presented in this section, with a specific focus on demonstrating learning in the detailed design.

### 6.4.0.4 Processes

In the system specification some high-level processes were specified by scenarios, and from these interaction diagrams were then designed. This section combines the two in order to graphically illustrate some the processes followed by the system. The diagrams for the defence, offence, survive and support processes are illustrated in figures 6.11a, 6.11b, 6.11c, and 6.11d respectively. It can be seen that the support process activates the **FindOptimalValues** goal after both of the **FindInventories** and **FindHealthPacks** have successfully finished. Hence, the support process firstly focuses on finding the home flag and the enemy flag, then in gathering important information such as the location of weapons and health packs, and then tries to improve performance by finding optimal values for behaviours.

One of the subgoals executed by **FindOptimalValues** is **FindOptimalMovement**. Recall from section 6.2.5.3 the **FindOptimalMovement** goal is concerned with learning the best paths for an agent to take in order to reach its required destination. For this reason, it is assumed that the behaviour beneath **FindOptimalMovement** initiates learning using a **LearnPath** event for each desired destination. The behaviour associated with learning a path is described in the remainder of the chapter.

### 6.4.1 Partial UASTroop Learning Agent

This section describes a partial detailed design of the **UASTroop** that illustrates how the *CHRIS* framework can be used for learning. Figure 6.12 illustrates the agent overview diagram for this partial implementation of **UASTroop**. This diagram can be thought of as looking inside one of the **UASTroop** agents.

The **UASTroop** agent design would normally handle all relevant goals discussed in the architectural design stage. However, as shown in figure 6.12, only the **FindPath** subgoal is considered. **FindPath** is an event of type **Goal**, hence, it is automatically handled by the *CHRIS* framework. The agent also contains the two capabilities, one is the *CHRIS* capability, and the other is the **Move** capability. There are also four plans declared directly within the agent that handle three **Target** events and one **ActionEvent** posted by the decision and learn-

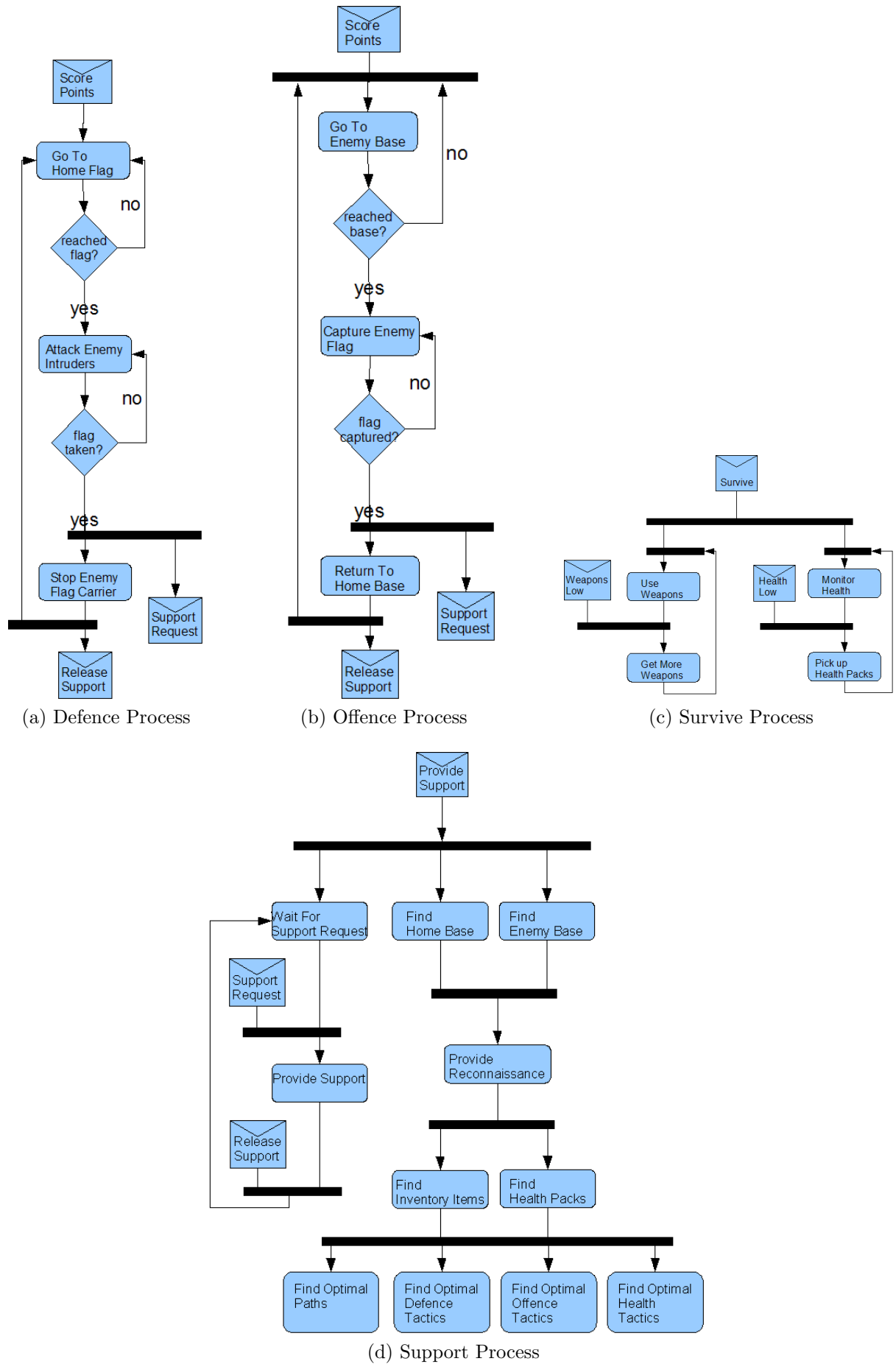


Figure 6.11: Process Specifications for UnrealAgents

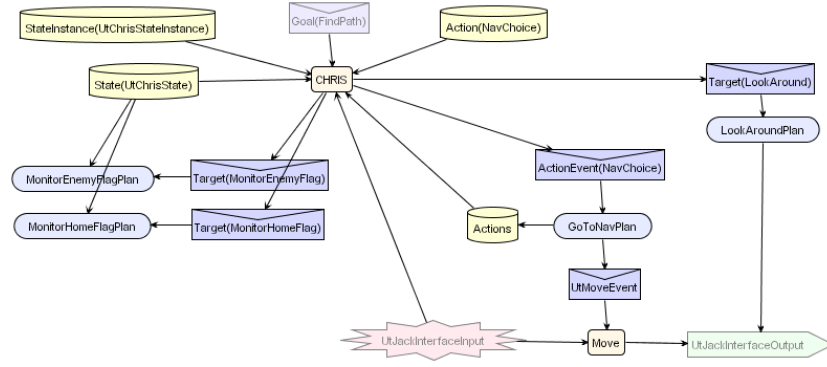


Figure 6.12: Partial UASTroop Agent Overview

ing packages respectively. The **Actions** data store is part of the action package, it is used to keep track of when the agent has started and finished handling the **ActionEvent**. The other data store is an object of type **NavChoice** that extends **Action**. Each of these components and their purpose is described in the following subsections.

#### 6.4.1.1 Move Capability

This section describes the design of the **Move** capability that allows an agent to move a player around in UT. If a designer wishes for a particular agent to be able to move a player in UT, there is no need to consider any additional events, data and plans. They are all automatically included when the **UtMove** capability is used. Hence, the **UtMove** capability should be included in all troop agents. Examples of other capabilities that should be included within all troops are **UtWeaponsCap** for the usage of weapons, and **UtMaintainHealth** for monitoring the health status, and asking the commander agent for permission to look for health when required.

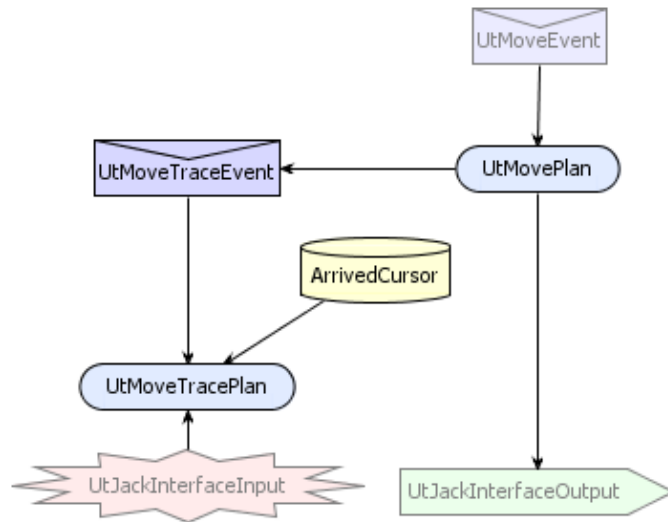


Figure 6.13: UT Movement Capability

The internal design of the **UtMoveCap** capability is illustrated in figure 6.13. The **UtMovePlan** plan handles the external **UtMoveEvent** event by sending a **RUNTO** message to UT (see appendix A for the message protocol) using **UtJackInterface**, and posting an internal **UtMoveTraceEvent** event for tracing the progress of the agent reaching the destination. The **UtMoveTraceEvent** is

handled by the `UtMoveTracePlan` plan monitoring the continuously changing location. When the agent reaches the specified location, the `UtMoveTracePlan` succeeds which also causes the `UtMovePlan` to succeed. Conversely, if the agent does not reach its destination within a specified time (5 seconds) then both of the plans fail.

## 6.4.2 Reasoning Process

The reasoning process of the agent is structured around the *CHRIS* framework which includes the *CHRIS* capability. Hence, there is no need to implement the code for the reasoning loop. The general design of the *CHRIS* capability for this problem is illustrated in figure 6.14. It is consisted of five sub-capabilities implemented as separate packages within the framework that use data and/or events relevant to the problem. Each of these are described in detail in the following subsections.

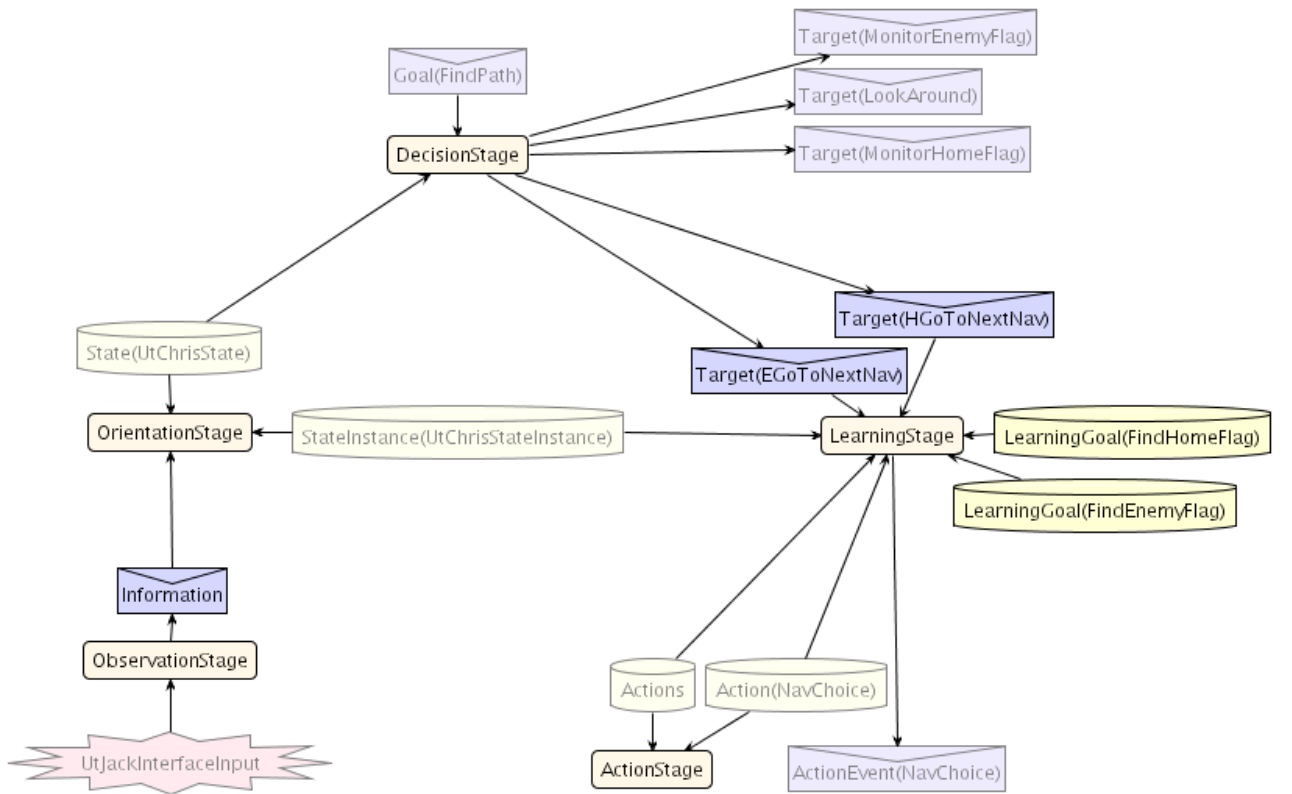


Figure 6.14: Partial UASTroop Reasoning Overview

### 6.4.2.1 Observing New Information

Figure 6.14 shows that the observation stage receives sensations and posts `Information` messages regarding the sensations it has received. The implementation of the system however, is a little different as necessitated by previous work, it involves the `UtChrisState` extending the `UtJackInterface` class and also implementing the framework's `State` interface as shown in code listing E.1. Hence, `UtChrisState` is able to perform all operations up to and including the `State` of the orientation package.

### 6.4.2.2 Updating the Beliefs

In order to obtain a working system, the orientation package requires the implementation of a **State** and a relevant **StateInstance** for the application. As mentioned previously, the **State** in this case is implemented by extending **UtJackInterface**. The **StateInstance** however needs to be specifically implemented such that it is able to obtain a complete snapshot of all beliefs available within **UtChrisState**.

The source code for **UtChrisStateInstance** is shown in code listing E.2. It contains internal variables for capturing all information within **UtChrisState** (lines 2-9). They are populated by the constructor (lines 12-19). The **equivalent()** method (lines 21-28) only returns **true** if all information between two instances is equal.

### 6.4.2.3 Decision Stage

The decision package manages goals for the agent. There is however no need to develop any events and plans as it only involves populating the framework's **Goals** beliefset accordingly. In a goal hierarchy, leaf nodes are posted as **Targets** by the framework. There are two ways to handle these targets. Firstly, implementing a plan that handles a **Target** event. Secondly, associating a **LearningGoal** with the **Target**.

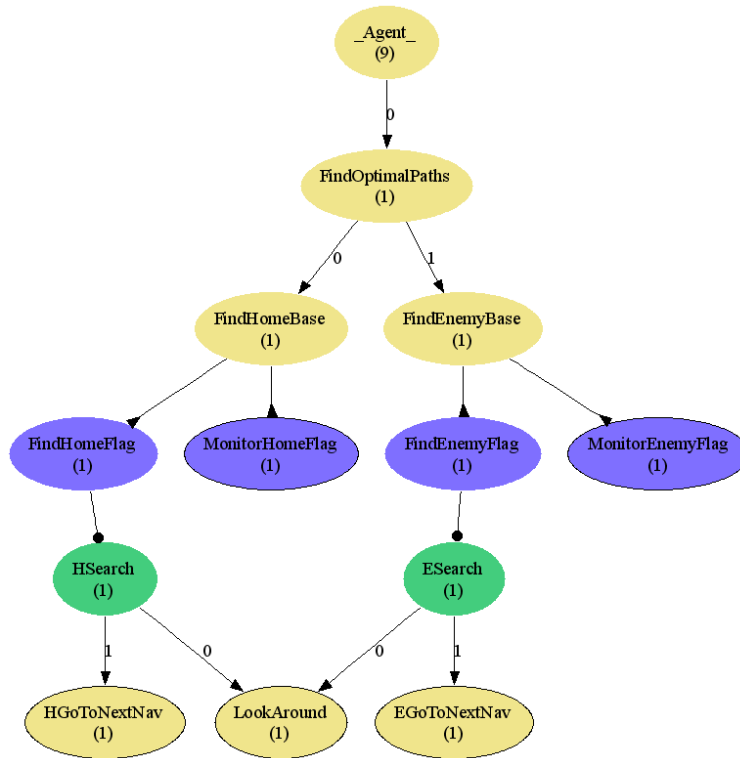


Figure 6.15: UASTroop Decision Goal Hierarchy

For this particular problem the **Goals** beliefset was populated with the structure shown in figure 6.15. The diagram shown was automatically generated using **GoalsGenerator**, a tool provided with the **CHRIS** framework. The root of the tree is called **\_Agent\_**, it is not an actual goal but it represents the level of the agent where all goals originate from.

In order to keep the example simple, the hierarchy for subgoals is presented after the **FindOptimalPaths** goal. The **FindOptimalPaths** goal has two procedural-type subgoals,



**FindHomeBase** and **FindEnemyBase**. Each of their subgoals are **either**-type subgoals. This means they are posted in parallel, and when one succeeds or fails then the associated parent goals also succeed or fail respectively. Considering that the two resulting branches are exactly the same but they are looking for the home base or the enemy base, only the home branch will be described.

An important thing to notice is that the **MonitorHomeFlag** is a leaf node whereas **FindHomeFlag** has further subgoals. This means that an external plan is required to handle **MonitorHomeFlag**. The **MonitorEnemyFlagPlan** plan and the **MonitorHomeFlagPlan** plan specifically handle the relevant **Target** and are used for monitoring the agent's proximity to each of the flags. When the agent reaches the associated flag, the corresponding plan succeeds, hence notifying the agent that it has reached its destination.

Careful observation of the goal hierarchy however will reveal that the **FindHomeFlag** can never succeed or fail. This is because the **ESearch** subgoal is a **resident**-type subgoal which will always repost itself until its parent goal becomes inactive by some external means. The effect of this is that the agent will keep on searching until it reaches the base, in which case the **MonitorHomeFlag** will succeed, this will deactivate processing for **FindHomeFlag** and also all resulting subgoals.

**HSearch** has two **procedural**-type subgoals. They are both leaf nodes and require specific plans to be implemented to handle them. The **LookAround** target and associated **LookAroundPlan** plan, cause the agent to look around its current location and build a list of the navigation choices available to it from that location. The **HGotoNextNav** target does not require a plan because the **FindHomeFlagLearningGoal** is associated with it. The source code for this is illustrated in code listing E.7.

#### 6.4.2.4 Action Stage

The action stage involves the definition of the actions available to the agent for the specific problem. This is an important component for UT because the number of actions available to the agent are infinite at any time. It therefore becomes necessary to filter the actions into the ones that are only available for the particular problem. In this case, the actions available to the agent have been filtered to moving from its current location to a navigational point that is directly reachable. The actions therefore reduce to a small finite number of navigation points close to its location as illustrated in figure 6.16, the source code for this action is shown in code listing E.3.

The **GoToNavPlan** plan handles an **ActionEvent** that contains an instance of a **NavChoice** action. It is responsible for causing the UT Player to move to the requested navigation point using the **Move** capability described previously. The source code for this plan is shown in code listing E.4.

#### 6.4.2.5 Learning Stage

The learning stage is responsible for learning the quickest path to the home flag and then the enemy flag. Before delving into the learning goals themselves, it is necessary to understand the **Perception** used.

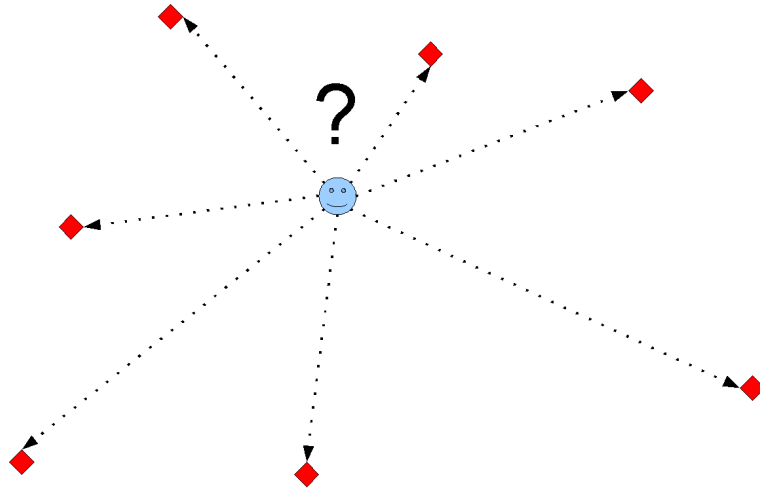


Figure 6.16: Which Action to Take?

The aim of the perception is to generalize the `StateInstance` such that it is only applicable for the specific learning goal. In this case the perception is defined as the agent's current location. If the agent's absolute location was used as the basis for the perception, then there would still be too many states. This is because absolute locations are decimal values with a very small granularity. Hence, there would be too many possibilities. For this reason, the `Location` perception is defined based on the closest navigation point to the agent. Considering there is a very limited amount of navigation points available in the game (usually less than 100) then there is also a limited number of possible perceptions. The source code for the `Location` perception is illustrated in code listing E.5.

The two learning goals required for learning the closest paths to the home and enemy flags are shown in code listings E.7 and E.6 respectively.

The `FindHomeFlagLearningGoal` uses the `QLearning` algorithm for learning and the `EGreedy` action selection policy (line 5). As explained previously, the `getPerception()` method creates a `Location` perception based on the closest `Nav` known to the agent (line 9). The reward function (lines 12-26) returns two types of rewards depending on the action the agent has chosen. Firstly, if the action results in the agent reaching the home flag (line 20), then function returns a reward of 20.

Although it would have been perfectly acceptable to return a reward of `-1` in any other case, a hint was chosen to be given to the agent. Two rewards are provided based on the direction the agent is traveling and the intuitive knowledge of the designer about the general direction to the flag. When the agent travels in the general direction of the flag, a reward of `-1` is given, whereas when it travels in the other direction, a reward of `-2` is given. The reward is always negative in order to force the agent to find the quickest possible path from the start to the destination.

## 6.5 Performance Results

The plot shown in figure 6.17 illustrates how quickly the agent is able to firstly reach the home flag and then the enemy flag. It can be seen that the agent initially performs very poorly

taking nearly 650 seconds (over 10 minutes) to find both flags. On subsequent runs however, the agent finds both flags much quicker. Initially dropping to approximately 220 seconds and then under 200 seconds. The simulation was left to run for many subsequent runs, it can be observed that after approximately 190 runs the agent always finds both flags in less than 100 seconds with the average time being approximately 50 seconds. As a matter of interest a human that knows exactly where to go, can reach both flags in approximately 45 seconds.

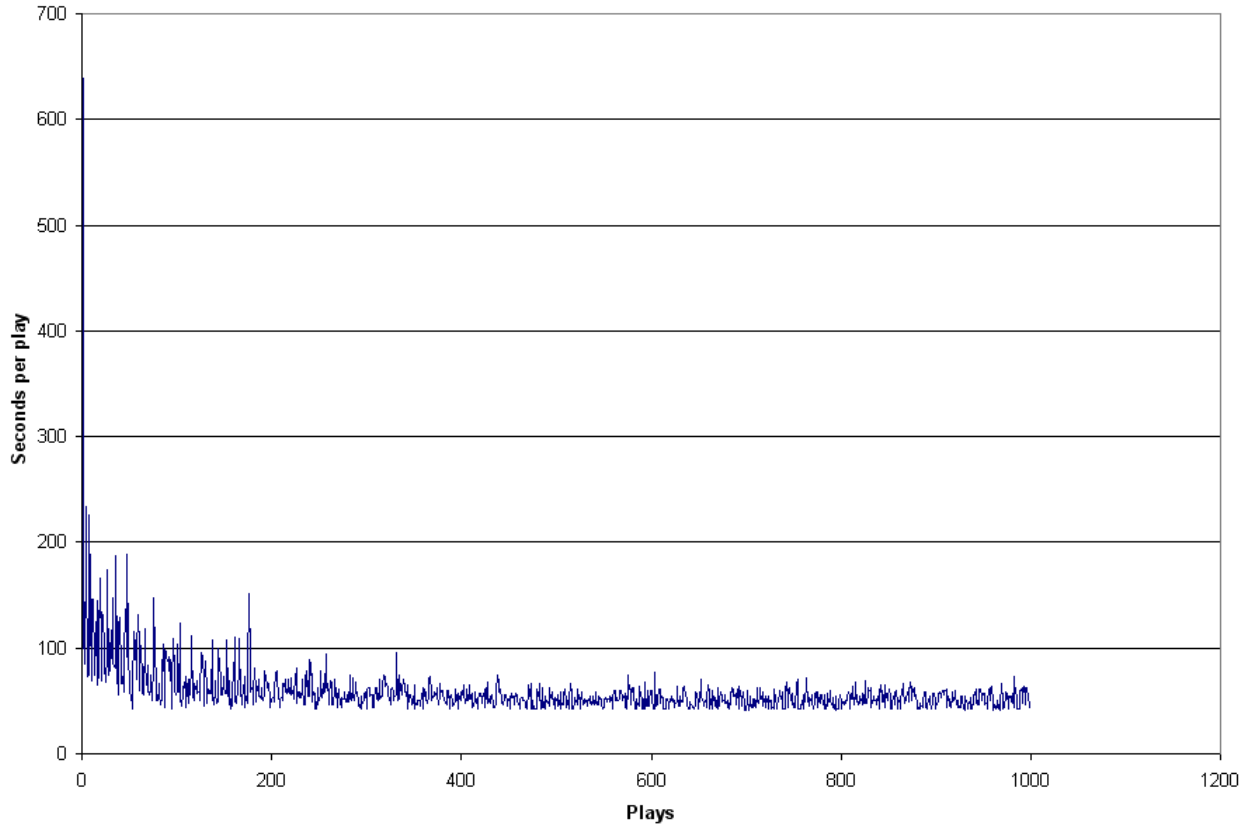


Figure 6.17: Finding The Quickest Path Between The Home Flag And Enemy Flag

## 6.6 Discussion

This chapter presents the design of an agent-based system required to control a team of players within UT. The design follows the Prometheus agent-oriented design methodology with a few extra artifacts specifically applicable to agent learning.

The Prometheus design methodology is segmented into three major phases. The specification phase borrows heavily from requirements engineering and aims to understand what exactly is required to implemented, this includes descriptions of learning problems to be solved. The architectural design phase defines additional details regarding the overall structure of agents. This includes the definition of learning descriptors. Finally, the detailed design phase defines how the internal components of agents are implemented. This includes how to observe new information and update beliefs. The goals of the agent, the actions available to the agent, and finally, the required **LearningGoals**. This chapter demonstrates the agent-oriented development process by presenting the design of a system controlling a team players within UT.

Finally, the detailed design focuses on solving the specific problem of finding the quickest path between the two flags in UT. This illustrates an important feature of the *CHRIS* framework. The ability to use agents in complex environments but only learn about specific aspects of that environment through generalization.

# Chapter 7

## Conclusions

This chapter summarizes the thesis, lists specific contributions and suggests directions for future research.

### 7.1 Review

The interface between an agent and its environment can be simplified into a loop involving agents receiving sensations from the environment, making decisions, and taking actions that affect the environment. Each environment can have a number of features based upon being observable, being deterministic, being episodic, being static, being discrete and being comprised of multiple agents or not. Agent technology has been used in many different application areas where agents are faced with different complex environments exhibiting different properties. Example of such areas include computer games, research, education, defence, and business.

The reasoning performed by agents can be tackled from different angles corresponding to thinking versus acting. Research on rational reasoning is concerned with agents acting as best they can given their limited knowledge. Related research areas include: knowledge representation, expert systems and production systems. Research on human reasoning is concerned with the development of models that describe how humans make decisions. Related research includes: practical reasoning via beliefs, desires and intentions, cognitive systems engineering and the decision ladder for understanding the reasoning process and skill levels of expertise, finally, tactical assessment for reducing the decision cycle into the four stages of observation, orientation, decision and action, as well as feedbacks that speed up the reasoning process.

The concept of learning has many definitions depending on the area that it is being applied to. Learning can be thought as obtaining new skills, altering behaviours, and increasing efficiency. Symbol-based learning relies on learning algorithms that operate on a knowledge base to infer new knowledge. Neural networks are comprised of interconnected artificial neurons, that learn by modifying the structure of the network in order to adapt based on inputs and outputs given. Genetic algorithms focus on the concept of evolution via shaping a population of agents. Reinforcement learning allows computers to learn by trial and error, it uses a value function for actions that is updated according to rewards generated by a reward function. The agent learns by trying to maximize the rewards that are received during operation.

Agents are used in software development to reduce complexity while increasing robustness.

When viewed as software entities, agents inhabit a software environment, they obtain information from the environment by invoking software functions, and can perform software actions such as sending messages on a computer network. There are however, a number of important pitfalls with regards to using agents in real world systems. Political pitfalls refer to people overselling agents for use in real world applications. Management pitfalls refer to using agents without having a clear idea on why they are needed. Conceptual pitfalls refer to believing that agents should always be used. Analysis and design pitfalls refer to not exploiting related mature technologies when developing agents. Agent-level pitfalls refer to the development of new agent platforms even when it is not required. Finally, society level pitfalls are concerned with multi-agent technologies.

Deliberate agents perform rational reasoning by taking actions after deliberating using their knowledge. Conversely, reactive agents reject any symbolic representation of decision making, instead their behaviour emerges from the interaction of various more simple behaviours directly linked to the environment that they occupy. Hybrid agents are capable of expressing both reactive and deliberative behavior.

JACK Intelligent Agents is a development platform for creating practical reasoning agents in the Java language using BDI reasoning constructs. It allows the designer to use all features of the Java as well as a number of specific BDI reasoning extensions. Each agent has beliefs about the world, events to respond reactively, goals that it desires to achieve and plans that define what actions it should perform. JACK agents can exhibit goal-directed behavior, context sensitivity, validation of approach and concurrency.

The decision ladder, the OODA loop and the BDI model described in chapter 3 are complementary and can be fused together to yield a new, hybrid and more detailed conceptual reasoning model. The general structure of the OODA loop is retained, by dividing the decision ladder and placing appropriate components in each of the four stages. The key components of BDI reasoning are also apparent within the new hybrid model. The ability to learn can be achieved by appropriately interfacing an additional learning stage in parallel to the main decision loop. The learning stage effectively replaces the “implicit guidance” link indicated by the OODA loop between the orientation stage and the action stage.

The Cognitive Hybrid Reasoning Intelligent Agent System (*CHRIS*) is an implementation of the conceptual model as an extension to the JACK agent development platform. It provides a framework to create learning agents without the need to worry about how the internal learning algorithms operate. Two simple learning agents have been developed in order to demonstrate the *CHRIS* framework in operation. The first agent solves the classic 10-armed bandit problem and the second agent is a slightly more complex example, where it learns how to play Tic-Tac-Toe and win against a computerized opponent.

The Prometheus design methodology is segmented into three major phases. The specification phase borrows heavily from requirements engineering and aims to understand what exactly is required to implement, this includes a description of learning problems to be solved. The architectural design phase defines additional details regarding the overall structure of agents, this includes the definition of learning descriptors. Finally, the detailed design phase defines how the internal components of agents are implemented. This includes how to observe new information, update beliefs, the goals of the agent, the actions available to the agent, and

finally the required LearningGoals. The agent-oriented development process is demonstrated by describing the design of a system controlling a team players within UT. It then presents the results for a partial implementation where an agent learns the quickest path between the home and enemy flags in UT.

## 7.2 Contributions

The following is a list of the specific contributions made in this thesis.

- Section 5.2 describes how the Decision Ladder, the OODA loop and the BDI model are complementary and can be fused together to yield a new, and more detailed *hybrid reasoning model* shown in appendix B. This model also provides sufficient detail to indicate how learning can be integrated into the reasoning process.
- Section 5.3 introduces the Cognitive Hybrid Reasoning Intelligent Agent System as an *implementation framework* for the new hybrid reasoning model. It provides an abstract way to create learning agents without the need to worry about how the internal learning algorithms operate. This framework also provides the means for a number of other key contributions.
- It is identified that in order to integrate learning into agents, a designer must first explicitly understand the *learning problems* that the agent system is required to solve within the initial stages of the system specification. This does not involve specifying how to implement learning.
- Using the *CHRIS* framework, an entire learning problem can be encapsulated within a single construct, the **LearningGoal**. A designer simply defines a learning problem by extending the **LearningGoal** and is not required to understand how the underlying learning algorithms work, as they have already been implemented in an abstract and portable way. This encapsulation is a major difference to other implementations who leave it up to a designer to implement learning algorithms.
- The identification of a learning episode in the context of agent development. A learning episode begins when a **Target** with an associated **LearningLoal** is activated and it ends when the **Target** is deactivated. A **Target** is an event normally posted by the decision stage for subsequent means-ends reasoning.
- The decision stage of the *CHRIS* framework introduces a contribution with respect to the JACK platform. It allows for an entire goal hierarchy to be defined simply by populating a beliefset. This is important when a moderately large goal hierarchy is required. JACK normally requires the explicit implementation of many events and plans in order to achieve the same effect.
- Although *active learning* is simply an implementation of traditional RL algorithms, *passive learning* is a feature unique to the *CHRIS* framework. It allows an agent to learn the behaviour of a pre-written plan by observing the actions taken by the plan with respect to a given **LearningGoal**.

- **SkillControl** is also a feature unique to the *CHRIS* framework, it allows the agent to switch between active and passive learning based on the *confidence* of its learning.
- The introduction of a **StateInstance** defines the need to capture snapshots of the agent’s entire belief knowledge base for a particular *instance in time*. This is a feature required for learning and is not well understood by designers.
- Learning algorithms use a **Perception** to perform learning. A **Perception** must be implemented to contain a subset of the entire **StateInstance** with respect to a particular learning problem. This allows the associated **LearningGoal** to only consider features that are important for what is required to be learned. In addition, using **Perception** it becomes possible to easily perform *generalization* (of states) or *translation* (of both states and actions) in order to improve the efficiency of learning.
- The action stage of the *CHRIS* framework describes another contribution with respect to the JACK platform by explicitly defining how to implement **Actions**. They are used by the framework to define the smallest unit of behaviour that agents should exhibit in the environment. The advantage of this approach is that complex behaviours are realized through collections of actions that can be changed in real-time. This feature is exploited by the framework to realize dynamic behaviours for learning.
- Section 6.1.1 describes how to extend an agent-oriented design methodology in order to accommodate learning issues. This includes the specification of learning problems, the definitions of learning descriptors. and finally the definition of **LearningGoals**.

## 7.3 Future Directions

Following the advances made in this thesis, a number of questions and avenues for future research arise.

As previously mentioned the *CHRIS* framework is an implementation of the hybrid reasoning model. However, the current version of the framework only implements the main reasoning loop and the learning feedback from the orientation to the action stage. The rest of the feedbacks between the different stages have not been implemented. Of particular interest, would be to investigate how each of the feedbacks can be integrated into the underlying reasoning components. Interesting questions include: what could cause the observation stage to take the heuristic path into the action stage? How does the observation stage use feedbacks being received from the decision and action stages? Interesting future research could also involve extending the *CHRIS* framework with additional learning algorithms that could potentially provide better performance in certain applications.

It has currently been planned that the design described in chapter 6 be fully implemented in order to investigate the many interesting issues raised by the Unreal Tournament environment. Of particular interest, are the issues regarding human-agent teaming. It has recently been suggested that learning will play an important role in achieving good human-agent team performance. One of the goals of future research is the ability of agents to dynamically adapt to the skill level of the human. For example, if a novice human joins the team, then agents



would take on more responsibility. As time passes however, and the human becomes experienced, the agent system automatically assigns more responsibility to the human and provides a supporting role.

Finally, another possible avenue of future research involves porting the *CHRIS* framework to an entirely different agent-development platform in order to investigate the issues involved. It could also be advantageous to consider moving away from proprietary solutions and embrace an open-source platform, or even creating a platform from scratch based on the hybrid reasoning model.

## 7.4 Concluding Remarks

This thesis describes how to integrate learning into agents via an analysis of previous research on reasoning. It begins by justifying why learning is actually required and then moves on to describe the different types of reasoning models and learning techniques. It then delves into the theory behind agents and provides a comprehensive overview of different agent types, and their associated development platforms. The thesis then makes a number of contributions by combining the theory presented previously into a new hybrid reasoning model and resulting implementation framework. The thesis concludes with describing how learning can be included in an agent-oriented design methodology such that it forms part of the entire development process.

# List of References

- [1] Gregory D. Abowd. Ieee pervasive computing: Introduction - the human experience. *IEEE Distributed Systems Online*, 4(6), 2003.
- [2] Activision. Star trek armada. [Online Accessed: 9th July 2005] <http://www.activision.com>, July 2005.
- [3] Rogelio Adobbati, Andrew N. Marshall, Andrew Scholer, Sheila Tejada, Gal Kaminka, Steven Schaffer, and Chris Sollitto. Gamebots: A 3d virtual world test-bed for multi-agent research. In *Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS and Scalable MAS, June, Montreal Canada*. AAAI Press, California, 2001.
- [4] Rogelio Adobbati, Andrew N. Marshall, Andrew Scholer, Sheila Tejada, Gal Kaminka, Steven Schaffer, and Chris Sollitto. Gamebots network api. [Online Accessed, 17 August 2005] <http://www.planetunreal.com/gamebots/docapi.html>, 2001.
- [5] AgentOrientedSoftware. Jack intelligent agents agent manual. [Online, accessed 6th June] <http://www.agent-software.com.au/shared/resources/index.html>, June 2005.
- [6] Christian Andersson, Anna Edberg, Claire Hennekam, Jason Khallouf, Lin Padgham, Mikhail Pereplechikov, Aman Sahani, John Thangarajah, and Michael Winikoff. Prometheus design tool. [Online Accessed: 2 September 2005] <http://www.cs.rmit.edu.au/agents/pdt/>, September 2005.
- [7] Sachio Arai, Katia Sycara, and Terry R Payne. Experience-based reinforcement learning to acquire effective behavior in a multiagent domain. In *proceedings of the Sixth Pacific Rim International Conference on Artificial Intelligence (PRICAI 2000)*, pages 125–135. Springer-Verlag, Berlin, Germany, 2000.
- [8] Sachiyo Arai and Katia Sycara. Effective learning approach for planning and scheduling in multi-agent domain. In *Proceedings of the 6th International*, pages 507–516, 2000.
- [9] Michael Baker, Tia Hanser, Richard Joiner, and David Traum. The role of grounding in collaborative learning tasks. *Collaborative learning. Cognitive and computational approaches*, Elsevier, Oxford, UK, pages 31–63, 1999.
- [10] Jeremy Baxter and Richard Hepplewhite. Agents in tank battle simulations. *Communications of the ACM, ACM Press*, 42(3):74–75, 1999.

- [11] Jeremy Baxter and Richard Hepplewhite. A hierarchical distributed planning framework for simulated battlefield entities. In *Proceedings of 19th Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG 2000)*, 2000.
- [12] Joseph P. Bigus, Don A. Schlosnagle, Jeff R. Pilgrim, W. Nathaniel Mills III, and Yixin Diao. Able: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002.
- [13] Blizzard. Starcraft. [Online Accessed: 9th July 2005] <http://www.blizzard.com/starcraft/>, July 2005.
- [14] John R Boyd. The essence of winning and losing. [Online Accessed, 17 August 2005] [http://www.belisarius.com/modern.business.strategy/boyd/essence/eowl\\_frameset.htm](http://www.belisarius.com/modern.business.strategy/boyd/essence/eowl_frameset.htm), August 2005.
- [15] Michael E. Bratman. *Intention, Plans, and Practical Reason*. Center for the Study of Language and Information, 1999.
- [16] Rodney A Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.
- [17] Chris Brown, Peter Barnum, Dave Costello, George Ferguson, Bo Hu, and Mike Van Wie. Quake ii as a robotic and multi-agent platform. Technical Report 853, University of Rochester, 2004.
- [18] Michael Buro. Real-time strategy games: A new ai research challenge. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15*, pages 1534–1535. Morgan Kaufmann, 2003.
- [19] Michael Buro and Timothy M Furtak. Rts games and realtime ai research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS 04)*, Arlington VA, 2004.
- [20] Todd J. Callantine. Air traffic controller agents. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), 19-23 August 2004, New York, NY, USA*, pages 952–953. IEEE Computer Society, 2003.
- [21] Kumar Chellapilla and David B. Fogel. Evolving an expert checkers playing program without using human expertise. *IEEE Trans. Evolutionary Computation*, 5(4):422–428, 2001.
- [22] S M Clemens. *The one with the most information wins? The quest for information superiority*. Master of science in information resource management thesis, Airforce Institute of Technology, 1997. p 128.
- [23] Oshkosh Truck Co. and Ohio State University. Technical paper for terramax. Technical report, Oshkosh Truck Co. and Ohio State University, 2004.

- [24] MSL Technical Committee. Middle size robot league, rules and regulations for 2005. Technical report, RoboCup, 2005.
- [25] R Connell, F Lui, D.Jarvis, and M Watson. The mapping of courses of action derived from cognitive work analysis to agent behaviours. In *Agents At Work Porkshop, Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003), July 14-18, Melbourne, Victoria, Australia*. ACM Press, New York, 2003.
- [26] Diane J. Cook, Michael Youngblood, Edwin O.Heierman, Karthik Gopalratnam, Sira Rao, Andrey Litvin, and Farhan Khawaja. Mavhome: An agent-based smart home. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications*, pages 521–524, 2003.
- [27] Lockheed Martin Corporation. Onesaf testbed baseline assessment. Technical report, Simulation, Training & Instrumentation Command, 1998.
- [28] Isaac G. Councill, Geoffrey P. Morgan, and Frank E. Ritter. dtank: A competitive environment for distributed agents. Technical report, School of Information Sciences and Technology, The Pennsylvania state University, March 2004.
- [29] DARPA. Darpa grand challenge 2005 rules. Technical report, Defence Advanced Research Projects Agency, October 2004.
- [30] Defence and National Interest. The ooda loop. [Online accessed, 18 August 2005] [http://www.d-n-i.net/fcs/ppt/boyds\\_ooda\\_loop.ppt](http://www.d-n-i.net/fcs/ppt/boyds_ooda_loop.ppt), August 2005.
- [31] Pierre Dillenbourg. What do you mean by collaborative learning? *Collaborative learning. Cognitive and computational approaches*, Elsevier, Oxford, UK, pages 1–20, 1999.
- [32] K R Dixon, R J Malak, and P K Khosla. Incorporating prior knowledge and previously learned information into reinforcement learning agents. Technical report, Institute for Complex Engineered Systems, Carnegie Mellon University, 2000.
- [33] Alexis Drogoul. When ants play chess (or can strategies emerge from tactical behaviours?). In Cristiano Castelfranchi and Jean-Pierre Müller, editors, *From Reaction to Cognition, 5th European Workshop on Modelling Autonomous Agents, MAAMAW '93, Neuchatel, Switzerland, August 25-27, Selected Papers*, volume 957 of *Lecture Notes in Computer Science*, pages 13–27. Springer-Verlag, Berlin, 1995.
- [34] Robert Englemore, Tony Morgan, and H Penny Nii. *Blackboard Systems*, chapter Introduction, pages 1–24. Addison-Wesley, England, 1988.
- [35] InfoGrames Epic Games and Digital Entertainment. Unreal tournament manual, 2000.
- [36] Richard Evans. Ai in games: A personal view. [Online accessed: 10 July 2005] <http://www.gameai.com/blackandwhite.html>, July 2005.

- [37] Jacques Ferber, Olivier Gutkecht, and Fabien Michel. Madkit development guide. [Online accessed: 25 August 2005] <http://www.madkit.org/madkit/doc/devguide/devguide.html>, 2005.
- [38] R E Fields, P C Wright, P Marti, and M Palmonari. Air traffic control as a distributed cognitive system: a study of external representations. In T R G Green, L Bannon, C P Warren, and J Buckley, editors, *Proceedings of the 9th European Conference on Cognitive Ergonomics*. University of Limerick, Ireland, 1998.
- [39] Free Software Foundation. Gnu general public license. [Online accessed 13th July 2005]: <http://www.gnu.org/licenses/gpl.html>, July 1991.
- [40] Stuart D Fowell and Roger Ward. The role of software agents in space operations. In *Proceedings of the 2002 Conference on Space Operations (SpaceOps 2002)*, on CD, 2002.
- [41] cois Félix Ingrand Fran Raja Chatila, Rachid Alami, and Frédéric Robert. Prs: A high level supervision and control language for autonomous mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation, Minneapolis, USA*, volume 1, pages 43–49. IEEE Press, 1996.
- [42] Matthew Freeland, Hasnah Mat-Amin, Khemanut Teangtrong, Wichan Wannalertsri, and Uraiporn Wattanakasemsakul. Pervasive computing: business opportunity and challenges. In *Portland International Conference on Management of Engineering and Technology*, volume 1, page 85. IEEE Press, 2001.
- [43] Joseph C Giarratano. Clips user guide. Technical report, Public Domain, 2002.
- [44] GraphViz. Graph visualization software. [Online Accessed, 17 August 2005] <http://www.graphviz.org>, 2005.
- [45] Object Management Group. Common object request broker architecture: Core specification. Technical report, Object Management Group, 2004.
- [46] Swarm Development Group. Documentation set for swarm 2.2. [Online Accessed: 25 August 2005] <http://www.swarm.org/swarmdocs-2.2/set/set.html>, August 2005.
- [47] Grant Tedrick Hammond. *The Mind of War: John Boyd and American Security*. Smithsonian Institution Press: Washington, USA., 2004.
- [48] Nicholas Hanssens, Ajay Kulkarni, Rattapoom Tuchida, and Tyler Horton. Building agent-based intelligent workspaces. In *International Conference on Internet Computing*, pages 675–681, 2002.
- [49] Mance E Harmon and Stepanie S Harmon. Reinforcement learning: A tutorial. Technical report, Wright Laboratory and Wright State University, 1997.
- [50] Fredrik Heintz and Patrick Doherty. A knowledge processing middleware framework and its relation to the jdl data fusion model. In *Proceedings of the Swedish Artificial Intelligence and Learning Systems Workshop (SAIS-SSLS 05)*, Vsters, Sweden, pages 68–77, 2005.

- [51] Clinton Heinze, Simon Goss, and Adrian Pearce. Plan recognition in military simulation: Incorporating machine learning with intelligent agents. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, Workshop on Team Behaviour and Plan Recognition*, pages 53–63, Stockholm, Sweden, 1999. ACM Press, New York.
- [52] Henry Hexmoor and Tim Heng. Air traffic control and alert agent. In *Proceedings of the fourth international conference on Autonomous agents, Barcelona, Spain*, pages 237–238. ACM Press, New York, 2000.
- [53] Feng-Hsiung Hsu. *Behind Deep Blue : Building the Computer that Defeated the World Chess Champion*. Princeton University Press, 2002.
- [54] Marcus J. Huber. *JAM Agents in a Nutshell*, 2001.
- [55] Marcus J. Huber. *UMPRS Manual*, 2001.
- [56] Nathan Huff, Ahmed Kamel, and Kendall Nygard. An agent based framework for modelling uav’s. In *Proceedings of the 16th International Conference on Computer Applications in Industry and Engineering (CAINE 2003)*. ISCA Press, Las Vegas, 2003.
- [57] Intelligent Automation Inc. Cybelepro agent infrastructure, users guide. Technical report, Intelligent Automation Inc., 2004.
- [58] Vincent Joyau. Airfight 2000. [Online, accessed 26 July 2005] URL:<http://www.planetunreal.com/airfight/>, July 2005.
- [59] Samin Karim and Clinton Heinze. Experiences with the design and implementation of an agent-based autonomous uav controller. In *AAMAS Industrial Applications, Proceedings of the fourth international conference on autonomous agents and multiagent systems (AAMAS 2005)*, pages 19–26. ACM Press, New York, 2005.
- [60] Sandia National Laboratories. Jess the rule engine for the java platform. [Online, accessed 5 July 2005] <http://herzberg.ca.sandia.gov/jess/>, July 2005.
- [61] Sandia National Laboratories. Jess, the rule engine for the java platform. [Online Accessed: 25 August 2005] <http://herzberg.ca.sandia.gov/jess/docs/index.shtml>, August 2005.
- [62] John E Laird and Clare Bates Congdon. The soar user’s manual. Technical report, Electrical Engineering and Computer Science Department, University of Michigan, June 2004.
- [63] Danny B Lange. Java aglet application programming interface (j-aapi) white paper - draft 2. Technical report, IBM Tokyo Research Laboratory, 1997.
- [64] Francois-Dominic Laramee. Chess programming. [Online, accessed 12 July 2005] <http://www.gamedev.net/reference/programming/features/chess1/>, 2005.

- [65] George Lee, Peyman Faratin, Steven Bauer, and John Wroclawski. A user-guided cognitive agent for network service selection in pervasive computing environments. In *PerCom, Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 2004), 14-17 March 2004, Orlando, FL, USA*, pages 219–228. IEEE Computer Society, 2004.
- [66] LionHead. Black and white. [Online, Accessed 10 July 2005] <http://www.lionhead.com/>, 2005.
- [67] James Llinas, Christopher L. Bowman, Galina L. Rogova, Alan N. Steinberg, Edward L. Waltz, and Frank E. White. Revisiting to the jdl data fusion model ii. In Per Svensson and Johan Schubert, editors, *Proceedings of the Seventh International Conference on Information Fusion*, volume II, pages 1218–1230, Mountain View, CA, Jun 2004. International Society of Information Fusion.
- [68] George F Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Pearson Education Limited, Harlow, England, 4th edition, 2002.
- [69] Frank Lui and Marcus Watson. Mapping cognitive work analysis to an intelligent agents software architecture: Command agents. In *Defence Factors Special Interest Group (DHFSIG) conference 2003, DSTO Fisherman’s Bend, Melbourne*, 2003.
- [70] Andrew N Marshall. Javabot for unreal tournament. [Online Accessed, 17 August 2005] <http://ubot.sourceforge.net>, 2005.
- [71] C. Martin, Debra Schreckenghost, R. Peter Bonasso, David Kortenkamp, Tod Milam, and Carroll Thronesbery. An environment for distributed collaboration among humans and software agents. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003), July 14-18, Melbourne, Victoria, Australia*, pages 1062–1063. ACM Press, New York, 2003.
- [72] James Mayfield, Yannis Labrou, and Timothy W. Finin. Evaluation of kqml as an agent communication language. In Michael Wooldridge, Jörg P. Müller, and Milind Tambe, editors, *Intelligent Agents II, Agent Theories, Architectures, and Languages, IJCAI ’95, Workshop (ATAL), Montreal, Canada, August 19-20, Proceedings*, volume 1037 of *Lecture Notes in Computer Science*, pages 347–360. Springer Verlag, Berlin, August 1995.
- [73] M McNaughton, J Schaeffer, D Szafron, D Parker, and J Redford. Code generation for ai scripting in computer role-playing games. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Workshop on Challenges in Game AI*, pages 129–133. The AAAI Press, California, 2004.
- [74] Sid Meyers. Civilisation. [Online Accessed: 7th July 2005] <http://www.civ3.com/>, July 2005.
- [75] Alexander M Meystel. Chapter 10. learning. Technical report, Drexel University, 2001.

- [76] Microsoft. Age of empires. [Online Accessed: 9th July 2005]  
<http://www.microsoft.com/games/empires/>, July 2005.
- [77] Geoffrey P. Morgan, Frank E. Ritter, and Mark A. Cohen. dtank: An environment for architectural comparisons of competitive agents. In L Allender and T Kelley, editors, *Proceedings of the 14th Conference on Behaviour Representation in Modelling and Simulation (BRIMS 2005), Universal City, CA*, pages 133–140. University of Central Florida, 2005.
- [78] X T Nguyen. Threat assessment in tactical airborne environments. In *Proceedings of the Fifth International Conference on Information Fusion*, volume 2, pages 1300–1307. IEEE Press, 2002.
- [79] Emma Norling. Learning to notice: Adaptive models of human operators. In D Precup and P Stone, editors, *In Second International Workshop on Learning Agents (Agents-2001), Montreal Canada, Montreal, Canada.*, May 2001. ACM Press, New York.
- [80] Emma Norling. Capturing the quake player using a bdi agent to model human behaviour. In S Jeffrey, J S Rosenschein, T Sandholm, M Wooldridge, and M Yokoo, editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 1080–1081, Melbourne, Australia, 2003. ACM Press, New York.
- [81] Stephen Ostermiller. Tic-tac-toe strategy. [Online, accessed 3 June 2005]  
<http://ostermiller.org/tictactoeexpert.html>, June 2005.
- [82] M G Oxenham. Enhancing situation awareness for air defence via automated threat analysis. In *Proceedings of the Sixth International Conference of Information Fusion, Radisson Hotel, Cairns, Australia*, volume 2, pages 1086–1093. IEEE Press, 2003.
- [83] Lin Padgham and Michael Winikoff. *Developing Intelligent Agent Systems A Practical Guide*. John Wiley and Sons Ltd., West Sussex, England, 2004.
- [84] Ernest H. Page and Roger Smith. Introduction to military training simulation: A guide for discrete event simulationists. In D J Medeiros, E F Watson, J S Carson, and M S Manivannan, editors, *Winter Simulation Conference*, pages 53–60. IEEE Press, 1998.
- [85] Adrian R. Pearce and Terry Caelli. The claret algorithm. In *Research and Development in Knowledge Discovery and Data Mining, Lecture Notes in Artificial Intelligence*, volume 1394, pages 407–408. Springer-Verlag, Berlin, 1998.
- [86] A Polychronopoulos, M Tsogas, A Amditis, A Scheunert, L Andreone, and F Tango. Dynamic situation and threat assessment for collision warning systems: the euclidean approach. In *Proceedings of the Intelligent Vehicles Symposium (IV'04), June 14-17, Parma, Italy*, pages 636–641. IEEE Press, 2004.
- [87] Pole Star Publications. Computer 'mobile agents' and robot tested by nasa. SpaceFlight Now [Online, accessed 1 July 2005]  
<http://spaceflightnow.com/news/n0405/02mobileagent/>, 2005.



- [88] J Rasmussen, AM Pejtersen, and LP Goodstein. *Cognitive Systems Engineering*. Wiley and Sons, New York, NY, 1994.
- [89] CarnegieMellon University Red Team Racing. Navigation and sensing. [Online accessed: 29 June 2005] <http://www.redteamracing.org/index.cfm?Method=page.display&page=technology.navigationAndSensing>, June 2005.
- [90] Brandon Reinhart. Mod authoring for unreal tournament. Technical report, Epic Games, Inc., 1999.
- [91] RoboCup. Humanoid kid and medium size league, rules and setup for osaka 2005. Technical report, RoboCup, 2005.
- [92] RoboCup. Laws of the f180 league 2005a. Technical report, RoboCup, 2005.
- [93] RoboCup. Robocup official website. [Online accessed: 13 July 2005] <http://www.robocup.org>, July 2005.
- [94] Wendy A Rogers and Elizabeth D Mynatt. How can technology contribute to the quality of life of older adults. In *M. E. Mitchell (Ed.), The Technology of Humanity: Can Technology contribute to the quality of life? Chicago. IL: Institute of Psychology and Institute for Science, Law, and Technology, Illinois Institute*, pages 22–30, 2003.
- [95] Stuart Russell and Peter Norvig. *Artificial Intelligence A Modern Approach*. Prentice Hall, New Jersey, 2nd edition, 2003.
- [96] A L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:210–229, 1959.
- [97] Jennifer Sandercock, Michael Papasimeon, and Clint Heinze. An agent a bot and a cgf walk into a bar... In *Proceedings of the 9th Simulation Technology and Training Conference (SimTecT '04) Canberra Australia*. on cdrom, Simulation Industry Association of Australia, 2004.
- [98] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53, Number 2-3:273–290, 1992.
- [99] Brad Shapcott. Fragball for ut. [Online, accessed 26 July 2005] URL:<http://www.planetunreal.com/fragball/>, July 2005.
- [100] Christos Sioutis, Nikhil Ichalkaranje, and Lakhmi Jain. A framework for interfacing bdi agents to a real-time simulated environment. In Ajith Abraham, Mario Koppen, and Katrin Franke, editors, *Design and Application of Hybrid Intelligent Systems*, Frontiers in Artificial Intelligence and Applications, pages 743–748. IOS Press, Amsterdam, The Netherlands, 2003.
- [101] Agent Oriented Software. A01: Introduction to agent oriented systems and intelligent agents. PowerPoint Presentation, 2002.

- [102] Sony. Aibo your artificial intelligence companion, product brochure. [Online, accessed 21 July 2005] [http://www.eu.aibo.com/1\\_2\\_library.asp](http://www.eu.aibo.com/1_2_library.asp), 2005.
- [103] Peter Stone. *Layered Learning in Multiagent Systems*. The MIT Press, Massachusetts, London, 1998.
- [104] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning An Introduction*. The MIT Press, Massachusetts, London, 2000.
- [105] K Sycara, J A Giampapa, B K Langley, and M Paolucci. The retsina mas, a case study. *Software Engineering for Large-Scale Multi-Agent Systems: Research Issues and Practical Applications*, LNCS 2603:232–250, July 2003.
- [106] Omar Syed and Aamir Syed. Arimaa - the game of real intelligence. [Online, accessed 13 July 2005] <http://arimaa.com/arimaa/>, July 2005.
- [107] Omar Syed and Aamir Syed. The arimaa challenge. [Online, accessed 19 July 2005] <http://arimaa.com/arimaa/challenge/>, July 2005.
- [108] Omar Syed and Aamir Syed. Arimaa game rules. [Online, accessed 13 July 2005] <http://arimaa.com/arimaa/learn/rulesIntro.html>, July 2005.
- [109] BBN Technologies. Cougaar architecture document. Technical report, A BBN Technologies, 2004.
- [110] Pierre Urlings. *Teaming Human and Machine*. PhD thesis, School of Electrical and Information Engineering, University of South Australia, 2003.
- [111] Pierre Urlings, Jeffrey Tweedale, Christos Sioutis, and Nikhil Ichalkaranje. Intelligent agents and situation awareness. In *Proceedings of the 7th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES 2003)*, 3-5 September, Oxford, United Kingdom, pages 723–733. Springer-Verlag, Berlin, Germany, 2003.
- [112] Gerhard Wei and Pierre Dillenbourg. What is ‘multi’ in multiagent learning? *Collaborative learning. Cognitive and computational approaches*, Elsevier, Oxford, UK, pages 64–80, 1999.
- [113] Steven Wollkind, John Valasek, and Thomas R. Ioerger. Automated conflict resolution for air traffic management using cooperative multiagent negotiation. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference. Aug, 16-19, Providence, RI.*, pages 16–19. American Institute of Aeronautics and Astronautics (AIAA), 2004.
- [114] Michael Wooldridge. *Multiagent systems: a modern approach to distributed artificial intelligence*, chapter Intelligent Agents, pages 27–78. MIT Press, 1999.
- [115] Michael Wooldridge. *Reasoning About Rational Agents*. The MIT Press, Massachusetts, London, 2000.

- [116] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley and Sons Ltd, Chichester, 2002.
- [117] Michael Wooldridge, Nicholas R Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. In *Proceedings of the conference on Autonomous Agents and Multi-Agent Systems*, volume 3, pages 285–312. Kluwer Academic Publishers, The Netherlands, 2000.
- [118] Michael Wooldridge and Nick Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, Cambridge University Press, 10(2):115–152, 1995.

# Appendix A

## The Gamebots Protocol

This information was obtained from the Gamebots API documentation [4].

### A.1 Synchronous Messages

**BEG** Beginning of a synchronous batch

**Time** Timestamp from the game

**SLF** Information about the bot's state

**Id** A unique id, assigned by the game

**Rotation** Which direction the player is facing in absolute terms

**Location** An absolute location

**Velocity** Absolute velocity in UT units

**Name** Players human readable name

**Team** What team the player is on. 255 is no team. 0-3 are red, blue, green, gold in that order

**Health** How much health the bot has left. Starts at 100, ranges from 0 to 200.

**Weapon** Weapon type the player is holding

**Currentammo** How much ammo the bot has left for current weapon

**Armor** How much armor the bot is wearing. Starts at 0, can range up to 200.

**GAM** Information about the game

**Playerscores** Player score will have a list of values - one for each player in the game

**Teamscores** Like playerscore, but for teams, a team is identified by the team index. This argument is not sent in deathmatch games.

**Dompoints** Also a multi-valued message. There is one item for each domination point in a Domination game. First value will be

**Id** of the DOM point, the second will be the index of the team that owns the domination point.

**Haveflag** Sent in CTF games if the bot is carrying an enemy's flag. Value is the team number of whose flag you have.

**Enemyhasflag** Sent in CTF games if the bot's team's flag has been stolen. Value is meaningless.

**PLR** Another character (bot or human) in the game. Only reports those players that are visible. (within field of view and not occluded).

**Id** A unique id for this player, assigned by the game

**Rotation** Which direction the player is facing in absolute terms

**Location** An absolute location for the player

**Velocity** Absolute velocity in UT units

**Team** What team the player is on

**Reachable** True if the bot can run to this other player directly, false otherwise. Possible reasons for false: pit or obstacle between the two characters

**Weapon** What class of weapon the character is holding

**NAV** A path node in the game. Pathnodes are invisible (at least to humans) objects placed around a level to define paths for the built in bots to follow. They provide a totally connected graph that spans almost all of the level. Note the Mutator called "Path Markers" that, when added to a game makes the path nodes visible to human players as a debugging aid.

**Id** A unique id for this pathnode, assigned by the game

**Location** An absolute location

**Reachable** True if the bot can run here directly, false otherwise

**MOV** A "mover". These can be doors, elevators, or any other chunk of architecture that can move. They generally need to be either run into, or activated by shooting or pressing a button. We are working on ways to provide bots with more of the information they need to deal with movers appropriately.

<b>Id</b>	A unique id for this mover, assigned by the game	<b>Team</b>	The team whose flag this is
<b>Location</b>	An absolute location	<b>Reachable</b>	True if the bot can run here directly, false otherwise
<b>Reachable</b>	True if the bot can run here mover, false otherwise	<b>State</b>	Whether the flag is "Held" "Dropped" or "Home"
<b>Damagetrig</b>	True if the mover needs to be shot to activated	<b>INV</b>	An object on the ground that can be picked up
<b>Class</b>	Class of the mover	<b>Id</b>	A unique id for this inventory item, assigned by the game
<b>DOM</b>	Identical attributes to NAV above except for Controller (see below). A domination point in a domination game	<b>Location</b>	An absolute location
<b>Controller</b>	Which team controls this point	<b>Reachable</b>	True if the bot can run here directly, false otherwise
<b>FLG</b>	A flag. (Only for CTF games).	<b>Class</b>	A string representing type of object
<b>Id</b>	A unique id for this flag, assigned by the game	<b>END</b>	End of a synchronous batch
<b>Location</b>	An absolute location of the flag	<b>Time</b>	Timestamp from the game
<b>Holder</b>	The identity of player/bot holding the flag (only sent if flag is being carried).		

## A.2 Asynchronous Messages

<b>NFO</b>	Helpful info about the game provided right after connection made to server. Your client should wait for this message BEFORE trying to send "init" back to the server.	<b>VMG</b>	Received tokenized message from another player. If you want to use these, enter the game as a player and use the voice menu (by default press the key "V" while playing). Send the messages you want to use to use and have a client log the incoming messages to figure out Types and Ids.
<b>Gametype</b>	What you are playing (botdeathmatch-plus, botteamgame, botdomination)	<b>Sender</b>	Unique id of player who sent message
<b>Level</b>	Name of map in play	<b>Type</b>	Type of message (e.g. Command, Taunt, etc...)
<b>Timelimit</b>	Maximum time game will last (if tied at end, goes into sudden death overtime)	<b>Id</b>	Message id. Specifies which message is being sent
<b>Fraglimit</b>	Number of kills needed to win game (bot-deathmatchplus only)	<b>ZCF</b>	Foot changed zones. Feet of bot changed from one artificial area in the game to another (can tell you when entered water or lava or some such)
<b>Goalteamscore</b>	Number of points a team needs to win game (botteamgame, botdomination)	<b>Id</b>	Unique id of zone entered
<b>Maxteams</b>	Max number of teams. Valid team range will be 0 to (maxteams - 1) (botteamgame, botdomination)	<b>ZCH</b>	Head changed zones. Its ok if feet are under water, but having your head under can mean trouble...
<b>Maxteamsize</b>	Max number of players per side (bot-teamgame, botdomination)	<b>Id</b>	Unique id of zone entered
<b>AIN</b>	Added inventory. Bot got new inventory item	<b>ZCB</b>	Bot changed zones. Entire bot now in new zone
<b>Id</b>	A unique id for this inventory item, assigned by the game. Unique, but based on a string describing the item type.	<b>Id</b>	Unique id of zone entered
<b>Class</b>	A string representing type of object	<b>CWP</b>	Bot changed weapons. Possibly as a result of a command sent by you, maybe just because it ran out of ammo in its old gun. (bots auto switch when empty, just like human players)
<b>VMS</b>	Received message from global chat channel	<b>Id</b>	Unique id of new weapon, based on the weapon's name
<b>String</b>	A human readable message sent by another player in the game on the global channel	<b>Class</b>	A string representing type of weapon
<b>VMT</b>	Received message from global chat channel	<b>WAL</b>	Collided with a wall. Note it is common to get a bunch of these when you try to run through a wall (or are pushed into one by gunfire or something).
<b>String</b>	A human readable message sent by a team mate in the game on the private team channel	<b>Id</b>	Unique id of wall hit

<b>Normal</b>	Normal of the angle bot collided at.	<b>KIL</b>	Some other player died
<b>Location</b>	Absolute location of bot at time of impact	<b>Id</b>	Unique ID of player
<b>FAL</b>	Bot just hit a ledge. If walking, will not fall. If running, you are already falling by the time you get this.	<b>Killer</b>	Unique ID of player that killed them if any (may have walked off a ledge)
<b>Fell</b>	True if you fell. False if you stopped at edge.	<b>Damagetype</b>	A string describing what kind of damage killed them
<b>Location</b>	Absolute location of bot	<b>DIE</b>	This bot died
<b>BMP</b>	Bumped another actor	<b>Killer</b>	Unique ID of player that killed them if any (may have walked off a ledge)
<b>Id</b>	Unique id of actor (actors include other players and other physical objects that can block your path.)	<b>Damagetype</b>	A string describing what kind of damage killed them
<b>Location</b>	Location of thing you rammed	<b>DAM</b>	Took damage
<b>HRP</b>	Hear pickup. You head someone pick up an object from the ground	<b>Damage</b>	Amount of damage taken
<b>Player</b>	Unique ID of player how picked up the object	<b>Damagetype</b>	A string describing what kind of damage
<b>HRN</b>	Hear noise. Maybe another player walking or shooting, maybe a bullet hitting the floor, or just a nearby lift going up or down.	<b>HIT</b>	Hurt another player. Hit them with a shot
<b>Source</b>	Unique ID of actor making the noise	<b>Id</b>	Unique ID of player hit
<b>SEE</b>	See player. A message generated by the engine periodically (on the order of 1 or 2 times a second) when another player is visible by you. Possibly useful if you have the delay between synchronous updates very long. In that case, this can prevent someone from walking by unseen. May be deprecated.	<b>Damage</b>	Amount of damage done
<b>Id</b>	A unique id for this player, assigned by the game	<b>Damagetype</b>	A string describing what kind of damage
<b>Rotation</b>	Which direction the player is facing in absolute terms	<b>PTH</b>	A series of pathnodes in response to a get path call from client
<b>Location</b>	An absolute location for the player	<b>Id</b>	An id matching the one sent by client. Allows bot to match answer with right query.
<b>Velocity</b>	Absolute velocity in UT units	<b>Multiple pathnodes</b>	A variable number of attribute items will be returned, one for each pathnode that needs to be taken. They will be listed in the order in which they should be traveled to. Each one is of form "0 id 3,4,5", with the number of the node (starting with 0) followed by a space, then a unique id for the node (will never have a space) then a location of that node.
<b>Team</b>	What team the player is on.	<b>RCH</b>	A boolean result of a check reach call.
<b>Reachable</b>	True if the bot can run to this other player directly, false otherwise. Possible reasons for false: pit or obstacle between the two characters	<b>Id</b>	An id matching the one sent by client. Allows bot to match answer with right query.
<b>Weapon</b>	What weapon the character is holding.	<b>Reachable</b>	True if the bot can run here directly, false otherwise
<b>PRJ</b>	Incoming projectile likely to hit you. May give you a chance to dodge.	<b>From</b>	Exact location of bot at time of check
<b>Time</b>	Estimated time till impact	<b>FIN</b>	No attributes. Sent when game is over
<b>Direction</b>	Rotation value that the projectile is coming from. Best chance to dodge is to probably head off at a rotation normal to this one (add 16000 to the yaw value)		

## A.3 Commands

<b>INIT</b>	Message you send to spawn a bot in the game world. You must send this message before you have a character to play in the game. DO NOT SEND UNTIL YOU RECEIVE NFO MESSAGE FROM SERVER.	<b>Name</b>	Desired name. If in use already or argument not provided, one will be provided for you.
		<b>Team</b>	Preferred team. If illegal or no team provided, one will be provided for you. Normally a team

game has team 0 and team 1. In botdeathmatch-plus, team is meaningless, but this will still set you skin colour to match what you select.

**SETWALK** Set whether you are walking or running (default is run). Note that walking only applies to RUNTO command. STRAFETO always moves at run speed.

**Walk** Send "True" to go into walk mode - you move at about 1/3 normal speed, make less noise, and won't fall off ledges. Send anything else to run.

**STOP** Stop all movement/rotation

**JUMP** Causes bot to jump. Not very useful yet, working on this one.

**RUNTO** Turn towards and move directly to your destination. May specify destination via either Target or Location argument, will be parsed in that order. (i.e. If Target provided, Location will be ignored). If you select an impossible place to head to, you will start off directly towards it until you hit a wall, fall off a cliff, or otherwise discover the offending obstacle.

**Target** The unique id of a player or object or nav point. The object must be visible to you when the command is received or your bot will do nothing. Note that something that was just visible may not be when the command is received, therefore it is recommended you supply a Location instead of a Target.

**Location** Location you want to go to. May be provided as space or comma delimited. ("40 50 45" or "40,50,45"). May also be provided as three separate argument value pairs, one each for X Y and Z ("X 40 Y 50 Z 45").

**STRAFE** Like RUNTO, but you move towards a destination while facing another point/object.

**Location** Location you want to go to. May be provided as space or comma delimited. ("40 50 45" or "40,50,45"). May also be provided as three separate argument value pairs, one each for X Y and Z ("X 40 Y 50 Z 45").

**Target** The unique id of a player or object or nav point that you want to face while moving. Must be visible to you currently.

**Focus** A location value of where to face while moving. Follows same rules as location for what to send. Used only if no Target.

**TURNTO** Specify a point, rotation value or object to turn towards.

**Target** The unique id of a player or object or nav point that you want to face. Must be visible.

**Rotation** Rotation you want to spin to. May be provided as space or comma delimited. ("0 50000 0" or "0,50000,0") and should be in absolute terms and in UT units ( $2\pi = 65535$  units). May also be provided as three separate argument value pairs, one each for Pitch Yaw and Roll ("Pitch 0 Yaw 50000 Roll 0"). Used only if no target provided.

**Location** Location you want to face. Normal rules for location. Only used if no Target or Rotation.

**ROTATE** Turn a specified amount.

**Amount** Amount in UT units to rotate. May be negative to rotate counter clockwise.

**Axis** If provided as Vertical, rotation will be done to Pitch. Any other value, or not provided, and rotation will be to Yaw.

**SHOOT** Start firing your weapon

**Location** Location you want to shoot at. Normal rules for a location specification.

**Target** The unique id of your target. If you specify a target, and it is visible to you, the server will provide aim correction and target leading for you. If not you just shoot at the location specified. Note you still must provide location.

**Alt** If you send True to this you will alt fire instead of normal fire. For normal fire you don't need to send this argument at all.

**CHANGEWEAPON** Start firing your weapon

**Id** Unique Id of weapon to switch to. If you just send "Best" as Id, the server will pick your best weapon that still has ammo for you. Obtain Unique Id's from AIN events.

**STOPSHOOT** Stop firing your weapon

**CHECKREACH** Check to see if you can move directly to a destination without hitting an obstruction, falling in a pit, etc...

**Target** The unique id of a player or object or nav point. Must be visible.

**Location** Location you want to go to. Normal location rules. Only used if no Target is sent.

**Id** Message id made up by you and echoed in response so you can match up response with query

**From** Exact location of bot at time of check

**GETPATH** Get a path to a specified location. An ordered list of path nodes will be returned to you.

**Location** Location you want to go to. Normal location rules.

**Id** Message id made up by you and echoed in response so you can match up response with query

**MESSAGE** Send a message to the world or just your team. This will likely have some restrictions placed on it soon.

**String** String to send.

**Global** If True it is sent to everyone. Otherwise (or if not specified), just your team.

**PING** If for some reason 10 updates a second or whatever your default is isn't frequent enough connection detection for your tastes, use PING. Server will return "PONG".

# Appendix B

## Reasoning and Learning Model

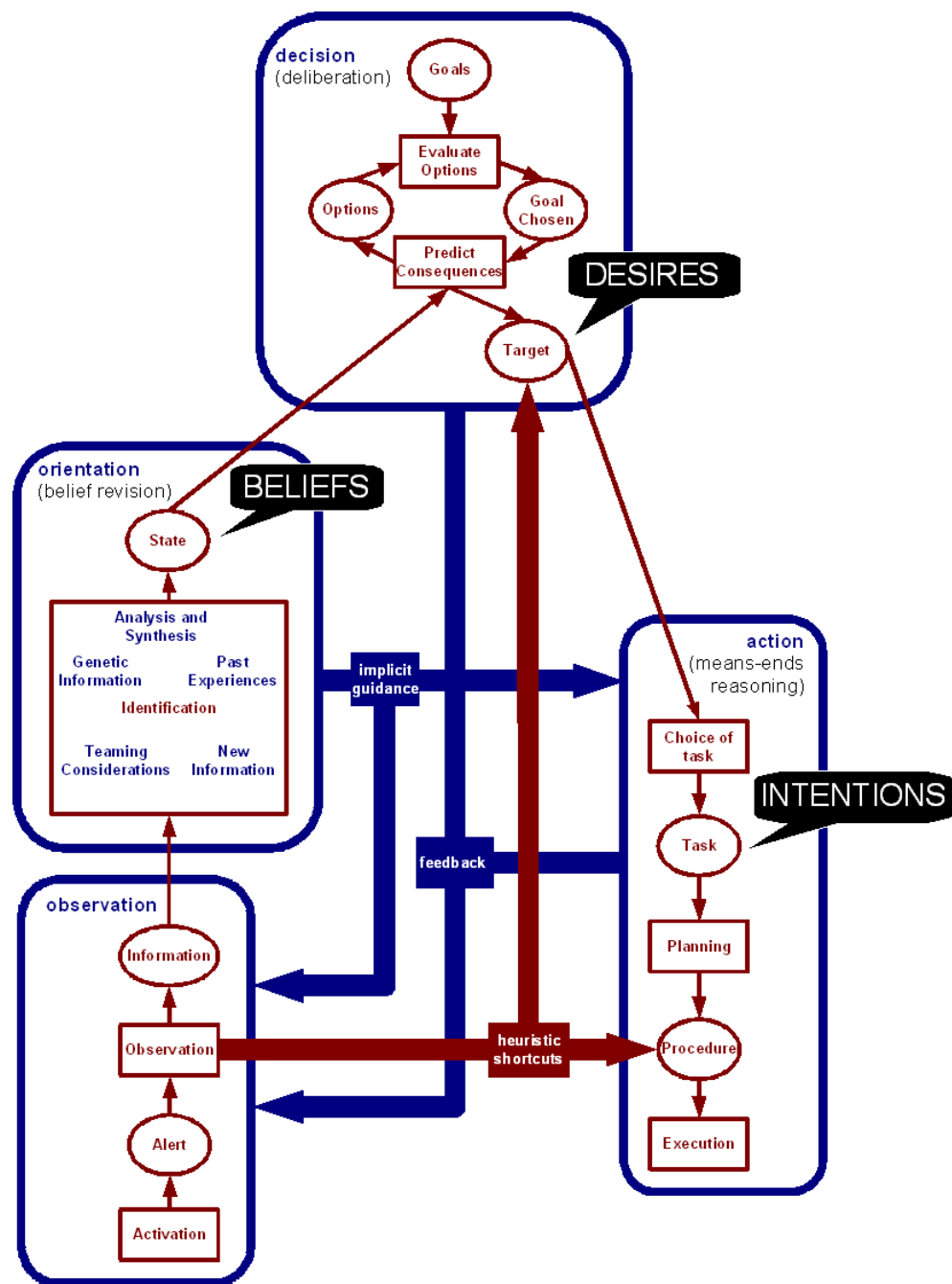


Figure B.1: Cognitive Hybrid Reasoning Model



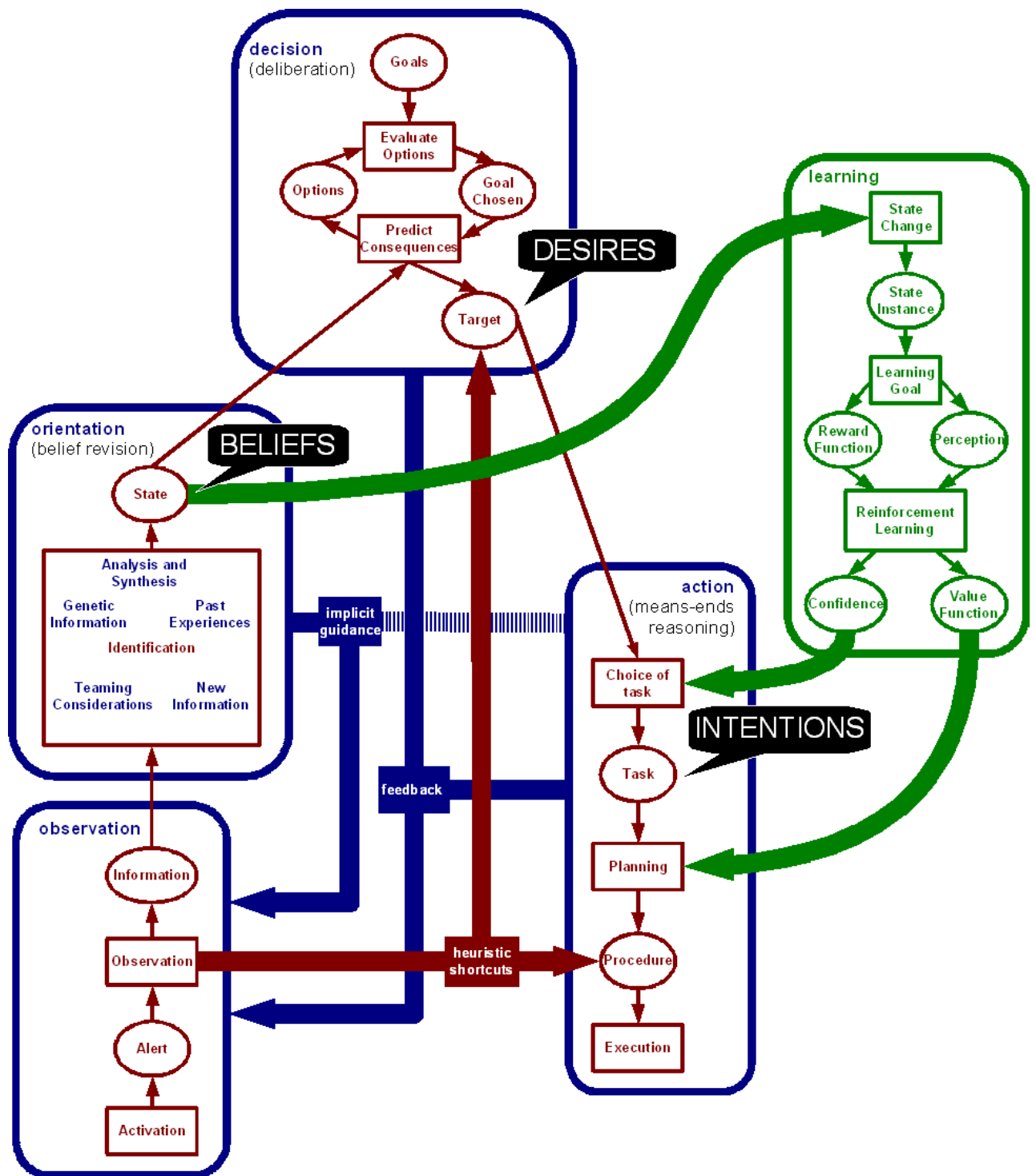


Figure B.2: Cognitive Hybrid Reasoning and Learning Model

# Appendix C

## The 10-Armed Bandit

```
1 //Note: some required package imports and/or code may be omitted for clarity
public agent Bandit extends LearningAgent implements BanditChoice{
3   #has capability CHRIS chris; //the CHRIS capability
   #private data TenChoiceTestbed testBed(); //the environment
5   #private data BanditState state(); //the State
   #posts event StateChange change; //generating StateInstances
7   #posts event Sensation sensation; //starting the reasoning process
   #posts event Target target; //bypassing the decision stage...
9   #handles event Target; //using Target directly because...
   #uses plan RandomSelectionPlan; //it is not needed for bandit
11  #private data banditActions = new LinkedList(); //actions available to Bandit
   #posts event ActionEvent; //the BanditChoice action
13  #handles event ActionEvent; //handling the BanditChoice action
   #uses plan BanditPlan; //performs the action
15  int numTries; //keeps track of the plays

17  public Bandit(int numberOfTries){ //the Bandit constructor
   super("bandit"); //use LearningAgent constructor
19   numTries = numberOfTries; //initialize number of plays
   testBed.init(new Integer(numberOfTries), this); //initialize the environment
21   for (int i=0; i<10; i++){ //initialize the actions
       banditActions.add(new BanditAction(i)); //10 actions are added
23   }
   state.add(0.0); //initialize State to 0.0
25   chris.learningGoals.add("FindBestAction", new BanditLearningGoal()); //add BanditLGoal to LGoal list
}

27  public void activateDecisionProcess(Object rawData){ //starts decision process
29   postEvent(sensation.sensation(rawData)); //posts a sensation event
}

31  public void updateWorldState(Object informationValues) { //updates State
33   state.add(((Double)informationValues).doubleValue());
}

35  public List getActionsAvailable(StateInstance state){ //returns available actions
37   return banditActions; //always the same actions
}

39  public void findBestAction(){ //for every play try to...
41   for (int i=0; i<numTries; i++){ //find the best action
       postEventAndWait(target.target("FindBestAction", "_Agent_", new BanditStateInstance(0)));
43   }
}

45  public void choose(int choice){ //used by BanditPlan to...
47   testBed.step(choice); //access environment
}

49  public void loadLearning(String fileName){ //loads learning data
51   chris.loadLearning(fileName); //not used with bandit
}

53  public void saveLearning(String fileName){ //saves learning data
55   chris.saveLearning(fileName); //not used with bandit
}
}
```

Code Listing C.1: BanditAgent.agent

```

1 //Note: some required package imports and/or code may be omitted for clarity
2 public class BanditLearningGoal extends LearningGoal{

4     public BanditLearningGoal() { //needs name, algorithm and policy
        super("LearnBestAction", CHRISConstants.RL_QLEARNING, CHRISConstants.POLICY_EGREEDY);
6         learning = CHRISConstants.LEARNING_ACTIVE; //or LEARNING_PASSIVE
        passiveTarget = "RandomSelection"; //used for passive learning
8         waitForStateInstanceUpdate = true; //also wait for StateInstance
        skillControl = false; //default skillControl
10        learningTypeControl = false; //default learning types control
        explorationControl = false; //default exploration control
12        explorationMaximum = 1.0; //default maximum egreedy exploration
        explorationChangeExponential = true; //default exploration control change
14    }

16    public Perception getPerception(StateInstance state) { //returns BanditPerception
        BanditStateInstance bs = (BanditStateInstance) state;
18        return new BanditPerception(bs.currentValue);
    }

20    //reward function uses StateInstance directly to avoid state generalization problems
22    //this can only be used with Bandit because StateInstance is same as reward value
    public double rewardFunction(Perception before, Action actionTaken, Perception after) {
24        BanditStateInstance s = (BanditStateInstance) currentStateInstance;
        return s.currentValue;
26    }

28    public int getGoalStatus(StateInstance state) { //bandit episodes have length...
        return CHRISConstants.GOAL_ACHIEVED; //of 1 step only
30    }
}

```

Code Listing C.2: BanditLearningGoal.java

```

1 //Note: some required package imports and/or code may be omitted for clarity
2 public class BanditAction extends Action{
3     int selectedNumber; //The action value selected

5     public BanditAction(int choiceNumber) { //The BanditAction constructor
        super(("Choice"+choiceNumber),"choice"); //All actions must have unique name
6         selectedNumber = choiceNumber;
    }

9     public boolean equivalent(Action other) { //Compares two BanditActions
11        BanditAction otherAction = (BanditAction) other;
        if(selectedNumber == otherAction.selectedNumber)
13            return true;
        return false;
15    }

17    public int hashCode(){ //unique integer for every action
        return selectedNumber;
19    }
}

```

Code Listing C.3: BanditAction.java

```

1 //Note: some required package imports and/or code may be omitted for clarity
2 public plan BanditPlan extends Plan{
3     #handles event ActionEvent ev; //handles action event
4     #uses interface BanditChoice choice; //used to access environment
5     #uses data Actions actions; //uses CHRIS Actions beliefset

6     public boolean relevant(ActionEvent ev){ //only relevant for BanditActions
8         return ev.action instanceof BanditAction;
    }

10    body(){
12        actions.add(ev.action, true); //notify CHRIS action is starting
        BanditAction action = (BanditAction) ev.action; //get required action
14        choice.choose(action.selectedNumber); //perform action
        actions.add(ev.action, false); //notify CHRIS action has finished
16    }
}

```

Code Listing C.4: BanditPlan.plan

```

1 //Note: some required package imports and/or code may be omitted for clarity
2 public beliefset BanditState extends ClosedWorld implements State{
3     #value field double currentValue; //State is value received
4     #indexed query get(logical double currentValue); //Simple JACK query
5
6     #function query double getCurrentValue(){ //returns state's value
7         logical double currentValue;
8         Cursor c = get(currentValue);
9         if(c.next())
10             return currentValue.as_double();
11         return 0.0;
12     }
13
14     public void moddb(){ //automatic callback when updated
15         stateChange(); //call stateChange method
16     }
17
18     #posts event StateChange stateChange; //posts a StateChange event
19     public void stateChange(){
20         postEvent(stateChange.stateChange(getInstance())); //CHRIS updates StateInstance
21     }
22
23     public StateInstance getInstance(){ //returns StateInstance from State
24         return new BanditStateInstance(getCurrentValue());
25     }
26 }

```

Code Listing C.5: BanditState.bel

```

1 //Note: some required package imports and/or code may be omitted for clarity
2 public class BanditStateInstance extends StateInstance{
3     public double currentValue; //StateInstance is currentValue
4
5     public BanditStateInstance(double value) { //simple constructor
6         currentValue = value;
7     }
8
9     public boolean equivalent(StateInstance other) { //compares BanditStateInstances
10         BanditStateInstance state = (BanditStateInstance) other;
11         return currentValue == state.currentValue;
12     }
13
14     public int hashCode(){ //generates unique integer
15         return (int) currentValue*1000;
16     }
17 }

```

Code Listing C.6: BanditStateInstance.java

```

1 //Note: some required package imports and/or code may be omitted for clarity
2 public class BanditPerception extends Perception{
3     public double value; //value returned from environment
4
5     public BanditPerception(double valueRecieved) { //constructor
6         value = valueRecieved;
7     }
8
9     public boolean equivalent(Perception other) { //all BanditPerceptions equal
10         return true;
11     }
12
13     public int hashCode(){ //unique integer for perceptions
14         return (int) value * 1000;
15     }
16 }

```

Code Listing C.7: BanditPerception.java

```

1 //Note: some required package imports and/or code may be omitted for clarity
2 public plan RandomSelectionPlan extends Plan{
3     #handles event Target ev; //handles Target
4     #uses interface BanditChoice bandit; //used to access environment
5     #uses data LearningGoals learningGoals; //used to access LearningGoal
6     #uses data CurrentStateInstance currentStateInstance; //used to access StateInstance
7     #uses data SelectedAction selectedAction; //handshaking with passive RL
8     #uses interface CHRISFunctions chris; //used to access available actions
9     #posts event ActionEvent action; //used to post Action
10
11     public boolean relevant(Target ev){ //relevant for RandomSelection...
12         return ev.name.equals("RandomSelection"); //targets
13     }
14
15     body(){
16         LearningGoal learningGoal = learningGoals.getLearningGoal(ev.parent); //access learningGoal
17         List actions = chris.getActionsAvailable(
18             currentStateInstance.getCurrentStateInstance()); //get available actions
19         int choice= chris.uniformRandom(actions.size()); //choose random action
20         selectedAction.add(learningGoal.name,(Action)actions.get(choice)); //notify passive RL of choice
21         while(selectedAction.get(learningGoal.name) != null) //wait for RL to read choice
22             @wait_for(new Change(selectedAction ,false));
23         @subtask(action.takeAction((BanditAction)actions.get(choice))); //take action
24     }
25 }

```

Code Listing C.8: RandomSelectionPlan.plan

```

1 //Note: some required package imports and/or code may be omitted for clarity
2 public class TenChoiceTestbed implements Environment{
3     LearningAgent learningAgent; //handle for LearningAgent
4     double qStar[] = new double[10]; //real action values
5
6     public TenChoiceTestbed() { //initialize action values...
7         for(int i=0; i< 10; i++) { //to random decimal
8             qStar[i] = uniform.nextDouble();
9         }
10    }
11
12    public void step(int choice){ //reward = value + random error
13        double currentReward = qStar[choice] + Math.abs(gaussian.nextGaussian());
14        learningAgent.activateDecisionProcess(new Double(currentReward)); //notify agent via sensation
15    }
16
17    public void init(Object initData , LearningAgent agent) { //get handle of LearningAgent
18        learningAgent = agent;
19    }
20 }

```

Code Listing C.9: TenChoiceTestBed.java

```

1 //Note: some required package imports and/or code may be omitted for clarity
2 public class BanditSimulation {
3     public static void main(String[] args) {
4         Bandit bandit = new Bandit(10000); //create a new Bandit agent that
5                                             //performs 10000 choices
6         bandit.findBestAction(); //tell agent to find best action
7     }
8 }

```

Code Listing C.10: BanditSimulation.java

# Appendix D

## Extending the Prometheus Methodology

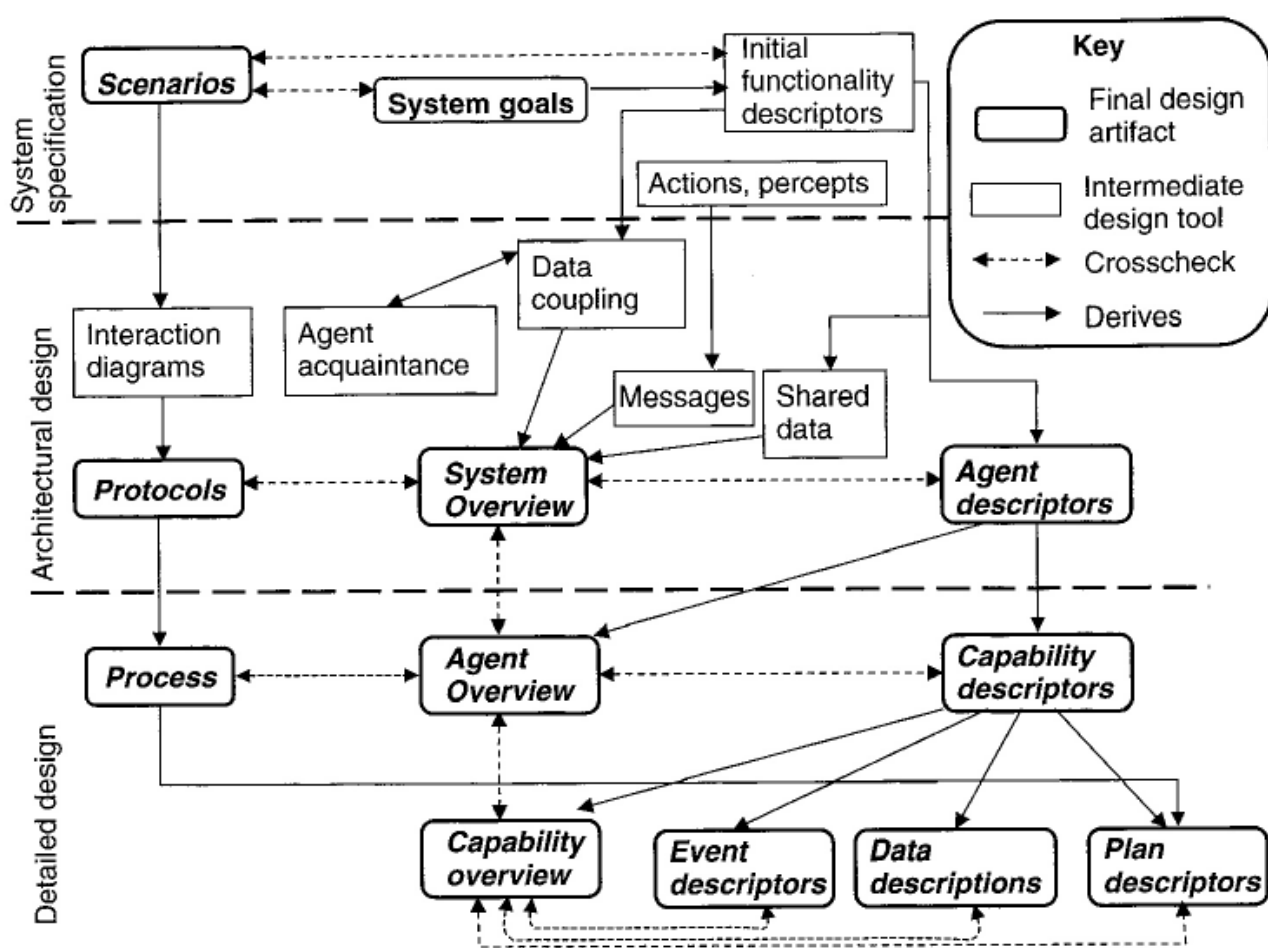


Figure D.1: The Prometheus Methodology [83],p24

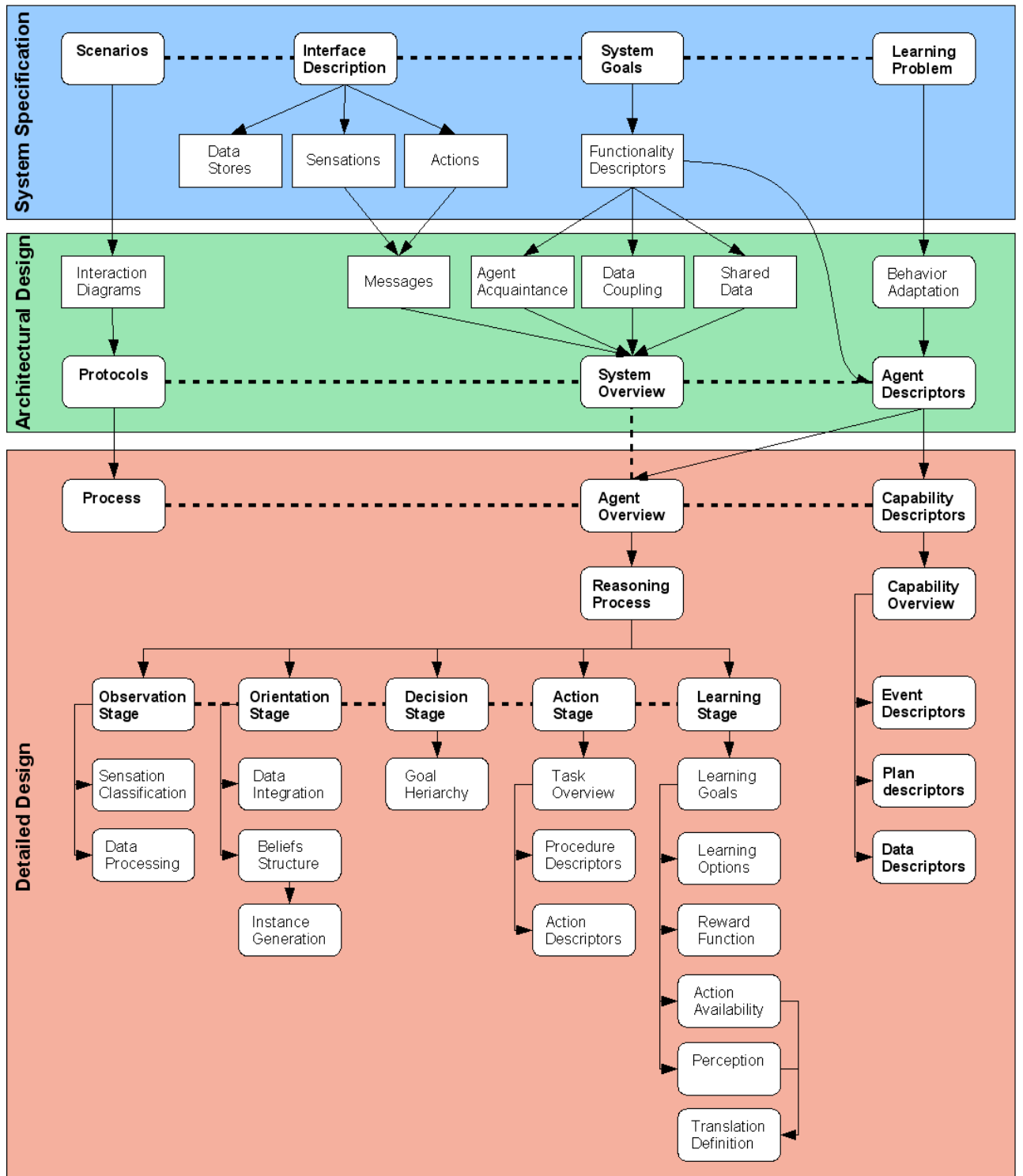


Figure D.2: Extension to Incorporate New Reasoning Model

# Appendix E

## Finding a Path in UT

### E.1 Orientation of UT

```
2 public class UtChrisState extends UtJackInterface implements State{
    CurrentStateInstance currentStateInstance;
    Semaphore syncMutex = new Semaphore(1);
4
    public UtChrisState(CurrentStateInstance currentStateInstance,
6        Self self, Game game, Players players, Navs navs, Movs movs, Doms doms,
        Flags flags, Invs invs, AsyncMsgs async){
8        super(self, game, players, navs, movs, doms, flags, invs, async);
        this.currentStateInstance = currentStateInstance;
10    }

12    public void stateChange(){
        currentStateInstance.add(getInstance());
14    }

16    public void processSyncMessage(UtSyncInfo sync){
        super.processSyncMessage(sync);
18        stateChange();
    }

20    public StateInstance getInstance(){
22        return new UtChrisStateInstance(new UtSyncInfo(self.getInfo(), game.getInfo(), players.getInfo(),
        navs.getInfo(), movs.getInfo(), doms.getInfo(), flags.getInfo(), invs.getInfo()));
24    }
}
```

Code Listing E.1: Collective State for UtJackInterface

```
1 public class UtChrisStateInstance extends StateInstance{
    public UtSelf self;
3    public UtGame game;
    public List players, navs, movs, doms, flags, invs;
5
    public UtChrisStateInstance(UtSyncInfo sync){
7        self = sync.self;
        game = sync.game;
9        players = sync.players;
        navs = sync.navs;
11        movs = sync.movs;
        doms = sync.doms;
13        flags = sync.flags;
        invs = sync.invs;
15    }

17    public boolean equivalent(StateInstance other) {
        UtChrisStateInstance otherInstance = (UtChrisStateInstance) other;
19        if(self.equals(otherInstance.self) && game.equals(otherInstance.game) &&
        layers.equals(otherInstance.players) && navs.equals(otherInstance.navs) &&
21        movs.equals(otherInstance.movs) && doms.equals(otherInstance.doms) &&
        flags.equals(otherInstance.flags) && invs.equals(otherInstance.invs))
23            return true;
        return false;
25    }
}
```

Code Listing E.2: StateInstance for UtJackInterface



## E.2 Actions as Navigation Choices

```
1 public class NavChoice extends Action{
2     UtNav choice;

4     public NavChoice(UtNav nav) {
5         super(nav.id,"NavPath");
6         choice = nav;
7     }

8     public boolean equivalent(Action other){
9         NavChoice c = (NavChoice) other;
10        if(choice.id.equals(c.choice.id))
11            return true;
12        return false;
13    }
14 }
15 }
```

Code Listing E.3: Action for Choosing a UT Navigation Point

```
1 public plan GoToNavPlan extends Plan{
2     #handles event ActionEvent ev;
3     #uses interface UtCommands ut;
4     #uses data Actions actions;
5     #posts event UtMoveEvent move;

6     static boolean relevant(ActionEvent ev){
7         return ev.action instanceof NavChoice;
8     }

9     body(){
10        actions.add(ev.action,true);
11        NavChoice choice = (NavChoice) ev.action;
12        @subtask(move.move((UtCoordinate)choice.choice.location));
13        actions.add(ev.action,false);
14    }

15    #reasoning method fail(){
16        actions.add(ev.action,false);
17    }
18 }
```

Code Listing E.4: Plan for Moving in UT

## E.3 Learning the Quickest Path

```
1 public class Location extends Perception{
2     public UtNav closestNav;

3     public Location(UtCoordinate location, UtNav closestNav) {
4         this.closestNav = closestNav;
5     }

6     public boolean equivalent(Perception other) {
7         Location p = (Location) other;
8         if(closestNav.id.equals(p.closestNav.id))
9             return true;
10        return false;
11    }
12 }
```

Code Listing E.5: Action for Choosing a UT Navigation Point

```

2  public class FindEnemyFlagLearningGoal extends LearningGoal{
    UtChrisStateInstance currentState;

4  public FindEnemyFlagLearningGoal() {
    super("FindEnemyFlagLearningGoal",CHRISConstants.RL_QLEARNING,CHRISConstants.POLICY_EGREEDY);
6  }

8  public Perception getPerception(edu.unisa.chris.orientation.StateInstance state) {
    currentState = (UtChrisStateInstance) state;
10   return new Location(currentState.self.location ,closestNav(currentState.self.location ,currentState.navs));
12 }

14 public double rewardFunction(Perception before, Action actionTaken, Perception after) {
    Location b = (Location) before;
    Location a = (Location) after;
16   UtSelf selfInfo = currentState.self;
    List flagList = currentState.flags;
18   for(int i=0; i<flagList.size(); i++){
        UFlag flagInfo = (UFlag) flagList.get(i);
20     if(flagInfo.team == UtConstants.TEAMRED && selfInfo.location.closeTo(flagInfo.location))
        return 10;
22   }
    if(a.location.y < b.location.y){
24     return -2;
    }
26   return -1;
28 }

30 public int getGoalStatus(edu.unisa.chris.orientation.StateInstance state) {
    return CHRISConstants.GOALACHIEVED;
32 }

```

Code Listing E.6: FindEnemyFlag Learning Goal

```

2  public class FindHomeFlagLearningGoal extends LearningGoal{
    UtChrisStateInstance currentState;

4  public FindHomeFlagLearningGoal() {
    super("FindHomeFlagLearningGoal",CHRISConstants.RL_QLEARNING,CHRISConstants.POLICY_EGREEDY);
6  }

8  public Perception getPerception(edu.unisa.chris.orientation.StateInstance state) {
    currentState = (UtChrisStateInstance) state;
10   return new Location(currentState.self.location ,closestNav(currentState.self.location ,currentState.navs));
12 }

14 public double rewardFunction(Perception before, Action actionTaken, Perception after) {
    Location b = (Location) before;
    Location a = (Location) after;
16   UtSelf selfInfo = currentState.self;
    List flagList = currentState.flags;
18   for(int i=0; i<flagList.size(); i++){
        UFlag flagInfo = (UFlag) flagList.get(i);
20     if(flagInfo.team == UtConstants.TEAMBLUE && selfInfo.location.closeTo(flagInfo.location))
        return 10;
22   }
    if(a.location.y > b.location.y){
24     return -2;
    }
26   return -1;
28 }

30 public int getGoalStatus(edu.unisa.chris.orientation.StateInstance state) {
    return CHRISConstants.GOALACHIEVED;
32 }

```

Code Listing E.7: FindHomeFlag Learning Goal

# Appendix F

## Which Weapon to Use?

This appendix briefly describes the different weapons available within the UT world. It is included to illustrate the many differences between weapons and therefore reinforce the importance of an agent knowing (or learning) which weapon to use depending on the situation it is faced with. Each weapon has a primary firing mode and also a secondary firing mode. They are listed here in order of their power.



---

**Name:** Impact Hammer

**Primary:** Compressed Air, must touch enemy with it to inflict damage. Hold to charge in order to inflict more damage. This weapon is carried by all players initially and requires no ammo so it has unlimited usage.

**Secondary:** None.

---



---

**Name:** Enforcer

**Primary:** Low power, fast projectile bullets with long range and high accuracy, but slow firing repetitions. This weapon is carried by all players initially with a full load of ammo.

**Secondary:** Improves repetition speed but reduces accuracy.

---



---

**Name:** Shock Rifle

**Primary:** Medium powered laser beam, instant projectile speed, long range, very accurate but slow firing repetitions.

**Secondary:** Plasma ball with more power but slow projectile speed. Also long range, very accurate but slow firing repetitions.

---



---

**Name:** Bio Rifle

**Primary:** High powered, slow projectile speed toxic sludge that can stick on walls. Short range, slow firing repetitions and inaccurate.

**Secondary:** Hold to charge a large blob of sludge that can kill in a single shot.

---



**Name:** Pulse Gun

**Primary:** Medium powered plasma balls, medium projectile speed, long range, fast firing repetitions and very accurate.

**Secondary:** Medium powered plasma bolt (like a lightning strike). Long range, constant firing possible until ammo runs out, very accurate.

---



**Name:** Sniper Rifle

**Primary:** High powered bullets, instant projectile speed and very accurate. Very long range but has very slow firing repetition.

**Secondary:** Activates the telescope in order to zoom-in to targets from far away.

---



**Name:** Ripper

**Primary:** Medium powered sharp metallic blades and medium projectile speed. Long range, fast firing repetitions and very accurate. Blades can also bounce off walls.

**Secondary:** High powered explosive blades with long range, slow firing repetition but good accuracy.

---



**Name:** Mini Gun

**Primary:** Low powered bullets with an instant projectile speed and medium accuracy. Long range with a very fast firing repetition speed.

**Secondary:** Doubles the firing repetition speed but halves the accuracy of the bullets.

---



**Name:** Flak Cannon

**Primary:** Medium powered chunks of metal, medium projectile speed with low accuracy and slow firing repetitions. A single shot spreads many pieces over a large area. Metal pieces have a long range but become too scattered over long distances and inflict little damage.

**Secondary:** High powered grenade that disperses chunks of metal. It has a short range and very slow firing repetition.

---



**Name:** Rocket Launcher

**Primary:** Very High powered explosive rockets, slow projectile speed with high accuracy and slow firing repetitions.

**Secondary:** Hold to start collecting rockets such that up to five rockets are fired spread over a larger area. Automatically fires when five rockets are collected.

---



**Name:** Redeemer

**Primary:** Extremely High powered atomic rockets, very slow projectile speed with high accuracy and very slow firing repetitions. When exploded the resulting shockwave kills everyone within a medium range open area.

**Secondary:** Allows for controlling the rocket's flight using a remote camera such that it can be guided to a particular destination

---