

University of Alberta

**USING BEHAVIOUR PATTERNS TO GENERATE SCRIPTS FOR
COMPUTER ROLE-PLAYING GAMES**

by

Maria Cutumisu

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

©Maria Cutumisu
Fall 2009
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.



Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

ISBN: 978-0-494-52438-1

Our file *Notre référence*

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.


Canada

Examining Committee

Duane Szafron, Computing Science

Michael Mateas, Computer Science, University of California, Santa Cruz

Michael Carbonaro, Educational Psychology

Jonathan Schaeffer, Computing Science

Paul Lu, Computing Science

Abstract

Character behaviours in computer role-playing games have a significant impact on game-play, but are often difficult for story authors to implement and modify. Many computer games use custom scripts to control the behaviours of non-player characters (NPCs). Therefore, a story author must write fragments of computer code for the hundreds or thousands of NPCs in the game world. The challenge is to create non-repetitive (more entertaining) behaviours for the NPCs without investing substantial programming effort to write custom non-trivial scripts for each NPC. Consequently, current computer games mostly rely on simplistic non-interactive behaviours for NPCs. This research describes the design and implementation of a novel behaviour model for interacting NPCs, based on generative design patterns, that requires no manual script writing. In this model, NPCs assume different roles during the story and select behaviours based on static probabilities or dynamic motivations. We also devised a reinforcement learning algorithm, ALeRT, based on Sarsa(λ) and we extended our behaviour model to support behaviour selection based on learning. In our model, an NPC can exhibit proactive, reactive, or latent behaviours that may be independent or collaborative. This behaviour architecture supports behaviours that can be interrupted and resumed based on priorities. The implementation of this model produces scripting code for BioWare Corp.'s *Neverwinter Nights* computer role-playing game.

Acknowledgements

I would like to thank my supervisor, Duane Szafron, for his continuous support, guidance, and enthusiasm, as well as to the members of my dissertation committee: Michael Mateas, Michael Carbonaro, Jonathan Schaeffer, and Paul Lu for their valuable comments and suggestions that helped me write a better manuscript.

I would also like to thank the members of the ScriptEase group, past and present, for their help and always entertaining company in the Software Systems lab. I would especially like to thank Kevin Waugh and Matt McNaughton for their support and advice regarding behaviour patterns and encounter patterns, respectively.

Table of Contents

1	Introduction	1
1.1	Interactive Character Behaviours	5
1.2	Behaviours in ScriptEase	6
1.3	Research Contributions	7
1.4	Organization	9
2	Related Work	10
2.1	The State of NPC Behaviours	10
2.2	Requirements for NPC Behaviours	18
2.3	Chapter Summary	20
3	NPC Behaviours	21
3.1	Behaviour Model	22
3.2	Behaviour Patterns	28
3.3	Use of Behaviour Patterns	30
3.4	The Structure of Behaviour Patterns	33
3.4.1	Cues	35
3.4.2	Performances and Roles	39
3.4.3	Motivations	42
3.4.4	Basic Behaviours	47
3.4.5	Proactive Behaviours	48
3.4.6	Reactive Behaviours	51
3.4.7	Latent Behaviours	52
3.4.8	Tasks	54
3.5	Collaborative Behaviours	56
3.6	Behaviour Dispatch and Implementation	63
3.6.1	Latent Queue	74
3.6.2	Collaborative Queue	75
3.6.3	Independent Queue	77
3.6.4	Behaviour Dispatch Summary	77
3.7	Concurrency Control Model	86
3.7.1	Synchronization	87
3.7.2	Deadlock	89
3.7.3	Indefinite Postponement	92
3.8	Chapter Summary	93
4	Reinforcement Learning in ScriptEase	94
4.1	Introduction	94
4.2	Related Work	98
4.3	Algorithm	100
4.3.1	Sarsa(λ)	100
4.3.2	ALeRT	103

4.4	Implementing ALeRT in <i>NWN</i>	115
4.5	Using RL in ScriptEase	120
4.5.1	Using an RL Performance	121
4.5.2	RL Cue Components	122
4.5.3	Creating and Adding RL Behaviours	127
4.5.4	Creating and Adding RL Roles	128
4.6	Integrating RL in ScriptEase	129
4.6.1	The RL Auxiliary File	133
4.6.2	Changes to Spronck's Arena Module	134
4.7	Experiments and Evaluation	135
4.7.1	Motivation - ALeRT, M1, and <i>RL</i> vs. Static Opponents	138
4.7.2	Dynamic Opponents - ALeRT and <i>RL</i> vs. M1	142
4.7.3	Adaptability in a Dynamic Environment	143
4.8	Observations	147
4.9	Chapter Summary	148
5	Evaluation of the Behaviour Pattern Model	151
5.1	Evaluation from a Usability Perspective	151
5.2	Behaviour Pattern Efficacy	156
5.2.1	Correctness - Case Study	156
5.2.2	Expressiveness - Case Study	159
5.2.3	Inheritance of Behaviour Patterns - Case Study	171
5.3	Evaluation Measures	174
5.4	Chapter Summary	179
6	Conclusions and Future Work	180
	Bibliography	185
A	An Introduction to ScriptEase	191
B	Behaviour Pattern Catalogue Description	197
B.1	Approacher	197
B.2	Attacker	198
B.3	Beckoner	198
B.4	Beseecher	198
B.5	Challenger	199
B.6	Checker	199
B.7	Destroyer	199
B.8	Dispossessor	200
B.9	Exclaimer	200
B.10	Exclaimer with Animation	200
B.11	Expert	201
B.12	Follower	201
B.13	Guard	201
B.14	Interactor	202
B.15	Loiterer	202
B.16	Manipulator	203
B.17	Patroller	203
B.18	Poser	203
B.19	Rester	204
B.20	Returner	204
B.21	Spawner	204
B.22	Spellcaster	205

B.23 Striker	205
B.24 User	205
B.25 Vanisher	206
B.26 Wanderer	206
B.27 Withdrawer	206
C Changes to the Arena Module	208

List of Figures

1.1	ScriptEase generates NWScript code automatically from patterns. The generated NWScript code is compiled into bytecodes that are interpreted by BioWare Corp.'s Aurora Engine.	4
3.1	The types of NPC behaviours: independent or collaborative, as well as proactive, reactive, or latent.	23
3.2	The multiple roles of a guard NPC.	26
3.3	NWScript code for an <i>NWN</i> guard.	29
3.4	Using a performance in ScriptEase: setting the <code>Actor</code> option of a ScriptEase Guard performance.	31
3.5	Using behaviour patterns in ScriptEase: setting the <code>Guarded</code> option of a ScriptEase role in the Guard performance.	32
3.6	The structure of the Guard performance in ScriptEase.	34
3.7	The cues that activate the components of a performance.	40
3.8	The components of a simple Rumour performance in ScriptEase.	41
3.9	The proactive vector used to determine the probability of selecting a proactive behaviour.	44
3.10	The update of the Rest proactive vector: the values of the Duty and Threat motivational attributes increase, while the value of the Tiredness attribute decreases.	46
3.11	The guard NPC is motivated by Duty (D) , Tiredness (Ti) , and Threat (Th) as it patrols (left), rests (center), and checks (right). The bars show the changes of the motivational attributes.	47
3.12	The Rest proactive independent behaviour of the Guard role.	49
3.13	The Guard role reveals three proactive independent behaviours.	50
3.14	Setting the selection probability for the Patrol proactive independent behaviour of the Guard role.	51
3.15	The Patrol task of the Patrol proactive independent behaviour.	52
3.16	Each pair of tasks in a protocol completes successfully before a new pair of tasks can be executed.	57
3.17	The proactive-reactive pairs Converse-Talk and Pose , as well as Converse-Talk and Converse-Listen collaborative behaviours may have different lengths. At run-time, Listen tasks are added to the shorter chain (Pose) until the chains are identical in length.	59
3.18	The lengths of reactive behaviours cannot be computed at compile-time due to multiple roles on each side of a collaboration and due to multiple collaborators.	62
3.19	Behaviour dispatch of the proactive, reactive, and latent behaviours. If no queued task is available, the dispatcher enqueues a new behaviour.	64
3.20	The NWScript events attached by ScriptEase to an NPC with behaviour patterns, shown as they appear in the Aurora Toolset.	65
3.21	<i>NWN</i> event loop for NPCs.	66

3.22	The behaviour selector for a guard NPC in our behaviour model.	68
3.23	A tavern server and a patron performing a collaborative behaviour: the server initiates a proactive Offer-fetch behaviour and the patron responds with a reactive Receive behaviour.	71
3.24	Behaviour dispatch detail for an NPC with two roles, Guard and Patron	71
3.25	Behaviour dispatch algorithm.	73
3.26	The Warn latent behaviour interrupts the Rest proactive independent behaviour of a guard NPC when an intruder is near the guarded chest.	81
3.27	The Converse-Listen reactive behaviour interrupts the Patrol proactive independent behaviour, since a proactive independent behaviour has a lower priority than a reactive behaviour.	82
3.28	After Bob and Mary finish their latent collaborative behaviour, Bob and the server resume their proactive collaborative behaviour. Then, Bob and Sally resume their interrupted proactive collaborative behaviour.	83
3.29	The Overhear-Talk latent collaborative behaviour interrupts the Converse-Talk proactive collaborative behaviour when the PC walks near the NPCs, since a latent behaviour has a higher priority than a proactive behaviour.	85
3.30	The barrier and eye-contact mechanisms that ensure behaviour synchronization.	89
4.1	The Sarsa(λ) linear gradient-descent algorithm.	102
4.2	The ALeRT algorithm.	104
4.3	Exploration/exploitation (epsilon) values when the phase changes from a Melee to a Ranged equipment configuration. The x-axis shows episodes starting with episode 500 and the y-axis shows epsilon values between $\epsilon_{min} = 0.005$ and $\epsilon_{max} = 0.015$	106
4.4	A trend is detected for the speed action starting with episode 408 of a Melee configuration. The upper trace represents the δ values for the speed action, while the lower trace represents the delta-bar-delta (dbd) values for speed immediately after the NPC drank a speed potion.	109
4.5	We compute the delta-bar and the average delta-bar, $\mu_{\bar{\delta}}$, for the speed action to determine whether the trend for the speed action is significant.	110
4.6	We identify a significant trend for the speed action: the current variation of delta-bar from the average delta-bar, $\mu_{\bar{\delta}}$, exceeds the standard deviation ($f = 1$) of delta-bar, $\sigma_{\bar{\delta}}$, for the speed action, while delta-bar-delta is positive.	110
4.7	When a significant trend for the speed action is detected, the value of the alpha parameter for the speed action is increased, so that the NPC learns faster.	111
4.8	No trend is detected for the melee action at episode 38 of a Melee configuration, therefore the value of the alpha parameter for the melee action is decreased.	112
4.9	Phase change from a Melee to a Ranged configuration: after the NPC learns to favour the ranged action, the values of the alpha parameters for both the melee and the ranged actions decrease.	114

4.10	Four of the states in the state space for the Fighter role: <i>HSL</i> (the NPC is injured), <i>EA</i> (the NPC has the speed enhancement potion in hand), <i>EO</i> (the NPC has just drunk the speed enhancement potion), and <i>DM</i> (the NPCs are within melee distance). The constant state is not represented in this figure.	117
4.11	The action space for the Fighter role.	117
4.12	An RL performance in ScriptEase that contains the RL-Combat and RL-Listener-Combat roles.	121
4.13	The Activate RL cue action starts a generic RL step, the Activate RL early cue action starts an early RL combat step, and the End episode action ends a generic RL episode.	123
4.14	The components of the <i>RL</i> role cue. The default parameter values were determined empirically by the pattern designer.	125
4.15	NWScript code for the update of the reward function.	128
4.16	The events that activate the <i>RL</i> role cue for combat.	130
4.17	The cues used in combat RL for a fighter NPC.	131
4.18	Two fighters in Spronck's arena performing a ranged (left) and a melee (right) action, respectively.	135
4.19	<i>RL</i> ₀ and <i>RL</i> ₃ vs. NWN and OPT.	137
4.20	ALeRT and M1 vs. NWN and OPT.	139
4.21	ALeRT vs. NWN and OPT.	140
4.22	M1 vs. NWN and OPT.	141
4.23	<i>RL</i> ₀ and ALeRT vs. M1 - Phase 1 (500 episodes).	142
4.24	<i>RL</i> ₀ and ALeRT vs. M1.	144
4.25	ALeRT vs. M1 - Melee-Ranged&Heal.	145
4.26	ALeRT vs. M1 - Ranged-Melee&Heal.	146
4.27	ALeRT vs. M1 - Heal-Melee&Ranged.	146
5.1	Behaviour and encounter patterns used by all 25 high-school students participating in the study. Individual behaviour patterns are grouped into one category.	152
5.2	Behaviour and encounter patterns used by nine high-school students in their interactive stories.	153
5.3	Pattern instances (encounter and behaviour) used by all students. Behaviour patterns are grouped into one category.	154
5.4	An owner, two tavern servers, and a patron exhibiting ScriptEase-generated behaviours in a tavern scene.	157
5.5	Encounter pattern reuse by module in the <i>NWN</i> campaign.	161
5.6	Generated behaviours in the <i>Prelude</i> : Duet-Converser-Converser	163
5.7	Generated behaviours in the <i>Prelude</i> : Duet-Spawner-Destroyer	165
5.8	Manually written NWScript code for the spawner and destroyer NPCs.	166
5.9	An instance of the Duet-Spawner-Destroyer pattern in the <i>NWN Prelude</i>	167
5.10	The Spawner and Destroyer pattern instances in the <i>NWN Prelude</i>	168
5.11	Using ScriptEase encounter patterns to generate behaviour scripts in the <i>NWN Prelude</i>	169
5.12	The inheritance hierarchy of ScriptEase behaviours using encounter patterns.	172
A.1	Creating and placing a container using the Aurora Toolset.	192
A.2	A generative pattern, its description and a dialog being used to set an option.	193
A.3	A portion of the code generated for the pattern in Fig. A.2.	194
A.4	Adapting a pattern by adding an action.	195

C.1	ud_igor_01 script	208
C.2	os_learn_01 script	209
C.3	in_learn_generic script	209
C.4	ou_learnlever_02 script	209
C.5	The constants defined in the i_se_rl_modific file.	210
C.6	Nera's Scripts: ALeRT learning algorithm.	210
C.7	Blanche's Scripts: Spronck's rule-based learning algorithm.	211

List of Tables

4.1	Agent equipment configurations and optimal strategies.	137
5.1	Tavern behaviours for three types of NPCs: a tavern server (S), a tavern patron (P), and a tavern owner (O).	158
5.2	ScriptEase encounter pattern statistics. <i>Chapter One*</i> consists of the <i>Prelude</i> , <i>Chapter One Finale</i> , and <i>Chapter One</i> modules.	160

Chapter 1

Introduction

Computer role-playing game (CRPG) designers have recently shifted their focus to enhancing the sophistication, excitement, and depth of their stories rather than just enhancing the realism of their graphics. Content creation has become the bottleneck in the production of current games due to consumer demand for richer content, as well as continuing advances in hardware that have helped game developers place more emphasis on content rather than form. Since most authors of interactive stories are writers and artists rather than programmers, new tools are needed to speed up this process and to allow authors to express their ideas easily and reliably, without manually scripting their stories. This dissertation addresses the content bottleneck for behaviour scripting.

A CRPG contains an engine designed to dispatch game events to scripts and play stories composed of individual modules that are constructed by story authors (game designers). A module is a self-contained file that includes areas, non-player characters (NPCs), and other game objects (props) that can be scripted to respond to game events. The state-of-the-art in game scripting is to manually script individual game objects that interact in the game. For example, if a game object must interact with the player character (PC) or another game object, a script must be written. For each scripted object in the game, a text-based scripting language is used to specify actions that run in response to game events. The game engine renders the story world's objects, generates events on the objects, dispatches events to scripts and executes the scripts. The construction of a complex story requires the author to script a considerable number of interacting game objects, such as props and NPCs. Thou-

sands of such objects must be tracked using criteria that include their physical areas, their associated sub-plots, and their static/adaptive status. Tracking the objects in this manner is hard, but tracking the scripts is even more difficult since most scripts involve the interaction of several objects. Scripts communicate with each other through global variables, object state, or events. The large number of objects in a CRPG virtual world requires story authors to focus their scripting efforts on a privileged set of objects vital to the story line. This situation has a negative effect on the breadth and immersiveness of the game experience. Since programmers are currently writing scripts manually, serious concerns about programming effort, reliability, and testability are raised. The scripting process can be particularly difficult for story authors who lack a programming background.

Different stories can be “played” with the same game engine using story-specific objects and scripts. Programmers create game engines using programming languages such as C or C++. The goal of this research is to improve the way game stories, not game engines, are created. A story author who is not usually a programmer [64] writes game stories by creating thousands of game objects and scripts for each story. For example, *Neverwinter Nights (NWN)* [58] is an award-winning CRPG from BioWare Corp. that uses the NWScript language to expose powerful scripting facilities to professional and amateur story authors. The *NWN* campaign story contains 54,300 game objects of which 29,510 are scripted, including 8,992 objects with custom scripts, while the others share a set of predefined scripts. The scripts consist of 141,267 lines of NWScript code in 7,857 script files [19]. Many games have a toolset that allows an author to create game objects and attach scripts to them. Examples are BioWare Corp.’s *Aurora Toolset* [1] that uses NWScript and Epic Game’s *UnrealEd* [28] that uses UnrealScript. *NWN* is a popular game that has a vibrant community of story writers, in addition to professional story authors from BioWare Corp. Thousands of individuals write and share their own game adventures on the web. For example, there are over 5,000 adventure stories posted in a common repository and the most popular adventure has been downloaded over 274,110 times [59]. Authors demand the ability to create custom scripts without relying on a set of predefined scripts or on a programmer to write custom scripts.

However, story creation should be more like story writing than programming, so an author should not have to write scripts either.

The interactive nature of games offers the player a plethora of possibilities, but raises as many difficulties for the story authors. An interactive story adds a new dimension over a traditional pen-and-paper linear story and the story authors have to be aware of this situation when creating their story concepts. Testing non-linear stories with a large number of scripts introduces additional challenges for the story authoring process. Many common errors are difficult to detect without playing through all of the game scenarios and trying all of the different combinations of player choices. For example, scripts are often created using cut-and-paste techniques and it is not uncommon for the programmer to cut-and-paste scripts without performing any changes needed for the new context. In addition to being time consuming, manual scripting causes errors due to mislabeled objects with obscure names such as “M1S04CPATRON”. There are so many game objects and scripts that it has become standard practice to use object numbers or script numbers as part of their names. An off-by-one error in a name often results in a syntactically legal script that performs incorrectly. In addition, the scripting code is repeated among scripts that have obscure names as well. The lack of comments or the existence of misleading comments left in the script file, even after the code has been changed or commented out, participates in creating scripts that are hard to understand. This problem is exasperated by the growing number of story authors who do not have programming skills and who must rely on programmers to write their scripts.

Researchers [50] proposed a fast and easy way of solving these problems through the use of generative design patterns [31][21]. Frequently occurring themes in the game, such as *pull a lever - open a door*, are captured using generative design patterns that can be adapted for various game scenarios, with no programming knowledge required. They have proposed [51] four types of patterns: *encounter patterns* - for generating scripts attached to inanimate objects, *behaviour patterns* - for generating scripts attached to creatures, *dialogue patterns* - for generating conversation scripts, and *plot patterns* - for generating scripts that control story plots. They developed a visual programming tool, ScriptEase [70], that automatically translates the

generative design patterns into code that a game engine understands, as illustrated in Figure 1.1. This is the only effort aimed at applying design pattern technology to the most difficult content creation problem in the computer games industry - script creation. Design patterns have previously been used in describing the rules and structure of game-play [3] or game engines [43], but not to generate content.

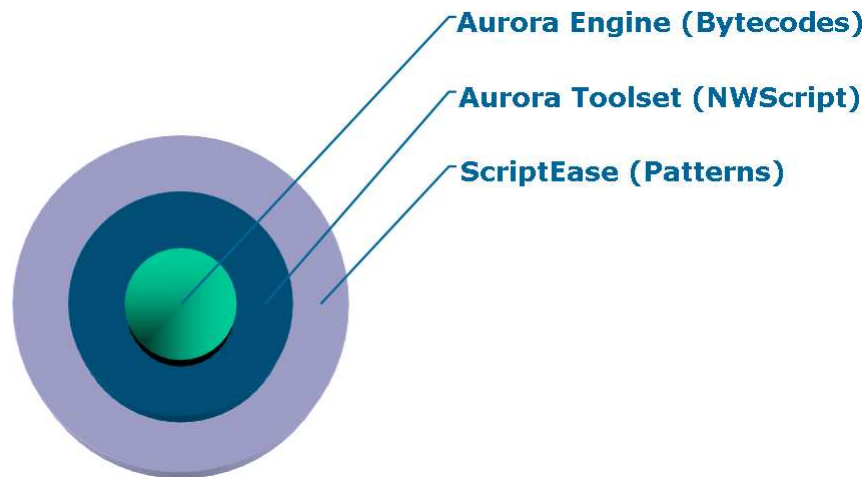


Figure 1.1: ScriptEase generates NWScript code automatically from patterns. The generated NWScript code is compiled into bytecodes that are interpreted by BioWare Corp.'s Aurora Engine.

ScriptEase [70] is a scripting tool that facilitates the game story authoring process using a high-level menu-driven programming model. ScriptEase solves the non-programmer problem by enabling the author to easily create scenes at the level of patterns [31][50] that generate complex interactive stories for computer role-playing games automatically. ScriptEase enables non-programmers to create complex scripts by selecting patterns and adapting them for specific game situations, without writing code. Then, ScriptEase generates reliable scripting code from patterns automatically. Generative design patterns are also used to organize the large number of generated scripts. An important software engineering advantage of ScriptEase is that the automatically generated code is self-documenting. The patterns themselves are used as documentation. Story authors that need to examine behaviour patterns in more detail use this documentation to adapt and test their patterns. ScriptEase also generates comments as part of the code, always reflecting

the existing state of the code. Thus, testing and debugging are considerably easier for programmers who need to tweak the code. Moreover, using patterns decreases programming errors and increases code generation productivity and reliability, also enabling rapid prototyping and code reuse. ScriptEase enables authors to manage their work by creating folders to organize their patterns during game development. The pattern import/export mechanism available in ScriptEase is a high-level mechanism that supports code reuse.

This novel way of programming at a higher level makes scripting more productive, eliminating many common types of errors. Using design patterns instead of writing code manually helps non-programmer story authors create more complex and entertaining stories quickly. ScriptEase supports a rich set of encounter patterns (described in Appendix A) as first-class objects. These encounter patterns re-occur in most CRPG stories and facilitate complex interactions between the PC and inanimate objects in the game, such as doors, props, and triggers. ScriptEase can also be used to create more specialized encounter patterns that occur frequently in the development of a particular story. The main contribution of the research described in this dissertation is the extension of the ScriptEase system to include *behaviour patterns* for NPCs as first-class objects.

1.1 Interactive Character Behaviours

Since many computer games use custom scripts to control the behaviours of NPCs, a story author must write fragments of computer code for the hundreds or thousands of NPCs in the game world. Using AI to create NPCs that exhibit near-realistic behaviours is essential, since richer background NPCs produce a more entertaining game. However, this requirement must be put in context: the storyline takes precedence. NPCs that are not critical to the plot are often added at the end of the game development cycle, only if development resources are available. Therefore, many NPCs that could potentially enrich a game adventure display repetitive behaviours due to the authors' concentrated efforts on developing NPCs directly involved in the storyline.

The challenge is to create entertaining and non-repetitive behaviours for the NPCs without investing substantial programming effort to write custom non-trivial scripts for each NPC. Most of the current computer games have simplistic behaviours for NPCs in which it is rare for NPCs to interact with each other. The most novel and challenging behaviours are collaborative (interacting NPCs) and they are seldom seen in computer role-playing games.

A tool that facilitates game story writing, one of the most critical components of game creation, should meet the following requirements that constitute one of the goals of this research, in the context of supporting behaviour authoring for NPCs:

1. it should be usable by non-programmers,
2. it should support a rich set of non-repetitive interactions,
3. it should support rapid prototyping, and
4. it should eliminate most common types of errors.

1.2 Behaviours in ScriptEase

This research adds a new type of generative design pattern to ScriptEase, called a *behaviour pattern*. An author begins by using BioWare Corp.'s Aurora Toolset to create the physical layout of a story, without attaching any scripts to objects. The author then selects appropriate behaviour patterns that generate scripting code for the NPCs in the story. For example, in a tavern scene, behaviour patterns for patrons, servers, and the owner would be used to generate all the scripting code to animate the tavern. ScriptEase is intended for a broad audience, from programmers to story authors without programming experience. ScriptEase was shown to be usable in practice by non-programmers, being integrated into a Grade 10 English curriculum [11][12]. Story authors can use commonly occurring patterns to generate scripting code without any programming knowledge.

In particular, this research has added facilities to ScriptEase that support the automatic generation of scripting code for background (proactive or reactive) behaviours, as well as latent (stimulated/triggered) behaviours of the numerous NPCs

that populate the CRPG world. ScriptEase generates scripting code automatically from behaviour patterns, responding to the challenge of creating entertaining and non-predictable behaviours for NPCs without the effort of writing custom complex scripts for each NPC. This research addresses only high-level behaviours, since the *NWN* game engine solves low-level problems. For example, if the original location of an NPC in the game is occupied by another creature when an NPC tries to return to it, the game engine moves the NPC as close as possible to that location. Subsequent return behaviours may allow the NPC to return to its original location.

The extra level of indirection between author and programmer, which increases the chances of creating a product that does not match the author's intentions, has been eliminated for NPC scripts. Such miscues are analogous to situations that occur between customers/requirements analysts and authors/programmers during the development of more general software systems. ScriptEase behaviour patterns have solved this critical problem for NPCs. The use of patterns to generate scripts will have a large impact on the methodology of game development. This technique will provide the story authors with the power to fulfill their vision without sacrificing valuable programming time writing detailed scripts. The saved time can be used by authors to create more compelling stories and richer NPC behaviours that produce a more realistic game experience. Time saved by programmers can be used for pattern writing and pattern testing. This research has doubled the impact of ScriptEase patterns by extending all the benefits accrued for encounter patterns (PC-object interactions) to behaviour patterns (NPC actions, PC-NPC interactions, and NPC-NPC interactions).

1.3 Research Contributions

This research has extended the generative pattern approach of ScriptEase to support the behaviours of NPCs. Interactions among NPCs require concurrency control to ensure that neither deadlock nor indefinite postponement can occur and to ensure that interactions are realistic. We constructed an NPC interaction concurrency model and built generative patterns for it.

An NPC who is guarding an item stored in a chest is used to illustrate our behaviour model. The guard NPC performs realistic actions such as patrolling around the guarded chest, resting on a bench near the guarded chest, checking on the charge, and conversing with other guards. If the guarded item is stolen, the guard no longer guards an empty chest. Instead, it flees the area to avoid facing the guard's captain. At the end of the shift, the guard travels to a tavern, talks to other patrons, and orders drinks. Richer NPC behaviours, such as the guard, are now being demanded in modern CRPGs, even for NPCs with bit parts. However, to create this kind of behaviour in thousands of background NPCs, manual scripting must be eliminated. Reusable behaviours that can be adapted to a wide range of NPCs must be created quickly and reliably. Authors with no programming background should be able to create and modify sophisticated NPC behaviours.

In this dissertation, we describe our novel approach to NPC behaviours. The research contributions of this dissertation are the following:

1. It is the first time patterns have been used to generate NPC behaviour scripts for computer games. Our behaviour model and patterns can be used to generate code for the *Neverwinter Nights* computer role-playing game.
2. Independent behaviours for individual NPCs and collaborative behaviours for interacting NPCs are supported. Behaviours can be interrupted and resumed based on priorities.
3. A fully implemented motivation model has been developed to select proactive behaviours.
4. A collaborative protocol mechanism allows story authors to easily create and reuse collaborative behaviours. This novel protocol simplifies the way in which NPCs interact and allows them to more easily collaborate with a broader range of NPC types.
5. Latent behaviours triggered by game events, other NPCs, or PCs are supported, including the ability to return to partially-completed behaviours after the latent behaviour is completed.

6. A *performance* mechanism allows an NPC to assume different *roles* at different story phases activated by *cues* (e.g., game events or timers).
7. A novel reinforcement learning algorithm, ALeRT, that drives the selection of NPC behaviours expands the capability of our flexible behaviour model.

We also address the following research question: can story authors generate complex and adaptive behaviours quickly and reliably without programming?

1.4 Organization

This dissertation is organized as follows. Chapter 2 presents the requirements of NPC behaviours in games and reviews the current state of NPC behaviours in the literature. Chapter 3 describes our behaviour model and implementation. Chapter 4 describes our learning model for NPC behaviours and the implementation of the ALeRT algorithm. Chapter 5 describes the experiments and the evaluation techniques conducted to validate our approach. Finally, Chapter 6 summarizes our contributions and presents ideas for future work. Appendix A provides an overview of the ScriptEase tool, Appendix B presents our behaviour pattern catalogue, and Appendix C describes our changes to an existing game module in which we integrated our RL algorithm.

Chapter 2

Related Work

NPC behaviours are central to an engaging interactive story, but the issues raised by their implementations hinder the proliferation of complex NPC behaviours in commercial games. During the dawn of CRPGs, game designers such as Richard Garriott, the creator of the *Ultima* series, had a revolutionary vision for what interactive stories could become. Richard Garriott drew his inspiration from the “Lord of the Rings” epic.

“The characters had to be believable, or at least exhibit some rudimentary sense of motivation. Plots and stories could be simple - most epics were - but they had to sweep up the player with a sense of urgency. [...] the more the non-player characters built into the world - not simply by dying or giving up cash, but with complex consequences recognizable from real life - the more compelling the world would be.” [42]

Although we focus on behaviours for NPCs in CRPGs, the applications for intelligent NPCs (called intelligent agents in other domains) extend far beyond the realm of CRPGs. Intelligent agents are used in sports games, educational games, training, and hand-held appliances, where complex behaviours may be crucial for success.

2.1 The State of NPC Behaviours

Behaviours for NPCs in computer role-playing games (CRPGs) have been traditionally implemented using scripting languages, state machines, and rule-based systems

[68]. Although scripting languages are supposed to be “higher-level” than programming the game engine directly, in practice they are similar in scope and abstraction level to C or C++. The increasing complexity of behaviours, especially for collaborative NPCs, makes manual scripting impractical [47]. A more serious problem is that the story authors are often not programmers [30][64] and their reliance on programmers for scripting can delay development and introduce errors. This situation results in predictable, repetitive behaviours, where code must be written for each NPC. However, a combination of generative behaviour patterns and reinforcement learning (RL) may overcome most of these problems. The same behaviour pattern can be used to produce scripts for many NPCs, it can be maintained easily, and its parameters can be tuned automatically using reinforcement learning. We will discuss RL techniques in more detail in Chapter 4.

Consider the state-of-the-art for behaviours in recent CRPGs. *The Elder Scrolls 3: Morrowind* [55] has a huge immersive world. However, NPCs either wander around areas on predefined paths or stand still, performing a simple animation, never interacting with each other, and ignoring the simulated day. The success of the sequel game, *The Elder Scrolls 4: Oblivion* [82], is due mostly to the “radiant AI” system that provides more interesting NPC behaviours. An NPC schedules behaviours based on the time of day or on current goals, can initiate or respond to conversations with other NPCs, and remembers previous conversations. The NPCs are provided with a set of initial goals, but they fulfill these goals based on their personalities and the world surrounding them, instead of following a pre-scripted set of actions. However, there is still much room for improvement. There are many situations in the game where NPCs do not react if the PC casts spells, steals from them [73], or even attacks nearby NPCs.

In the *Fable* series [29], the NPCs wake at dawn, walk to work, run errands, go home at night, and make random comments about the disposition and appearance of the PC. Through repeated interaction, a player may influence an NPC’s attitude towards the PC. However, the behaviours and comments are repetitive and NPCs never interact with each other.

In *NWN* [58], a very popular CRPG from BioWare Corp., the NPCs use scripts

to perform repetitive proactive behaviours that do not involve other NPCs, as well as a few simple behaviours in response to the PC's actions (PC-interactive behaviours). For example, the default behaviour of a house guard NPC is walking a fixed path between predefined waypoints. In addition, many of the NPCs included in this game are not scripted, therefore they are not able to interact with any objects in the game. They stand still and only respond to a conversation initiated by the PC, if the author provides them with a conversation file. For example, in the official *NWN* campaign's *Prelude* module, 49 out of 61 NPCs are scripted and in the *Chapter One Finale* module, only 19 out of 47 NPCs are scripted. In addition to having repetitive behaviours, an NPC is not always able to resume its interrupted behaviours if a game event causes the NPC to be moved from its current location. For example, in the *Prelude* module, a trainee thief faces a target (combat dummy) and performs a "pickpocket" skill. If the PC clicks on the thief NPC to initiate a conversation, the NPC turns to face the PC. When the conversation completes, the NPC often fails to return to the target. This occurs if the PC jostled the NPC during the conversation. Unfortunately, the thief NPC tries to perform the skill regardless of the facing and distance from the target. As a result, the trainee tries to "pickpocket" an empty space instead of the combat dummy.

The mechanism for implementing PC-interactive behaviours in *NWN* is ineffective for several reasons. First, the *OnPerception* event triggers a behaviour when the NPC notices a creature (NPC or PC). The disadvantage of using this mechanism is that the *OnPerception* event is fired every time the PC enters the NPC's perception range and it is not fired again as long as the PC remains in the perception range of the NPC. Moreover, if the creature that is perceived by the NPC (e.g., the PC or another NPC) is already in the perceived range when the game starts, then the event is not fired at all. This situation can occur if the game is saved and then later reloaded and played. Second, a behaviour can be triggered when a creature enters or exits an area or a *trigger* (i.e., an object that is represented by a polygon on the floor in the Aurora Toolset, being invisible to the player). For example, one of the NPCs in the *Prelude* is located in a trigger's centre that is represented by a waypoint object (invisible to the player) in the Aurora Toolset. When the PC enters the trigger, the

NPC walks to the PC and starts a conversation. As a mechanism to ensure that the NPC eventually returns to the original location, a script is written for the *OnExit* event on the trigger. When the NPC exits the trigger, the script is executed and, as a consequence, the NPC returns to the waypoint that marks the centre of the trigger. This solution has two disadvantages. First, it involves two more game objects in addition to the NPC (a trigger and a waypoint). Second, if the PC leaves the trigger, the NPC can exit the trigger and return to the original location, instead of engaging in a conversation with the PC. If the PC walks through the trigger area and exits on the other side while avoiding the NPC, the conversation that may be essential for the plot may never occur.

The NPCs in *NWN* do not truly collaborate with each other, although collaboration is simulated through several techniques. For example, although the NPCs in the original *NWN Prelude* do not perform collaborative behaviours per se, the story authors attempted to simulate collaborative behaviours in many scenes. First, six NPCs grouped in pairs mimic a conversation by facing each other and performing independent speaking gestures. Second, two spellcasters cast spells successively on a combat dummy and the appearance of alternation is achieved by applying a delay to one of the NPCs. Third, one NPC spawns a skeleton and another NPC destroys the spawned skeleton. In this case, to handle the lack of true collaboration, one NPC spawns skeletons at fixed time intervals and the other destroys any perceived skeletons. In Chapter 5, we describe the *NWN* behaviour problems in more detail, together with our solutions.

In *The Sims* series [27], players control the NPCs (Sims) by choosing their behaviours. Sims choose their own behaviours using a motivational system if they are not told what to do. Their behaviours are impressive, but they hinge on a game model that is integral to this game and not easily transferable to other game genres, including CRPGs. Will Wright, the designer of *The Sims*, found inspiration for the Sims' behaviours from Christopher Alexander's work on the influence of environmental and architectural design on people's behaviour.

“Essentially the game designer wanted to build a digital dollhouse that would have an influence on the people who lived in it. The team strug-

gled to come up with a way to model human behaviour in a way that would be realistic enough to be fun without veering too deeply into the quicksand of artificial intelligence. A sandwich would broadcast what were essentially advertisements for the ability to make people happy by eating. Individual Sims would have different happiness preferences, so they wouldn't respond to the objects' advertisement zones." [42]

Recently, Will Wright designed *Spore* [71], a game with more depth than *The Sims*. In *Spore*, the player controls the development of a species through five phases: *Cell*, *Creature*, *Tribal*, *Civilization*, and *Space*. The NPC AI is player-centric, with the NPC performing specific roles depending on the player's experience. For example, the NPCs are passive, letting the players initiate interaction and the NPC AI is asymmetric in that the NPCs do not follow the same game rules as the player (e.g., unlike the PC, the NPCs do not grow and they have different goals). However, the AI in *Spore* has not met the expectations of many players. On the official *Spore* forum, one player demanded more developed NPC AI that would "breathe more life into Spore". Another player claimed that the game lacked "any signs of AI at all, in any stage" [72]. The game stages employed various AI techniques, highly customized and driven by aspects of game design, but with no general tool that could be used by story authors. For example, the *Cell* stage used flocking AI for movement, the *Creature* stage used behaviour trees to create behaviours that respond to stimuli, are interruptible and resumable, and can be scheduled, the *Tribal* stage used group behaviours, including a *join* and a *lead* behaviour for the group, the *Civilization* stage used strategic AI (e.g, for buildings) implemented in C++, and the *Space* stage used new techniques for ecology simulation and nine empire personality types. The game still lacks NPCs that change over time and that are reactive not only to the PC, but to other NPCs as well [34].

The *Quake* games [65] employ FSMs (finite state machines) [38][39], one of the most used technology for bot AI. Most games contain an implementation of an FSM, since FSMs are efficient, simple to program, and expressive. However, FSMs do not allow reusability in different contexts, which results in repeated code, complexity, and potential for errors.

As a step further from FSMs, hierarchical finite state machines (HFSSM) [36], used in the *Destroy All Humans! 2* [22] game, group a set of states (*super-states*) that share transitions. A game designer can create transitions to super-states instead of to individual states and can create a hierarchy of super-states. As in the case of FSMs, a game designer cannot reuse states for different situations, since transitions are hardcoded in these states.

Behaviour trees (BT) [40], more general than HFSSMs, arrange behaviours hierarchically (directed acyclic graph) and were used in *Halo 2* [8]. This popular game is a first-person shooter (FPS) with about fifty behaviours arranged in four layers and it includes support for “joint behaviours” [40][85]. *Halo 2*’s general AI model is described, but no model for joint behaviours is provided. However, in general, behaviour trees are hard to create, because of the large number of states or behaviours and they do not provide support for creating interruptible-resumable behaviours per se. Collaborative behaviours are also hard to express using behaviour trees. In *Halo 3* [8], BTs were augmented using a blackboard to share knowledge stored in game props, although the behaviour tree architecture provides no direct support for collaboration [24][23]. A story author still has to understand the behaviour tree, write custom code for behaviours, and debug and adapt behaviours, which may be difficult.

AI planning techniques have been successful in controlling NPC opponents in commercial FPS games, such as Monolith’s *F.E.A.R.* [61] and *Unreal Tournament* [13][87]. For a large number of NPCs and complex behaviours, planning is still computationally expensive. Recent research [41] introduced an approach to offline hierarchical task network (HTN) [67] planning that generates behaviour scripts automatically for the *Oblivion* game [82]. However, the system is designed to be used by game developers and expert story authors. The author needs to design an HTN encoding the game world, create AI packages that implement atomic behaviours (such as eat or sleep for the HTN actions), and specify planning problems together with their initial world states.

Façade [46] has an excellent collaborative behaviour model for NPCs, but there are only two NPCs, therefore it is not clear if it will scale to hundreds or thousands

of NPCs. The NPCs use collections of behaviours called *beats* that can be interrupted by the PC and that create reactive and believable characters. The authors of *Façade* comment on the amount of manual work required from a story author when using their framework. There is a need for a mechanism to provide behaviours of this quality to many more NPCs with minimal work for the story author.

Other related research includes planning, PaTNets, sensor-control-action loops [2][63], and automata controlled by a universal stack-based control system [10] for both low-level and high-level animation control, but not in the domain of commercial-scale computer games. Crowd control research involves low-level behaviours such as flocking and collisions [56] and has been extended to a higher-level behavioural engine [9]. Group behaviours provide a formal way to reason about joint plans, intentions, and beliefs [33]. Constructing believable character behaviours is challenging, therefore developers have focused on particular attributes of conveying NPC realism. Some of these systems do not modify their behaviours according to the NPC's experience. Our system integrates a behaviour learning system into a more general behaviour architecture that selects appropriate behaviours according to the character's experience.

Our approach is dictated by the practical requirements of commercial computer games. ScriptEase behaviour patterns are much easier for non-programmers to use than manual scripts, even if a modular behaviour script library, such as that provided by the *Memetic AI* [53] toolkit, is available. The *Memetic AI* toolkit API is written in NWScript and, although it provides a priority-based system for NPCs in which behaviours ("memes") may be interrupted and resumed, it is still difficult to use by non-programmers. *Lilac Soul's NWN Script Generator* [45] is written in the Delphi programming language and it provides a wizard-like interface that allows an author to navigate through a series of questions before a script is automatically generated. The author must be familiar with basic concepts of the NWScript language in order to adapt scripts and relate scripts to events. *Lilac Soul* does not have any persistent abstract representation of the task that the script is intended to perform. Therefore, if the author decides to abandon this script authoring process before the script is generated, the author's work cannot be partially saved. Epic's *UnrealKismet* [84]

for the Unreal engine is a visual scripting system that supports hierarchies of scripts that can be organized into units. The onus is on the author to create scripts rather than adapt existing scripts. The author can connect graphically simple events and actions created in advance by programmers. However, for a real commercial game with increasing requirements, the behaviour charts can become unmanageable in a visual scripting system. The *TES (The Elder Scroll) Construction Set* used by *Oblivion* [82] has the same problems as the Aurora Toolset used by *NWN*. The authors must become familiar with their respective scripting languages, *TES Script* and *NWScript*, before using them.

A visual scripting tool that, like ScriptEase, addresses story authors who are not programmers has been used in the development of an NBA (National Basketball Association) game [69]. The tool has an underlying data-driven AI system similar to abstract state machines. In this model, a *situation* comprises game conditions under which the situation fires, roles that also include conditions under which the roles fire, as well as specifications for the occurrence of the situation if the roles fire, and behaviour assignments (singular or chains) for each role. Similar to ScriptEase, in this model the NPCs can have various roles that are activated by specific conditions, trigger their underlying behaviours, and can be interrupted based on priorities. In addition, a role can be assigned to several NPCs at once. Analogous to a starred action in *NWScript*, behaviour assignment can start immediately or it can be queued for later execution. However, this tool is not publicly available, therefore a more in-depth comparison could not be performed and it is not clear how tied it is to the NBA game design. The author mentions that parts of the system, such as the perception system (e.g., game state variables like *the distance to the ball*) and the low level animation helpers are still “code-based” rather than data-driven. The author also mentions that the learning curve required by the *Situation editor* is one week for programmers and longer for non-programmers. In contrast, the ScriptEase generative pattern abstraction is easy to understand and to use, enabling Grade 10 high-school students with no programming experience to create interactive stories in four and a half hours following a two-day training period on the use of the *NWN* game, the Aurora Toolset, and ScriptEase [11][12]. Our pattern model shields story

authors from manual scripting and the synchronization issues posed by collaborative behaviours, allowing them to concentrate on story construction. Authors can easily group, manage, and reuse hundreds of behaviour patterns.

2.2 Requirements for NPC Behaviours

Researchers have identified computational and functional requirements [74] for interactive NPC behaviours. We have adapted these requirements to evaluate our work and we defined additional requirements (marked with an asterisk). Our behaviour system should also satisfy the computational and functional requirements of commercial computer games. The main challenges introduced by NPC behaviours in CRPGs are the following:

- **Adaptability** An NPC should be able to adjust its behaviours according to an unpredictable environment, even when the details of the interactions are not completely specified.
- **Clarity/Consistency/Intentionality** The *player* should infer an NPC's behaviour by observing the NPC's actions. General NPC behaviours should be easy to predict based on some reasonable criteria, even if specific actions are unpredictable. For example, in *The Sims* [27], the game designers were looking for behaviours "that seemed plausible at any given time", but not predictable [42]. At the same time, a *story author* should be able to infer the purpose of an NPC behaviour before assigning it to an NPC.
- **Effectiveness** An NPC's behaviours must always seem correct, especially when learning is employed that may cause an NPC to learn inferior behaviours.
- **Robustness** An NPC's behaviours should always work properly, especially in unpredictable, random game environments.
- **Variety** Behaviours for background characters are pivotal in the development of an interactive story, not only because NPCs with complex behaviours enrich a story, but because their versatility engages the player. Repetitious NPC

behaviours lead to monotonous game-play in which the player quickly loses interest.

- **Autonomy*** An NPC should be able to act independently of other creatures in the game (NPCs or the PC).
- **Alertness*** An NPC should be alert whenever it is solicited by the PC or an NPC. The system that implements NPC behaviours should be computationally fast, since the speed of the system is reflected in the NPC's response time.
- **Interactivity*** An NPC should be able to initiate and respond to appropriate interactions with NPCs or with the PC.
- **Reusability*** An NPC's behaviours should be modular and easy to understand and reuse.
- **Scalability*** We define scalability as the ability of a system to work without noticeable degradation in performance when the number of NPCs and the number and complexity of their behaviours increases. The complexity of a behaviour is high if it includes complicated conditions, it involves many game object interactions, or it varies over time. We distinguish three types of scalability. First, the behaviours of NPCs should be displayed without degradation when a large number of NPCs is used in a module (across all areas) or in an area of a module. We call this measure *scalability of NPC instances*. Second, the efficiency of the system should not be adversely affected if either of the following two values increases: the number of behaviours for each NPC (an NPC can display many kinds of behaviours) or the number of types of behaviours across NPCs (many NPCs can display many kinds of behaviours). We call this measure *scalability of behaviours*. Third, a behaviour system must be manageable and easy to use by authors even if a large number of behaviours is needed at a time. We call this measure *scalability of use*.

These requirements will be used in Chapter 5 to evaluate ScriptEase behaviour patterns.

2.3 Chapter Summary

In this chapter, we presented the current state of NPC behaviours as described in the literature and as exhibited by current computer games. We also discussed the problems associated with implementing behaviours in different domains. We outlined some problems with the background and PC-interactive behaviours implemented in the *NWN* game. Although we highlighted the issues present in *NWN*, these issues are representative of problems that occur across all role-playing games and other genres of games as well. In fact, *NWN* is an award-winning CRPG that has fewer problems than most games of this genre. In addition, we outlined the computational and functional requirements for interactive NPC behaviours. In Chapter 5, we evaluate our work by revisiting these requirements in the context of our behaviour system.

Chapter 3

NPC Behaviours

In this chapter, we present an easy-to-use behaviour model for NPCs that requires no manual script writing. This AI behaviour architecture supports responsive collaborative interruptible and resumable behaviours using behaviour queues. The architecture wraps sets of actions into tasks, tasks into behaviours, behaviours into roles, and roles into performances, providing a simple efficient mechanism for encapsulating behaviours into components that can change dynamically, based on environmental criteria. We describe an implementation of this model that generates scripting code for a commercial game, BioWare Corp.'s *Neverwinter Nights* (*NWN*). Two implementations of behaviour patterns were constructed. The first one implemented behaviour patterns using ScriptEase encounter patterns. This implementation was used to prototype the model. A second implementation introduced primitives to represent behaviour patterns natively in ScriptEase, so that the behaviour architecture can be utilized with no coding skills.

The research contributions described in this chapter are the following:

1. A new behaviour model that generates behaviours automatically, without manual script writing, for BioWare Corp.'s *NWN* game.
2. Proactive, reactive, and latent NPC behaviours. Proactive and latent behaviours can be independent (for individual NPCs) or collaborative (for interacting NPCs). Reactive behaviours can only be collaborative.
3. A motivation model that selects proactive behaviours based on motivational

attribute values as an alternative to selecting behaviours based on static probabilities.

4. A collaborative protocol mechanism based on a common topic allows story authors to easily create and reuse collaborative behaviours, without having to know the collaborator until the game is played.
5. Interruptible and resumable behaviours through tasks and queues with different priorities: a latent queue, two collaborative queues, and a proactive independent queue.
6. A performance mechanism that activates different roles of an NPC based on cues.

3.1 Behaviour Model

We developed a mechanism that generates engaging NPC behaviours without explicitly writing code. More specifically, we created a behaviour model that selects a behaviour (i.e., a set of basic actions) performed by an NPC based on its motivations and perception of the dynamic game world. We describe each of the major components of the model and we use our guard example to illustrate how these components work together to provide behaviours that are expressive enough to meet the current needs of CRPGs and intuitive enough for story authors to understand quickly. Each term in our ontology is highlighted in *italics* the first time it appears and, if it is not immediately defined, it is highlighted again when it is defined. We also highlight in **bold** the NPC behaviour components.

We distinguish NPC behaviours on two axes, *independent* vs. *collaborative*, and *proactive* vs. *reactive* vs. *latent*. However, reactive behaviours must be collaborative, as shown in Figure 3.1.

An *independent* behaviour is a behaviour that the NPC performs alone (without the PC or another NPC). A *collaborative* behaviour is a behaviour that the NPC performs jointly with another NPC (not PC). For example, a guard NPC can walk randomly near the guarded object (**Patrol** independent behaviour) or initiate a con-

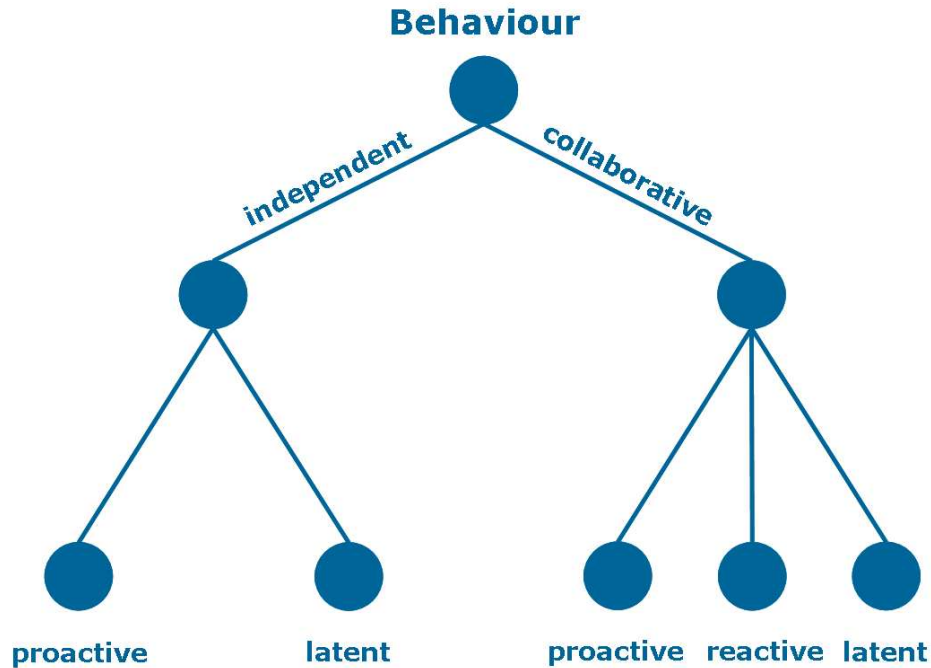


Figure 3.1: The types of NPC behaviours: independent or collaborative, as well as proactive, reactive, or latent.

versation about a *topic* (e.g., “weather”) with another guard (**Converse-Talk** collaborative behaviour). In a **Converse-Talk** behaviour, the initiator NPC speaks and then it listens, while the responder NPC listens and then it speaks. A collaborative behaviour must include concurrency control to ensure that the actions of the two NPCs are synchronized. For example, the second NPC should not start **Speaking** until the first NPC finishes **Speaking**.

A *proactive* behaviour is a behaviour that the NPC spontaneously initiates on a random basis or based on motivations (*proactive* - independent or collaborative). A *reactive* behaviour is one that the NPC performs in response to a collaborative behaviour initiated by another NPC (*reactive* - collaborative). For example, a guard NPC can select from a set of proactive independent behaviours: **Patrol** near the guarded object, **Rest** on a nearby seat, or **Check** that the guarded object is safe. The guard can also initiate a talk with a friendly nearby NPC using a **Converse-Talk** proactive collaborative behaviour. At any time, the NPC chooses from a set of possible proactive behaviours. When an NPC finishes a proactive behaviour, it

selects another proactive behaviour. A proactive behaviour is selected either on a random basis using static probabilities or based on *motivations* that can change dynamically due to game events that affect the NPC. This continues indefinitely or until the NPC is interrupted by an external event, such as an interaction with the player character. To collaborate on a specific topic (discussed in detail in Section 3.5), both the initiator and the responder must not be busy and the responder must have a reactive behaviour (e.g., **Converse-Listen**) on that topic. For example, when the guard NPC tries to initiate a proactive **Converse-Talk** about a topic, there must be an available NPC with a reactive behaviour (e.g., **Converse-Listen**) on the same topic. The guard can also respond with a reactive behaviour **Converse-Listen** on a topic to any creature who has any proactive collaborative behaviour on that topic. We discuss the mechanism for selecting among multiple possible proactive and reactive pairings through registering behaviours with topics in Section 3.5.

A game event may interrupt the NPC while it is performing the current proactive or reactive behaviour (proactive independent, proactive collaborative, or reactive). A *latent* behaviour is an NPC's reaction to an external game event triggered by the player character, another NPC, or a general game event. For example, the PC or another NPC walks within a close range to an NPC or the game time reaches a specific hour. This type of behaviour is not considered proactive or reactive, since it is triggered by events external to the NPC and not initiated directly by another NPC who is attempting a collaboration. Unlike a proactive behaviour, a latent behaviour is not performed in a loop. A latent behaviour is triggered only when an event occurs and certain conditions are satisfied. An event that triggers a latent behaviour can be constructed from any game event such as a timer, a creature coming within some range, or a container being opened. In this case, the NPC responds by performing a latent behaviour. After the latent behaviour is completed, the NPC resumes the interrupted behaviour at the point of interruption, not merely restarting that behaviour.

For example, if a PC or an unfriendly NPC moves close to the guarded object, the guard will interrupt the current proactive behaviour (**Patrol**, **Rest**, **Check**, **Converse-Talk**, or **Converse-Listen**) and it will perform a **Warn** latent behaviour.

The guard warns the intruder and then it returns to the previous behaviour at the appropriate stage. If the guard were in the middle of a conversation with a friendly NPC, then the conversation would be resumed at the appropriate stage. A latent behaviour may also be independent (**Warn**) or collaborative. For example, when the PC approaches the guard NPC, the PC may trigger a latent collaborative behaviour (**Converse-Talk** with topic “clue” that reveals an important story clue to the PC) on the guard. As a result, the guard starts a conversation with another NPC to reveal the clue to the PC. A reactive behaviour may be reused, being triggered in response to a proactive or a latent collaborative behaviour of an initiator NPC.

A *behaviour cue* controls the selection of each proactive behaviour, it responds to requests for reactive collaboration, or it triggers a latent behaviour. For example, each proactive behaviour has a *spin-based* cue. A spin-based cue is a cue that probabilistically selects a behaviour based on either static probabilities or motivational attributes that dynamically bias these probabilities. We call the mechanism for selecting a behaviour a *behaviour dispatcher* and we call a *spinner* the process by which the behaviour dispatcher probabilistically selects a proactive (not reactive or latent) behaviour from its available proactive behaviours.

A single NPC may exhibit a different *role* at different times in the story. For example, an NPC may have a **Guard** role at one time of the day, a **Patron** role later in the day, and a **Sleep** role at the end of the day.

We define an NPC *performance* as a group of roles for that NPC. For example, we group all of a guard’s roles into the **Guard** performance shown in Figure 3.2, where a *P* denotes a performance and an *R* denotes a role. A role contains all of the basic behaviours (proactive, reactive, and latent) that can be performed in a particular context. For example, the **Guard** role includes the behaviours **Patrol**, **Rest**, **Check**, **Converse-Talk**, **Converse-Listen**, **Warn**, and **Exclaim**. At any time, the NPC uses only a single active role to select proactive behaviours (triggered by spin-based cues) or to respond to cues that trigger reactive or latent behaviours. For example, when the **Patron** role is active, the NPC is in a tavern and selects from proactive behaviours that include **Order-drink** and **Converse-Talk** on the topic “drink”. During the **Patron** role, the NPC would never **Warn** or **Exclaim**. A role

may be active during the same time interval each day (or some other temporal unit) or it may become active only when specific events occur in the story, as illustrated in Figure 3.2. The active role of an NPC is changed by a *role cue*, denoted by the > symbol in Figure 3.2. Each role cue is a latent cue. For example, our guard NPC uses a **Guard** role during a specific time interval each day (a timer cue) and has a **Patron** role for an interval each evening (another timer cue). The guard uses a third role, **Sleep**, which contains a single proactive behaviour with the same name that is used at night.

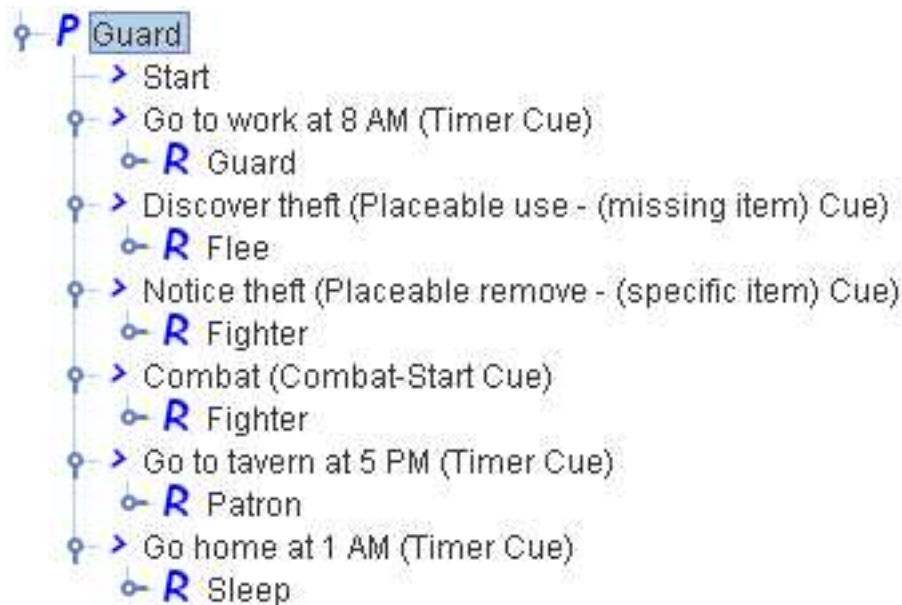


Figure 3.2: The multiple roles of a guard NPC.

A timer-based role cue activates the NPC’s **Guard** role at a specific time each day, changing the NPC’s active role from **Sleep** to **Guard** in the morning.

Another role cue activates the NPC’s **Patron** role, changing the active role from **Guard** to **Patron** at the end of the day. After midnight, the NPC’s **Sleep** role is activated by a timer role cue and, therefore, the NPC goes home to sleep. For each role, the guard needs a behaviour to travel from the current location to the target location. We use a latent *Start* cue to trigger this travel. A *Start* behaviour cue becomes active whenever a role activates or reactivates, as explained in Section 3.4. An **Approach** behaviour causes the NPC to travel to a target location. We include an **Approach**

behaviour in the role's *Start* cue, so that every time the role is activated, the NPC moves to the appropriate location. As a consequence, when a guard NPC finishes its shift, its **Patron** role is activated by a timer role cue that activates the **Patron's Start** behaviour cue. This cue causes the NPC to immediately perform the **Approach** behaviour, i.e., to walk to the tavern. If the guarded item is stolen (*Discover theft* latent behaviour cue), the next time the guard checks the guarded item and finds it missing, a cue changes the guard's role to **Flee**. A *Discover theft* cue is based on the **Placeable use - (missing item)** encounter. A placeable is a generic game object, such as a chest or a statue, with which the PC can interact. The missing item refers to the game object that the NPC was guarding and that was removed from the placeable. As a result, the guard leaves the area to avoid retribution from the guard's captain. Encounter patterns are described in Appendix A. The same effect could be obtained by replacing the **Flee** role with a **Flee** behaviour added in the **Guard** role. However, having a **Flee** role is a more general approach, as the NPC may flee for different reasons (i.e., activated by different cues), not only when the NPC performs the **Guard** role. Most importantly in our scenario, having a **Flee** role instead of a **Flee** behaviour in the **Guard** role ensures that when the conditions of the cue that activates the **Flee** role are satisfied, the NPC does not return to the **Guard** role. If the **Guard** role included a **Flee** behaviour triggered by the same *Discover theft* cue, then the NPC would return to the **Guard** role after the completion of the **Flee** behaviour. Lastly, when the NPC starts a role, the NPC performs that role until another of its roles becomes active. Thus, if the **Flee** role has a few behaviours that the NPC could choose to flee (rather than having one **Flee** behaviour in the **Guard** role), the NPC will execute them, maximizing the NPC's chances to reach its flee destination. If the guard notices the item being stolen (*Notice theft* latent role cue based on the **Placeable remove - (specific item)** encounter), the guard changes its role to **Fighter**. In general, a role or a behaviour can be activated by multiple cues. For example, the **Fighter** role can also be activated by a *Combat-Start* latent role cue, as illustrated in Figure 3.5. This cue is activated when a character perceives or is attacked by a hostile creature. Each NPC has a specific class (e.g., fighter, sorcerer, cleric, etc.). If the class of the NPC is not fighter, then the story author

only needs to switch the **Fighter** role with the **Class** role (e.g., **Sorcerer**, **Cleric**, etc.), using the same *Combat-Start* latent role cue to start the combat.

In order to support rich NPC behaviours, we need to provide independent and collaborative behaviours, as well as proactive, reactive, and latent behaviours.

3.2 Behaviour Patterns

Many computer games use custom scripts to control the behaviours of NPCs. For example, to script a guard behaviour for an NPC in the *NWN* game, an author has to write NWScript code that implements the desired behaviour. Even if the same behaviour is used for another guard NPC, the script may have to be modified to refer to that NPC, since every object in the game has a different identifier. In addition, the script may refer to other objects in the game and their references have to be changed manually to refer to different objects. The difficulties of manual scripting have been documented previously [49]. Figure 3.3 illustrates a fragment from the *OnHeartbeat* script of a guard in the original *NWN* campaign story. This script executes approximately every six seconds during the game and, among other actions, it causes the NPC to walk a set of waypoints. Despite the complexity of the script code, this guard is quite simplistic: it does not choose from a set of proactive behaviours either based on static probabilities or motivations. For example, this guard does not get tired and need to rest.

A guard is a very common NPC in CRPGs. We captured this and other common NPC behaviours into ScriptEase *behaviour patterns*. A behaviour pattern is a category of reusable generative design patterns [31][50]. We have extended the generative pattern approach of ScriptEase to support NPC behaviours. NPC interactions require concurrency control to ensure that neither deadlock nor indefinite postponement can occur and that interactions are realistic. We constructed an NPC interaction concurrency model and built generative patterns for it. For example, we created a behaviour pattern, **Guard**, that can be applied to any NPC in the game that acts like a guard, i.e. patrols a set of patrol points, guards an item, rests on a bench, converses with friendly NPCs, and challenges intruders. In Appendix B,

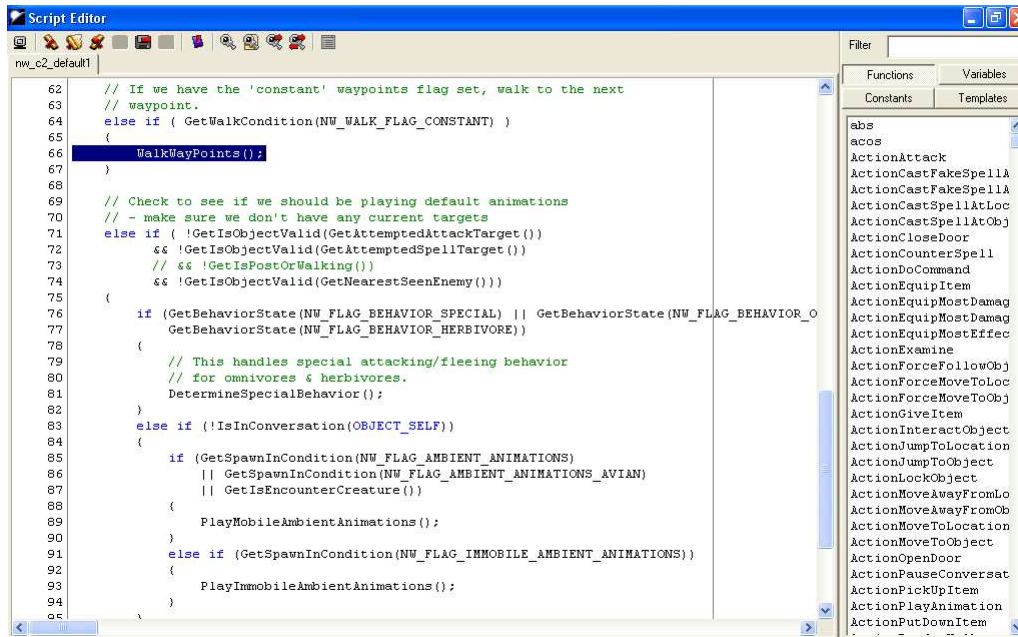


Figure 3.3: NWScript code for an *NWN* guard.

we briefly describe the **Guard** role as part of the catalogue of behaviour patterns that we developed at the initial stages of this research. We realized our model using ScriptEase behaviour patterns. We illustrate how each of the major components of the model (performance, role, proactive behaviour, reactive behaviour, latent behaviour, independent behaviour, collaborative behaviour, task, motivation, and cue) can be represented and manipulated using new ScriptEase constructs. Each behaviour pattern contains easy-to-assemble reusable components.

Behaviour patterns encapsulate proactive, reactive, and latent behaviours for NPCs. In addition, these patterns generate complex and non-repetitive NPC scripts, since they allow the NPC to select from a wide range of behaviours. Our patterns generate scripting code automatically, facilitating the process of authoring an interactive story by non-programmers. Behaviours hide the level of complexity necessary to create a realistic interactive story and can be reused by several NPCs. We constructed a library of behaviour patterns responsible for common interactions among the NPCs and between the NPC and the environment in CRPG settings.

3.3 Use of Behaviour Patterns

A story author begins by using BioWare Corp.'s Aurora Toolset to create the physical layout of a story, without attaching any scripts to objects. An interactive story requires a story author to use the Aurora Toolset to construct an area, populate it with objects, including NPCs, and save it in a module. The author then opens the module in ScriptEase and selects patterns from the available pattern library. To attribute behaviours to NPCs, the author selects appropriate performance patterns and attaches them to the NPCs that were created using the Aurora Toolset. After adapting these patterns to the context of the story, the author saves and compiles the module in ScriptEase, which is then used to generate the scripting code automatically for the NPCs in the story. Appendix A contains an excerpt from a paper [15] that illustrates in detail how an author can create and use encounter patterns. We have implemented behaviour patterns similarly.

For example, to create a guard NPC, the story author uses the Aurora Toolset to construct an area and populate it with a guard, a guarded item, a guarded chest and a seat, and then saves the area in a module. The author then opens the module in ScriptEase to perform three kinds of actions.

1. Create an instance of the **Guard** pattern by selecting the **Guard** performance from a menu, as shown in Figure 3.4. An instance can be attached to any NPC in the game that should assume the guard role.
2. Bind the options of the pattern instance to game objects previously constructed with the Aurora Toolset. Figure 3.4 shows how to set options for the guard NPC at the performance level. This pattern instance is attached to an NPC (*Shara*) using a ScriptEase pick dialog. In addition to picking the `Actor` option at the performance level (highlighted), the author must also pick the options necessary for each of the performance's roles and cues. For example, the **Guard** role requires a `Guarded` option, which is the item that is being guarded, a `Container` option, which is the container that holds the guarded item, and a `Seat` option, which is the prop that the guard can sit on to rest. Figure 3.5 shows how to set the `Guarded` option to the *Cloak of*

Compassion at the role level (highlighted).

3. Compile (*Save and Compile* ScriptEase menu command) the module to generate the scripting code (NWScript) for all of these behaviours. The game story can be played in *NWN*.

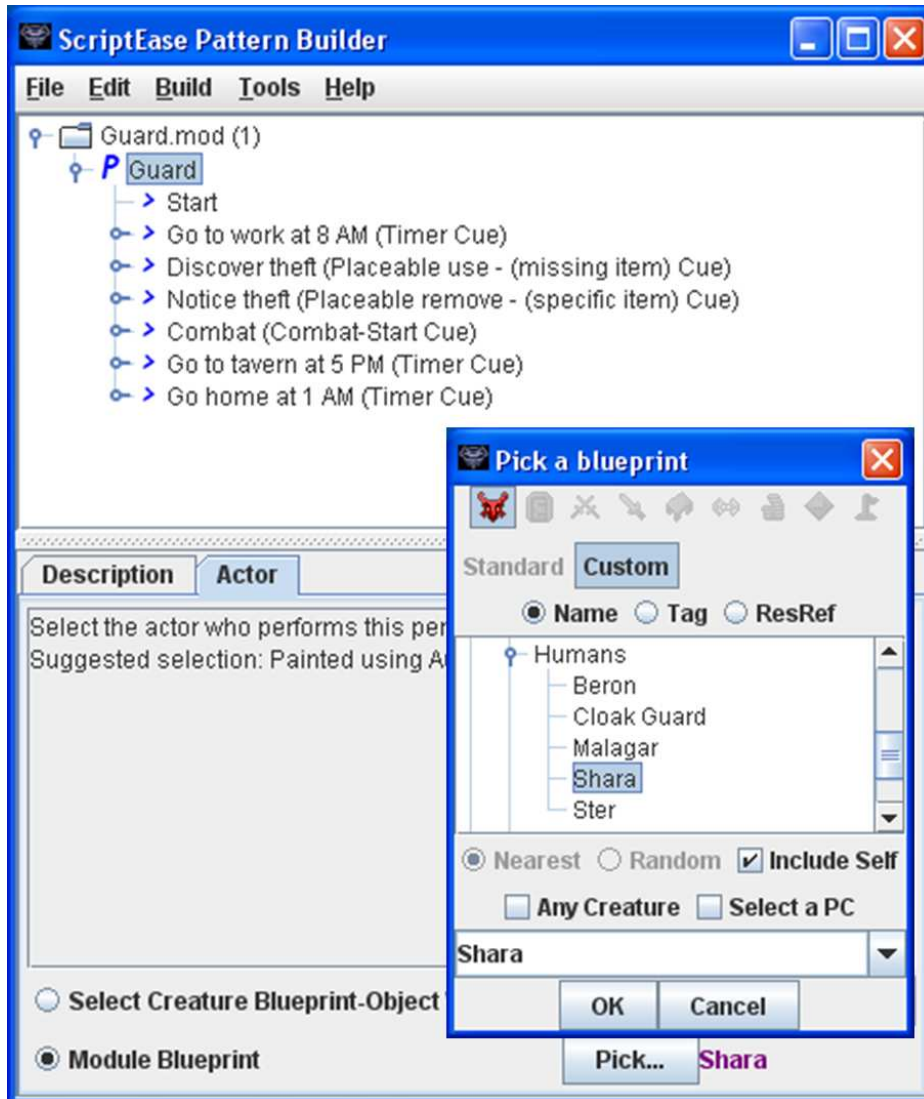


Figure 3.4: Using a performance in ScriptEase: setting the Actor option of a ScriptEase **Guard** performance.

The simplicity of the process hides the fact that a large amount of scripting code is generated to model complex interactive behaviours. This **Guard** pattern generates 1,065 non-comment lines of NWScript code, which is a large amount of

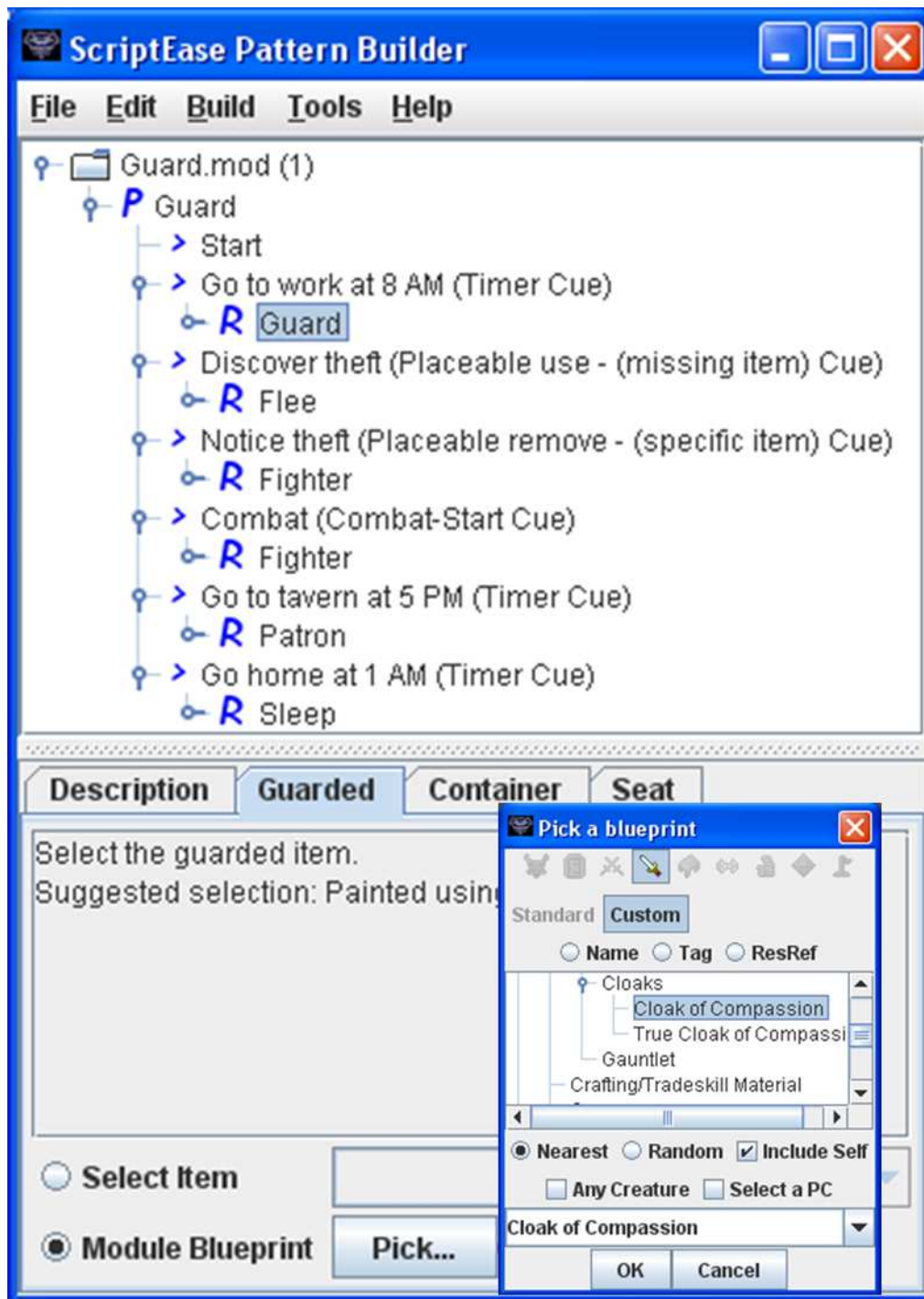


Figure 3.5: Using behaviour patterns in ScriptEase: setting the Guarded option of a ScriptEase role in the **Guard** performance.

game code for the amount of work required by an author to use it. Although there has also been work done in creating this pattern, that work is amortized over the many instances of the **Guard** pattern that occur in many stories (game adventures). In addition to the **Guard** role of the **Guard** performance, the **Flee**, **Patron** and **Sleep** roles have been attached to *Shara*, as illustrated in Figure 3.5.

In this example, the author only sets the options of the roles in the **Guard** performance, but in general more adaptations may be required. The author needs to understand the structure of a behaviour pattern to perform more extensive adaptations or to create new behaviour patterns.

3.4 The Structure of Behaviour Patterns

A behaviour pattern is composed of several levels that can be revealed successively. The top level is called a *performance* and, as shown in Figure 3.6, each performance contains multiple *roles*. An author does not have to open a role, except to adapt it. The **Guard** performance has been opened to reveal some of the components. The **Guard** performance (*P*) has a default latent role cue (\rightarrow), *Start*, that becomes active when the performance starts, i.e., when the NPC spawns in the game. In the case of our guard NPC, there is no role activated by this cue, since the specific behaviours to perform immediately after being spawned are dependent on the time of day. In this case, the timer cues will fire immediately and select the appropriate role for the guard. Figure 3.6 shows that the **Guard** performance includes a **Guard** role (*R*) activated by a *Timer* role cue, **Go to work**, when it is time for the guard to go to work in the morning. This role has been opened to reveal a motivation (*M*), **Guard Motivation**, three proactive independent behaviours, **Patrol**, **Rest**, and **Check** triggered by *Independent Proactive Cues* (i.e., spin-based cues), one proactive collaborative behaviour, **Converse-Talk** triggered by a *Collaborative Proactive Cue* (i.e., a spin-based cue), one reactive behaviour, **Converse-Listen** triggered by a *Reactive Cue*, and two latent independent behaviours, **Warn** and **Exclaim** triggered by *Independent Latent Cues*. Recall that roles have a default *Start* latent behaviour cue that becomes active when the role activates or reactivates. A user-defined event with a

specific number corresponding to the *Start* behaviour cue is fired every time a role changes. This allows the story author to specify a particular behaviour that happens once, every time a role becomes active. As mentioned previously, a **Patron** role could have an **Approach** behaviour triggered by the *Start* latent behaviour cue. Therefore, every time the NPC switches to the **Patron** role, the NPC starts walking to the tavern.



Figure 3.6: The structure of the **Guard** performance in ScriptEase.

Each proactive, reactive, and latent behaviour is represented by a ScriptEase *ab-*

stract behaviour. When an abstract behaviour is instantiated, a story author selects the particular kind of cue that will trigger it. The abstract behaviour is instantiated into a *basic behaviour* of one of three kinds, corresponding to the type of cue selected: proactive, reactive, or latent. Each basic behaviour consists of a cue (\rightarrow), an update clause (U) that updates the NPC's motivations, and a set of tasks (T). Each task is composed of a set of actions (A). The update clause (U) is discussed in Section 3.4.3 in connection with motivations. For example, Figure 3.6 shows the **Rest** proactive behaviour. A proactive cue (i.e., spin-based cue) always triggers a behaviour on the basis of motivations (described later). The **Rest** behaviour contains an update clause (U) and two tasks, **Walk** near the guard's seat and **Sit**. The **Sit** task contains three actions: face the seat, walk to the seat, and sit.

3.4.1 Cues

A cue is an event that triggers a behaviour or activates a role. We distinguish the following types of cues in our model:

- spin cue: it triggers a proactive behaviour. It is based on a user-defined event that spins continuously and it has two flavours: proactive independent cue and proactive collaborative cue. The author may select either of these spin cues to trigger a proactive behaviour (independent or collaborative, respectively). However, the author does not need to adapt these cues, except for specifying their selection probabilities.
- reactive cue: it triggers a reactive behaviour. It is based on a user-defined event that loops continuously checking for whether another NPC is trying to initiate a collaborative behaviour for which this NPC has a reactive behaviour on the same topic. The author does not need to adapt this cue.
- latent cue: it activates a role or it triggers a latent behaviour. It is based on an event specified by the author. The author must specify an event that activates the latent cue. The event can be a standard game event (e.g., *Creature spawn*, *Discover theft*, etc.) or a custom or user-defined event (e.g., *Range* cue).

We have constructed specific latent cues for our model:

- *Start* cue: it activates a role or it triggers a behaviour. It is based on a user-defined event. The author does not need to adapt this cue. In a performance, a *Start* cue becomes active when the NPC is spawned in the game. If the author provides a role at the top level of the performance or a role that is specifically activated by the *Start* cue, then the NPC activates this role. In a role, the *Start* cue becomes active as soon as the role becomes active. If the author triggers a behaviour using the *Start* cue, then that behaviour is the first behaviour to be executed as soon as the role becomes active. This cue is useful for behaviours that the author intends to be executed once, every time the role becomes active, such as walking to a destination.
- *RL* cue: it activates a role or it triggers a behaviour. It is based on a series of standard game events. The author does not need to adapt this cue. This cue is specifically created to include RL into the behaviour model. An experienced author can create variations of the *RL* cue for other RL learning algorithms by adapting the existing *RL* cue.
- *Start-RL* cue: it triggers a behaviour. It is based on a user-defined event. The author does not need to adapt this cue. The *Start-RL* behaviour cue simulates a learning step and it selects a single behaviour for each learning step as soon as a role that uses learning is activated. The *Start-RL* behaviour cue is a refinement of the normal *Start* behaviour cue and it adds support for RL data structure updates.

Cues are more than just events. Similar to actions, tasks, behaviours, and roles, cues provide a powerful mechanism of reuse. They define scope and may encompass options, definitions, conditions, and even actions. For example, a *Range* cue has a user-defined event with a specific number that detects new range events. The cue has a number of options: a `Target` object option, a `Range` float option that specifies the distance threshold between the NPC and the `Target`, and a `Continuous` option that specifies whether the cue should fire only once or repeatedly while the NPC is within the `Target`'s `Range`. The *Range* cue has two

definitions, a *Distance* definition that computes the distance between the NPC and the *Target* object and an *In Range* binary definition that compares the *Distance* value to the *Range* value. In addition, the cue has a condition that tests whether the binary definition is true. If the cue conditions are met, the cue is activated. This cue can be reused after it is constructed and it can be further adapted by changing its options, adding definitions, or adding conditions. The cue may also have actions, if actions are frequently used with that cue. For example, an action can be added to the *Range* cue to trigger a behaviour or even to activate a different cue. We provide examples of such cues in Section 4.5.2 of Chapter 4.

Cues are used for two purposes. First, each role is activated under proper circumstances by a latent role cue. For example, a **Guard** role has a latent role cue that is activated every morning when the NPC goes to work. Each different role is activated by its own latent role cue. For example, the guard NPC may have a second role, **Patron**, whose role cue activates it when the guard's shift ends. When the **Patron** role is active, the guard ignores all the basic behaviours of the **Guard** role and uses the basic behaviours of the **Patron** role, such as **Approach** the bar and **Order-drink**.

Second, a behaviour cue is also used to select a basic behaviour within an active role. All spin-based cues in a single role act together to select a single proactive behaviour. For example, all the spin-based cues (independent and collaborative) in the **Guard** role act together to select from the **Patrol**, **Rest**, **Check**, and **Converse-Talk** proactive behaviours. A reactive cue triggers the **Converse-Listen** reactive behaviour. A latent range behaviour cue triggers the NPC's **Warn** latent behaviour. When the **Guard** role of an NPC is active and when an intruder walks within a certain distance of the guarded chest, the range behaviour cue triggers the **Warn** latent behaviour. A latent (event-based) behaviour cue triggers the guard's **Exclaim** latent behaviour. This event-based cue is activated when the guarded chest is opened by an intruder.

It is possible that the same event can trigger two cues, one of which triggers a behaviour in a role A and the other activates a transition out of role A and into role B. In this case, if the conditions of the role cue are satisfied, role B is activated and

the new behaviour selected in role A is ignored.

Some cues (\rightarrow) are based on encounter patterns (E) and some are new, allowing the author to create custom cues for various game scenarios. For example, the **Warn** latent behaviour is triggered by a new *Near* range cue and the **Exclaim** latent behaviour is triggered by a *Chest opened* cue, which is inspired by the **Placeable open** encounter pattern. Each proactive behaviour, **Patrol**, **Check**, **Rest**, and **Converse-Talk** is triggered by the collective proactive cues, while the reactive behaviour **Converse-Listen** is triggered by a reactive cue initiated by the collaborator. The behaviour cues that trigger the **Warn** and **Exclaim** latent behaviours interrupt the proactive behaviours of the **Guard** role, while the role cues that activate the **Patron** and **Flee** roles change the active role of an NPC.

There are two reasons to base role cues and behaviour cues on encounter patterns. First, many cues occur when an NPC interacts with an object. Encounter patterns were created for this reason and there is a large existing library of encounter patterns that can be reused as cues. For example, there is a **Placeable open** encounter pattern that triggers when a placeable is opened by any character. We would like a cue that triggers when a placeable is opened by anyone other than the guard, so that the guard can notice the theft. Therefore, we can build a cue from the encounter by simply adding a condition that ensures that the opener is not the guard. We use this new latent behaviour cue to trigger the latent **Exclaim** behaviour.

Second, there are two kinds of cues that did not exist as encounter patterns, but both were easy to express as encounter patterns. We constructed a custom cue event for a timer that fires at a specific time each day and a second custom cue event that checks for a creature to be *Near* (within a certain range of) an object. The timer-based cue is used to switch the guard's role from **Guard** to **Patron** at the end of the shift each day. The range cue is used to trigger the latent **Warn** behaviour when an intruder approaches (within a specific distance) the chest containing the guarded item. In our experience, the cue-based timing mechanism, general purpose cues (based on game state), and our motivational scheme for specifying NPC behaviours should be sufficient to meet all the needs of NPC behaviours in CRPGs. For example, they allow us to express the semantics provided by the character AI in *Oblivion*

[82]. Unlike *Oblivion*, our generative pattern technique enables the story author to use a catalogue of predefined behaviour patterns from which behaviour scripts are generated automatically, without manual scripting. In our model, an inattentive *Oblivion* merchant [73] can easily become receptive to the PC or other NPCs, and consequently behave in a more realistic manner, by adding appropriate cues in its behaviour pattern. For example, in this case, we can activate a behaviour or a role in our catalogue using a *Notice theft* cue or a *Discover theft* cue, so that the merchant can react when an item is stolen or when the theft is discovered, respectively.

Our library of cues covers a wide range of game situations, including specific time of day, distance between objects, and perception range. The same cue can be used as a role cue or as a behaviour cue. For example, the range cue was used as a behaviour cue in the **Guard** role to activate the **Warn** behaviour. The range cue can also be used as a role cue to activate a different role when a creature is near the NPC. For example, when the PC approaches an NPC, the range cue switches the NPC's role to a **Rumour** role, so that the PC can overhear an important plot clue.

Since the events created for these types of cues are quite general, we can use them in encounters as well. Encounter patterns can be constructed based on the cue events for situations where certain common actions are executed, such as spawning a monster when the time/distance changes.

3.4.2 Performances and Roles

A performance subsumes all the behaviours of an NPC and it is set when the NPC is spawned in the game. If an NPC has more than one performance, the behaviour system will merge all the roles of these performances. In this case, ScriptEase uses a performance that contains the union of all roles in all performances. If a performance has several active roles, one of these roles is selected randomly. If a role selects more than one appropriate behaviour, one of these behaviours is selected randomly.

The role cues in a performance employ an inheritance mechanism. Any role located at the top level of a performance (i.e., outside all role cues) is inherited by all the cues. This mechanism provides the NPC with more variety in selecting

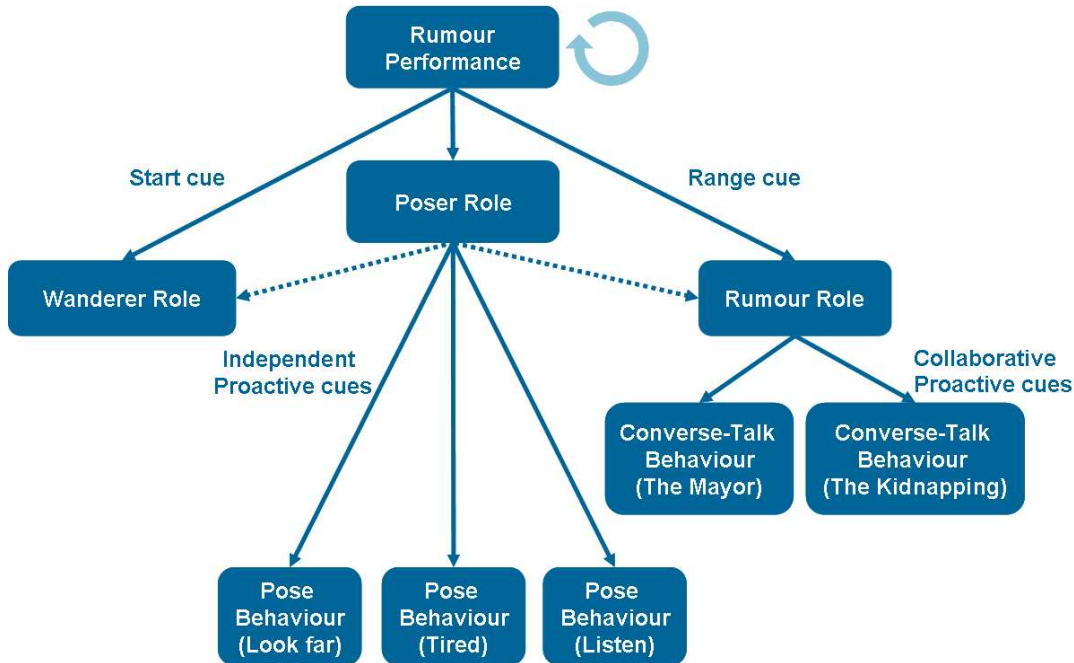


Figure 3.7: The cues that activate the components of a performance.

roles and consequently generates richer behaviours. For example, a role that contains behaviours that can execute under any circumstances can be included at the top level of a performance and can thus ensure that the NPC performs behaviours at any time, under any cue. Figure 3.7 shows the **Rumour** performance in which the **Poser** role is located at the top of the performance (the arc between the performance and the role is not labeled) and thus inherited by the *Start* role cue and the *Range* role cue. This can also be seen in the ScriptEase performance shown in Figure 3.8. As mentioned previously, each performance has a default *Start* role cue that is implemented as a *Creature spawn* encounter-based cue. Therefore, whenever the creature is spawned, its *Start* role cue is activated and one of the roles in the *Start* role cue is selected randomly and it becomes active. In Figure 3.8, the NPC randomly selects between two roles, the **Wanderer** role and the inherited **Poser** role. The *Start* role cue is very important for an author who designs frequent scenarios in which the NPC is required to perform a role when the creature spawns in the game. When the *Range* role cue in Figure 3.8 fires, the NPC randomly selects between the **Rumour** role and the inherited **Poser** role.

Inside a role, behaviours are selected based on the selection values and the priorities of their behaviour cues. A proactive behaviour cue selects randomly among the proactive behaviours that it has available. For example, the rumour NPC can perform a few **Converse-Talk** proactive behaviours on different topics (e.g., “The Mayor” and “The Kidnapping”).

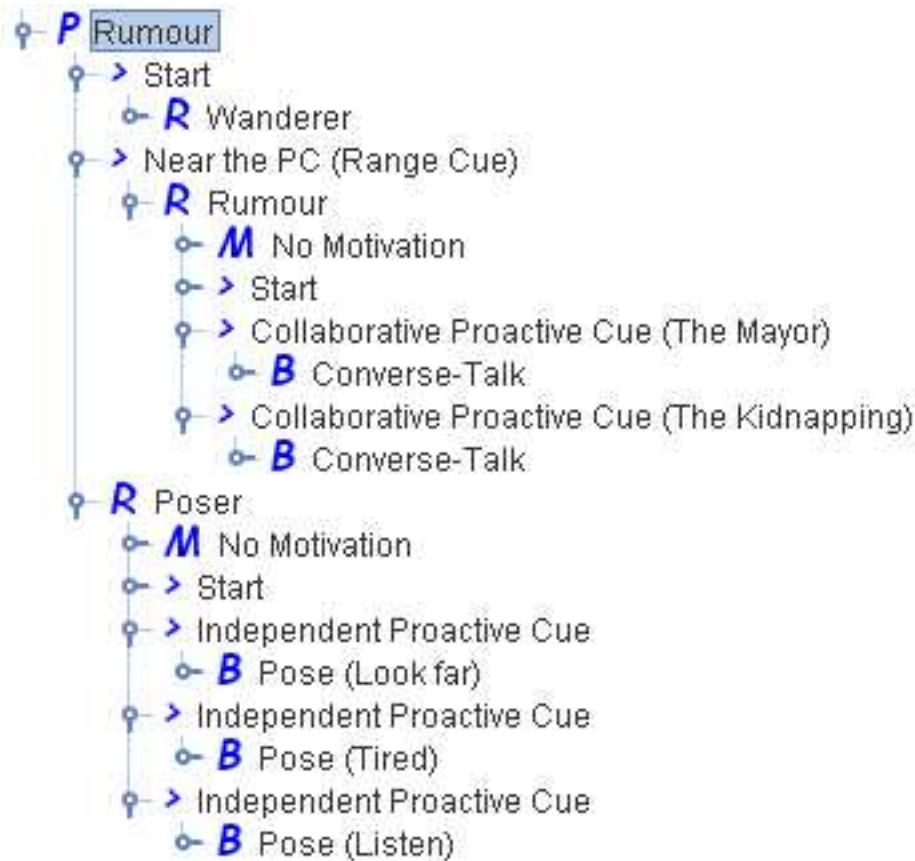


Figure 3.8: The components of a simple **Rumour** performance in ScriptEase.

The *Start* role cue of the **Rumour** performance selects randomly between the **Wanderer** role and the **Poser** role, if the PC is not within the range distance of the rumour NPC when the game starts. Therefore, the **Rumour** role is not active. Then, as soon as the PC walks near the rumour NPC, the *Range* role cue activates the **Rumour** role.

The set of behaviours in the **Rumour** role consists of two **Converse-Talk** behaviours on different topics (e.g., “The Mayor” and “The Kidnapping”), as shown in Figure 3.7 and Figure 3.8.

The rumour NPC selects roles either using static probabilities provided by the author or uses motivations, as described in the next section.

3.4.3 Motivations

Static probabilities are enough to model many NPC proactive behaviours. However, we developed a more expressive motivational model in which a behaviour supports motivational selection of proactive behaviours. For example, a simple guard may have static probabilities for choosing among **Patrol**, **Rest**, **Check**, and **Converse-Talk**. In this case, a simple spin-based behaviour cue is used to select proactive behaviours based on static probabilities. A more sophisticated guard could choose among these behaviours based on motivations, such as **Duty**, **Tiredness**, and **Threat**, that reflect the predilection of the NPC to select any proactive behaviour, such as patrol, rest, check, or initiate a conversation. The motivational construct (M) and the update clause (U) are used to support our motivational model.

The active role of an NPC uses a motivation (M) to select the next proactive behaviour. The simplest (default) motivation is named **No Motivation**. If the default **No Motivation** is used in a role, then each proactive behaviour will have a fixed static probability of being chosen whenever the behaviour “spins” to select a new proactive behaviour. In this case, each proactive behaviour in the active role has an option called `Selection` that represents the relative probability of selecting this proactive behaviour. However, before assigning actual probabilities for the spin, each proactive behaviour is checked to see if its conditions are satisfied (or if it is *possible* for that spin). For example, a condition could be added to the **Check** behaviour that requires the guard to be within a certain distance of the guarded container for this behaviour to be selected. If a condition is not satisfied at the time of a spin, the **Check** behaviour is not included when the relative probabilities are normalized. For example, if the current `Selection` values of **Patrol**, **Rest**, **Check**, and **Converse-Talk** are 20, 1, 4, and 4, but the **Check** behaviour is not currently possible, then the probability vector for selecting these behaviours for this spin would be $(\frac{20}{25} = 0.8, \frac{1}{25} = 0.04, 0, \frac{4}{25} = 0.16)$. When a pattern designer created the **Guard** pattern, default values were set for these proactive behaviours. When an

author creates a pattern instance that includes a **No Motivation**, the default static selection values appear. If an author desires to change these probabilities, the **Guard** pattern instance can be adapted by opening the pattern to reveal the behaviours and modifying the `Selection` option values for these behaviours. Note that the selection values are automatically normalized within the behaviour to compute the probabilities after all conditions are checked.

With static probabilities, the selection values for **Patrol**, **Rest**, **Check**, and **Converse-Talk** are always 20, 1, 4, and 4. After the execution of a proactive behaviour, these values are not updated. Past execution of behaviours based on static probabilities does not have an effect on future behaviour selections. The probability vector of a behaviour, p_i , where $i \in [1, n]$ and n is the number of possible proactive behaviours in the current role, represents the vector of selection values for the possible proactive behaviours of a role. The proactive spinner computes a random number larger than zero and smaller than the sum of all possible proactive behaviours, $\sum_{i=1}^n p_i$, divided by the number of possible proactive behaviours, n for normalization. Normalization is necessary, since we did not want to force the authors to select either a set of probabilities that added to 1 or a set of numbers that added to 100%. If the random number generated is within the $\left[\sum_{j=1}^i p_j/n, \sum_{j=1}^{i+1} p_j/n \right)$ interval, then the i -th proactive behaviour is selected for execution.

Simple static probabilities are sufficient to provide non-deterministic behaviours for many background NPCs. However, for the NPCs whose behaviours are most closely aligned with the story, the PC may have to predict the NPCs' behaviours based on some reasonable rational criteria. For example, if the PC intends to use stealth to remove the guarded item from the chest, the PC may want to wait until the guard is tired and would have a greater probability of resting. Our model includes motivations that can be used for those NPCs that should select their proactive behaviours based on dynamic criteria. Our behaviours support NPC motivations that reflect the NPC's personality and that change dynamically during the game. These motivations influence the way the NPC selects the next proactive behaviour, since each proactive behaviour executes with a probability that is based on the NPC's current motivations. A motivational model allows the NPC to select a proactive be-

behaviour based on the values of attributes in the motivation. At any time, the attribute values are stored in the motivation vector and updated as the game progresses.

We can modify the **Guard** behaviour to select proactive behaviours based on motivations rather than on static probabilities. We replaced the default motivation (**No Motivation**) in the **Guard** behaviour by a **Guard Motivation** that contains three attributes: **Duty** for **Patrol**, **Tiredness** for **Rest**, **Threat** for **Check**, and a combination of **Duty** and **Tiredness** for **Converse-Talk**. The attribute **Duty** refers to the sense of duty that the NPC feels at any time and it can change during the guard's shift. The attribute **Tiredness** refers to the fatigue caused by the guard's work (patrol and check). The attribute **Threat** refers to the sense of threat that the NPC feels when it did not patrol or check the chest, or when an intruder is nearby. When the guard's sense of duty is elevated or when the guard is tired, the guard will try to converse with another NPC. However, when the guard feels threatened, no conversation will be initiated.

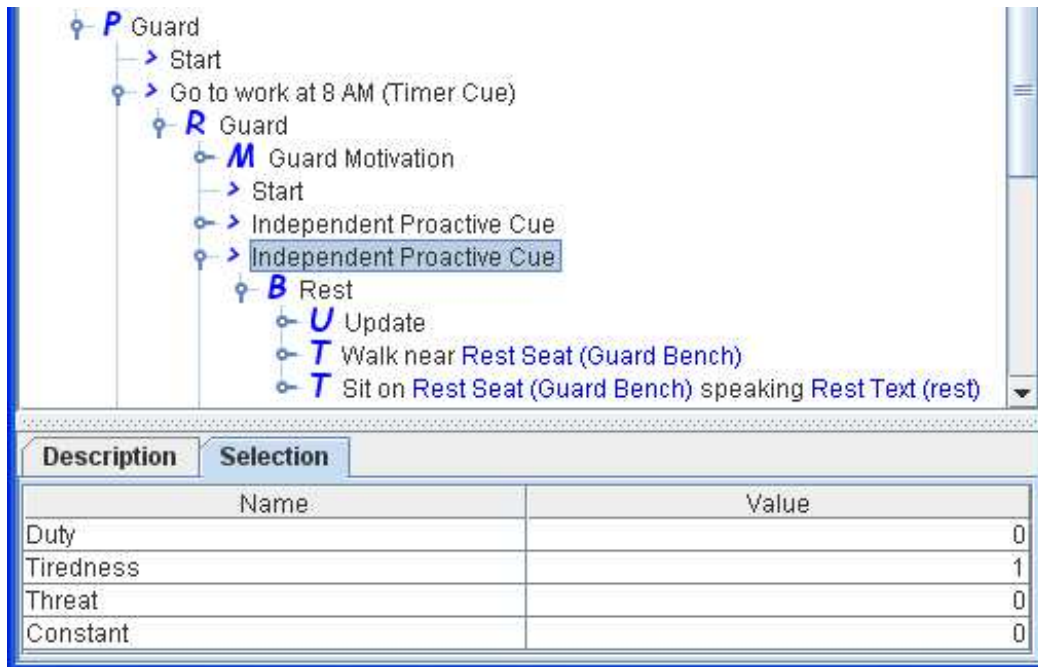


Figure 3.9: The proactive vector used to determine the probability of selecting a proactive behaviour.

The selection value for a proactive behaviour is a constant plus the dot product

of the *motivation vector* and the *proactive vector*. The *motivation vector* consists of the current values of the motivation attributes. The *proactive vector* contains one weight, w_i , for each motivation attribute that reflects how important this attribute is in selecting the proactive behaviour. For example, the **Rest** constant and proactive vector for the guard are shown in Figure 3.9. The constant, $c_r = 0$, is added to the dot product of the **Rest** proactive vector of motivational weights, $w_{Rest} = (0, 1, 0)$, with the current motivation vector, $m = (\mathbf{Duty}, \mathbf{Tiredness}, \mathbf{Threat})$, to obtain the current selection value for the **Rest** proactive behaviour. The selection value, $S(Rest)$, is computed using Equation (1).

$$(1) S(Rest) = c_{Rest} + w_{Rest}m.$$

The default weights and constant in the proactive vector are set by the pattern designer, as illustrated in Figure 3.9 for the **Rest** proactive behaviour. The author can change these values if desired. If the current motivation vector is (20, 30, 15), representing the values 20 for **Duty**, 30 for **Tiredness**, and 15 for **Threat**, then the selection value for the **Rest** proactive behaviour would be: $0 + (0, 1, 0) \cdot (20, 30, 15) = 30$. In this case, the selection value for **Rest** is just the current value of the **Tiredness** attribute, indicating that the more tired the NPC feels, the more the NPC rests. The simplest motivation is one in which a proactive behaviour depends on a single attribute. The **Converse-Talk** behaviour is an example of a behaviour with a more complex proactive vector. Our guard will converse with another guard when the sense of duty or tiredness is high, but not when the threat is high. Therefore, the proactive vector is (1, 1, 0). If the constant and motivation vector for the **Converse-Talk** proactive behaviour were 0 and (20, 30, 15), then the current selection value for this proactive behaviour would be $0 + (1, 1, 0) \cdot (20, 30, 15) = 50$. The selection value is converted to a probability by first normalizing the selection values for all proactive behaviours, as in the static case.

Once selected for execution, a proactive behaviour updates the NPC's motivations, because the behaviour potentially changed the state of the NPC. For example, after a **Patrol**, the **Duty** decreases because the guard has recently patrolled. A guard who patrols for an extended period of time becomes more and more tired with each patrol. The motivational system reflects this by increasing the **Tiredness** attribute.

The value of the **Threat** attribute slightly increases as well, since the NPC has not checked the chest while patrolling. Since the **Threat** attribute is used to select the **Check** behaviour, a high value for **Threat** translates in a better chance of choosing a **Check** behaviour over **Patrol** or **Rest** and potentially discovering the theft when the intruder is near, but not visible to the guard NPC.

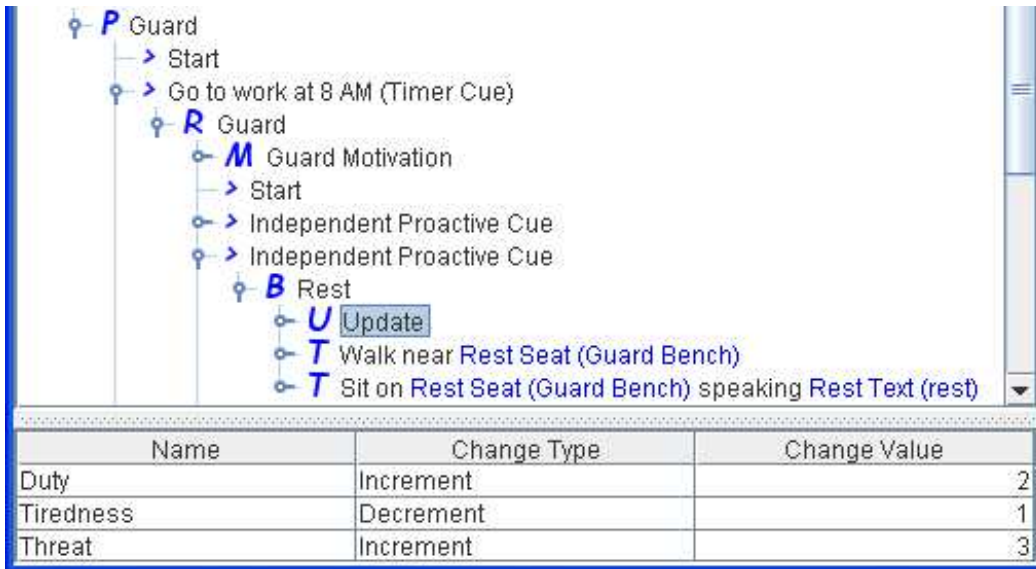


Figure 3.10: The update of the **Rest** proactive vector: the values of the **Duty** and **Threat** motivational attributes increase, while the value of the **Tiredness** attribute decreases.

An NPC starts with a set of initial values for the motivational vector, representing the NPC predilections at the beginning of the game. In time, these values change. Figure 3.10 shows the **Update** clause (*U*) for the **Rest** proactive behaviour. An author can edit the **Update** clause during pattern adaptation if desired. After the **Rest** behaviour, the **Tiredness** is reduced, while the **Duty** and **Threat** are increased, since the guard has not been on patrol for a while and has not checked the chest. A guard who rests is more likely to react slower if an intruder appears. Therefore, the value for the **Threat** attribute increases, as illustrated in Figure 3.10 and by the size of the increasing bar that represents the **Th** vector in the resting panel (center image) of Figure 3.11.

Our model ensures that the values of the motivation vector range between 0 and 100. If the guard executes the **Check** behaviour, the value of the **Threat** attribute



Figure 3.11: The guard NPC is motivated by **Duty (D)**, **Tiredness (Ti)**, and **Threat (Th)** as it patrols (left), rests (center), and checks (right). The bars show the changes of the motivational attributes.

could be reduced to zero. Other events in the game can also update the motivation vector. The latent and reactive behaviours are not selected based on static probabilities or motivations. However, they can still affect the NPC’s motivations, if that NPC uses a motivational model. For example, whenever the NPC warns an intruder, the **Warn** latent behaviour may increase the NPC’s **Threat** attribute.

Behaviour patterns support *proactive*, *reactive*, and *latent*, as well as *independent* and *collaborative* behaviours for NPCs. The ScriptEase behaviour patterns generate complex non-repetitive NPC behaviours, since they allow the NPC to select from a set of proactive behaviours dynamically, based on the NPC’s current motivations and the world state. Therefore, there is significant variation in choosing a behaviour to perform at any time. We developed a mechanism in which we can select behaviours based on learning. We discuss this mechanism in detail in Chapter 4.

3.4.4 Basic Behaviours

A basic behaviour can be used in one of five modes inside a behaviour, depending on the behaviour cue that activates it:

1. a *proactive independent* behaviour can be initiated by the NPC independently of other NPCs or the PC,
2. a *proactive collaborative* behaviour is initiated by an NPC and requires an-

other NPC,

3. a *reactive* behaviour is executed when the NPC responds to a collaborative behaviour initiated by another NPC,
4. a *latent independent* behaviour is performed in response to an external game event, and
5. a *latent collaborative* behaviour is also performed in response to an external game event, but it requires a collaborator.

A basic behaviour can contain conditions that must be satisfied before it can be selected. Each basic behaviour includes one or more tasks (*T*) that are performed in order, when the conditions of the basic behaviour are satisfied. For example, as illustrated in Figure 3.6, **Patrol** is used as a proactive independent behaviour. **Converse-Talk** is used as a proactive collaborative behaviour and **Converse-Listen** is used as a reactive behaviour, since they involve two NPCs who take turns speaking. **Warn** is used as a latent independent behaviour triggered when an intruder (PC or NPC) approaches the guarded item.

The same basic behaviour can be used in different modes by activating it with a different behaviour cue. For example, the proactive collaborative **Converse-Talk** behaviour can be used as a latent behaviour (latent collaborative behaviour) if the author's intent is to trigger a conversation between two NPCs when the PC approaches one of the NPCs (using a range cue). This ensures that the overheard conversation between these two NPCs provides a clue to the nearby PC. This has the same functionality as the **Rumour** role, but in this case, the **Converse-Talk** behaviour is inserted into the NPC's regular **Guard** role.

3.4.5 Proactive Behaviours

A *proactive* behaviour can be selected by an NPC as an independent behaviour or as a collaborative behaviour. For example, a proactive independent behaviour may cause a guard to **Patrol**, which invokes a **Patrol** task. A different proactive independent behaviour may cause a guard to **Rest**, which in turn invokes two tasks: **Walk**

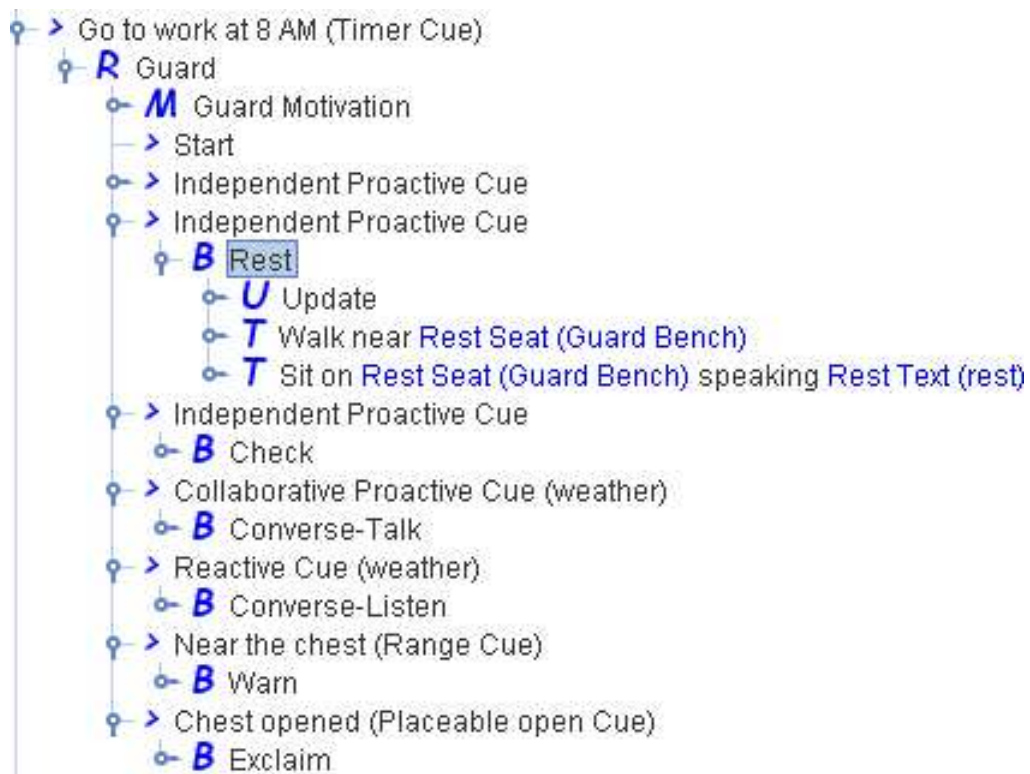


Figure 3.12: The **Rest** proactive independent behaviour of the **Guard** role.

and **Sit**, as illustrated in Figure 3.12. A behaviour triggered by an *Independent Proactive Cue* is both independent and proactive. The **Walk** task is an **Approach** task (used in an **Approach** behaviour) in which the NPC walks to a target. The possible movement option in an **Approach** task are walking or running.

A proactive collaborative behaviour causes the guard to **Converse-Talk**, i.e., initiate a conversation with another NPC, which invokes two tasks, **Speak** and **Listen**. The **Listen** task is a **Pose** task that uses a “Listen” animation. A behaviour triggered by a *Collaborative Proactive Cue* is both collaborative and proactive.

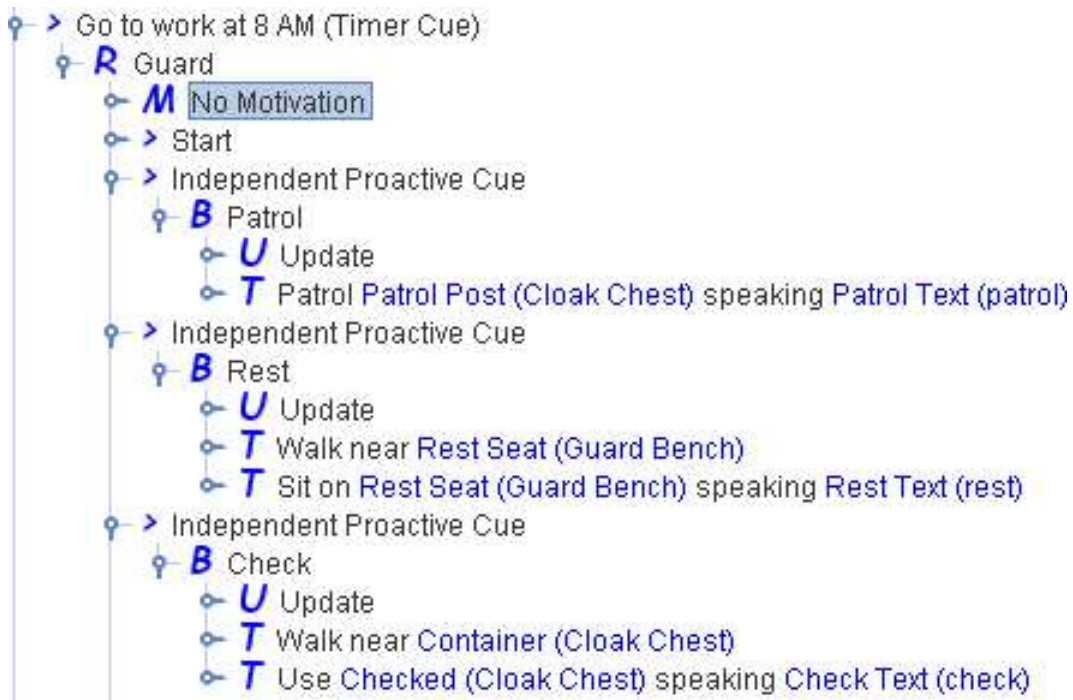


Figure 3.13: The **Guard** role reveals three proactive independent behaviours.

The guard NPC has a **Guard** role that consists of three proactive independent behaviours: **Patrol** near the guarded container (e.g., a chest) in a non-deterministic manner, **Rest** on a seat (e.g., a bench), and **Check** the container for the guarded item (e.g., an amulet). Fig 3.13 shows the **Guard** behaviour with three proactive independent behaviours, **Patrol**, **Rest**, and **Check** composed of their respective sets of tasks, **Patrol**, **Walk/Sit**, and **Walk/Use**. Latent behaviours can be independent as well. For example, both of the guard’s latent behaviours, **Warn** and **Exclaim**,

are independent behaviours. The **Exclaim** behaviour consists of a **Speak** task. We discuss latent behaviours in detail in Section 3.4.7.

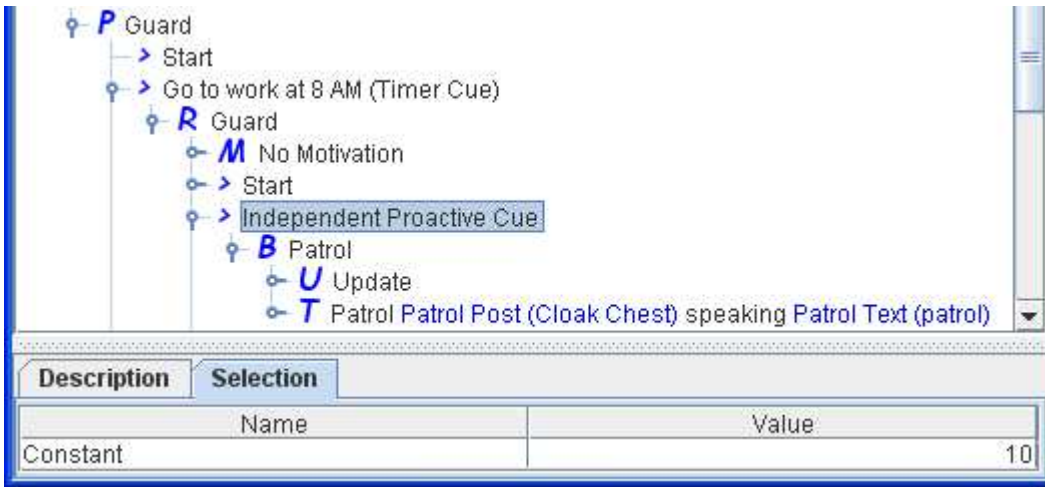


Figure 3.14: Setting the selection probability for the **Patrol** proactive independent behaviour of the **Guard** role.

The NPC repeatedly selects one of the four proactive behaviours, of which three are independent, and executes it. The simplest guard uses a static probability (default **No Motivation**) to select a specific proactive behaviour, as illustrated in Figure 3.13. In this case, the relative probabilities of the **Rest**, **Check**, and **Converse-Talk** (the latter is a proactive collaborative behaviour, therefore it is not shown in Figure 3.13) behaviours are 1, and the relative probability of the **Patrol** behaviour is 10, as illustrated by the **Selection** tab of the **Patrol** behaviour in Figure 3.14. This means that the guard will patrol about ten times more often than it will rest, check, or converse. The **Patrol** behaviour is opened to show its task, **Patrol**, as illustrated in Figure 3.15.

3.4.6 Reactive Behaviours

A *reactive* behaviour is triggered by a proactive counterpart in a collaborative behaviour. For example, a guard can initiate a conversation **Converse-Talk** on a topic with a fellow guard, but it can also respond to this conversation by providing a reactive collaborative behaviour, such as **Converse-Listen**, on the same topic. The tasks invoked by the **Converse-Listen** reactive behaviour are **Listen** and **Speak**. When

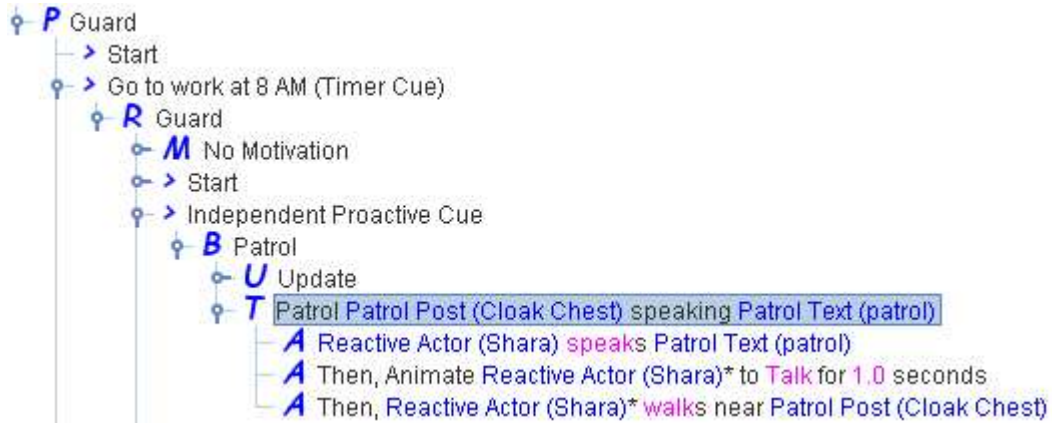


Figure 3.15: The **Patrol** task of the **Patrol** proactive independent behaviour.

performing a concerted collaborative behaviour such as a conversation in which the initiator performs a **Converse-Talk** proactive collaborative behaviour and the responder performs a **Converse-Listen** reactive behaviour, the initiator talks while the responder listens and vice versa. A behaviour triggered by a *Reactive Cue* is a reactive behaviour. A reactive behaviour can be triggered by either a proactive collaborative or a latent collaborative behaviour that was previously triggered by a cue on the collaborating NPC.

3.4.7 Latent Behaviours

An *Independent Latent Cue* or a *Collaborative Latent Cue* can trigger a *latent* behaviour. Although a proactive behaviour happens probabilistically or based on motivations, a latent behaviour executes every time a cue triggers it, after which the NPC resumes any interrupted behaviour.

For example, a **Guard** role has a **Warn** *latent independent* behaviour that is triggered by a range cue when the PC or another NPC walks close to the guarded chest. If the distance between the intruder (an NPC or a PC) and the guarded container is less than some threshold, the range cue triggers the **Warn** behaviour. The **Guard** role also has an **Exclaim** latent independent behaviour that uses a *Placeable open* cue (triggered when an intruder opens the guarded chest). The guard responds to this cue by uttering a remark (selecting a random one-liner from a conversation

file) when the chest opens, even if the intruder is not seen.

When a guard is approached by the guard's captain, the guard could start a **Converse-Talk** *latent collaborative* behaviour with the captain on the topic "report", with the purpose of reporting the daily activities to the captain. The captain could respond with a reactive behaviour **Converse-Listen** on the "report" topic. Note that the same **Converse-Talk** basic behaviour is reused for both the proactive collaborative behaviour **Converse-Talk** with topic "weather" and the latent collaborative behaviour **Converse-Talk** with topic "report", the only difference being the topic employed by each basic behaviour. Recall that when the PC walks near the guard rather than the guarded chest, the guard may perform a **Converse-Talk** behaviour in which it approaches a nearby NPC and starts a conversation with it to provide the PC with a hint.

Latent behaviours have priority over proactive (independent or collaborative) and reactive behaviours. Their priority status allows them to interrupt a current behaviour and allows the interrupted behaviour to resume its execution after the latent behaviour is completed. For example, when a range cue activates a **Warn** latent behaviour, the guard stops the current behaviour and starts executing the **Warn** behaviour. When this behaviour is completed, the guard resumes the interrupted behaviour by restarting the task that was interrupted.

Each latent behaviour has a priority. When a behaviour cue activates a new latent behaviour with a higher priority than a currently executing latent behaviour, the new behaviour cancels the current latent behaviour. If the priority of the new latent behaviour is lower or equal than the currently executing latent behaviour, then the new latent behaviour is ignored. For example, when a **Warn** latent behaviour is executing, if an event-based cue triggers an **Exclaim** latent behaviour, the **Exclaim** behaviour interrupts the **Warn** behaviour since its priority is higher. Latent behaviours can be independent or collaborative. Due to priority levels, a latent independent behaviour could cancel a latent collaborative behaviour. For example, when the PC walks close to a patron, it can cause a collaborative **Overhear-Talk** behaviour in which the patron approaches a random (or the nearest) patron and starts a conversation. If a PC walks very close to a pair of NPCs engaged in a latent

conversation (low priority, e.g., **Overhear-Talk**), the NPCs stop their conversation and greet the PC (high priority). When this latent independent behaviour completes, the NPCs pick one of their possible proactive behaviours. If the conditions for the initial latent collaborative behaviour (**Overhear-Talk**) are satisfied again, the NPCs interrupt their active behaviour (if any), and start a latent conversation. In this particular case, the NPCs may perform this conversation immediately after they greet the PC, since the player is likely to be close to one of the patrons after the latent independent behaviour completes, giving the player the illusion that the patrons resumed their dialogue.

There is no structural difference between proactive, reactive, and latent behaviours in ScriptEase. They are each represented as basic behaviours and the same library of ScriptEase basic behaviours can be reused in any of the three capacities.

3.4.8 Tasks

A *task* (T) consists of basic low-level components, such as a series of actions, definitions and conditions. A task is designed to express a basic rational behaviour and to maximize reuse. For example, a **Give** task consists of a Boolean definition (*has item*) that is true if the NPC has the item that is given away, a condition (*if positive*) that tests if the definition is true, and three actions: move to the target (the recipient), face the target, and give the item to the target. A simple task, **Sit**, consists of three actions: sitting on a prop (the seat), performing a speech animation, and uttering a text.

When a behaviour is executed, the actions of the tasks are added to the NPC's *action queue*. For the *NWN* engine, all objects (except for modules, areas, and items) have a data structure called an action queue that is composed of all the pending actions attributed to the object. An object executes actions from its own action queue in order, each action executing only after the previous action is completed. Each action that will be performed by that object is added at the end of the action queue and it is executed in a *first in, first out* (FIFO) fashion. For example, NWScript provides a few movement functions, such as `ActionMoveToObject` or `ActionMoveToLocation`. In contrast with most NWScript functions, the

movement functions take more time to complete. For this reason, the movement functions are represented as actions, not commands, i.e., they are added to the action queue. When an NPC moves across a large room, the NPC cannot control the processor for this entire time at the expense of all other game objects. Therefore, the technique used by the *NWN* game engine to circumvent this problem is to add the movement function on the action queue and to leave it on the action queue if the destination has not been reached.

ScriptEase provides advanced authors with an *Atom Builder* - the interface between the implementation language (NWScript) and the ScriptEase pattern building blocks - that the pattern designers and programmers can use to create actions that ultimately compose patterns. The *Atom Builder* is included in the ScriptEase *Pattern Designer* tool that pattern designers can use to create new patterns from a set of basic pattern components, using the same menu-driven techniques for adapting existing patterns.

Tasks ensure synchronization within a collaborative behaviour, as well as atomicity within basic behaviours. One advantage of using tasks over actions is a concurrency issue: tasks ensure synchronization within a collaborative basic behaviour. For example, a tavern server starts executing an **Offer-fetch** collaborative behaviour together with a patron. The server offers (e.g., **Speak**) the patron a drink. If the patron accepts the offer (e.g., **Speak**), the server walks to the bar (e.g., **Approach** the bar), picks up the drink (e.g., **Pose**), walks back to the patron (**Approach** the patron), and gives the drink to the patron (e.g., **Give**). Suppose that the server had basic behaviours composed of actions instead of tasks. When the execution of the basic behaviour starts, these actions are all added to the NPC's action queue and executed regardless of the collaborator's actions. Therefore, the server may walk to the bar to fetch a drink before the patron accepts the offer. The use of tasks solves this problem, since we developed a mechanism (Section 3.5) that allows each task to be executed only when certain criteria are met. In this example, after offering to fetch a drink for the patron, the server proceeds to the **Approach** task only after the patron accepts the offer (by performing a **Speak** task).

Another advantage lies in the execution of a basic behaviour. If a behaviour is

interrupted to perform another behaviour, the NPC starts executing the interrupted task when the interrupting behaviour is finished. For example, if the guard's **Rest** behaviour that consists of the **Walk** and **Sit** tasks is interrupted during the execution of the **Sit** task to warn an intruder, then only the **Sit** task is restarted after the **Warn** behaviour is completed. Suppose the **Rest** proactive behaviour is implemented as a set of actions (a contiguous block of actions instead of two tasks). In this case, when an interruption occurs, the whole proactive behaviour is restarted (not only the interrupted component, such as **Sit** in this scenario), even if at the interruption time the behaviour were almost completed. An alternative would be to abort the proactive behaviour. Decomposing basic behaviours into rational components (tasks) supports more realistic restart points than restarting or aborting the entire basic behaviour. For example, an **Attend to object** task includes three actions: walking to the object, facing the object, and performing an animation. In this example, if a latent behaviour (e.g., **Warn**, which moves the NPC near an intruder) interrupts the execution of the **Attend to object** task before the last action (perform an animation) is completed, the whole **Attend to object** task will be restarted when the interrupted behaviour is resumed. If the NPC continued the execution of the task from the interrupted action (perform an animation), then if the NPC were moved from the target object as a result of the **Warn** latent behaviour, the NPC would perform an animation near a different object and without facing the object. The three actions composing the task ensure that, if the task is interrupted at any time, the NPC performs a rational behaviour composed of valid actions at all times. As a consequence, the NPC approaches and faces the right object before it performs the animation.

3.5 Collaborative Behaviours

Collaborative behaviours are hard to express and implement due to their complexity. We developed a simple collaborative system that authors can use to specify abstract behaviours that can constitute initiating and responding components in a collaborative behaviour.

NPCs collaborate on a particular *topic* that is represented by a string. An NPC that collaborates on a topic is automatically registered for that topic, meaning that the NPC can collaborate with other NPCs that are interested in the same topic. To collaborate on a particular topic, one NPC must be the initiating actor (initiator) and the other NPC must be the reacting partner (reactor). In Figure 3.16, the initiator has a **Converse-Talk** proactive collaborative behaviour and the reactor has a **Converse-Listen** reactive behaviour on the same topic.

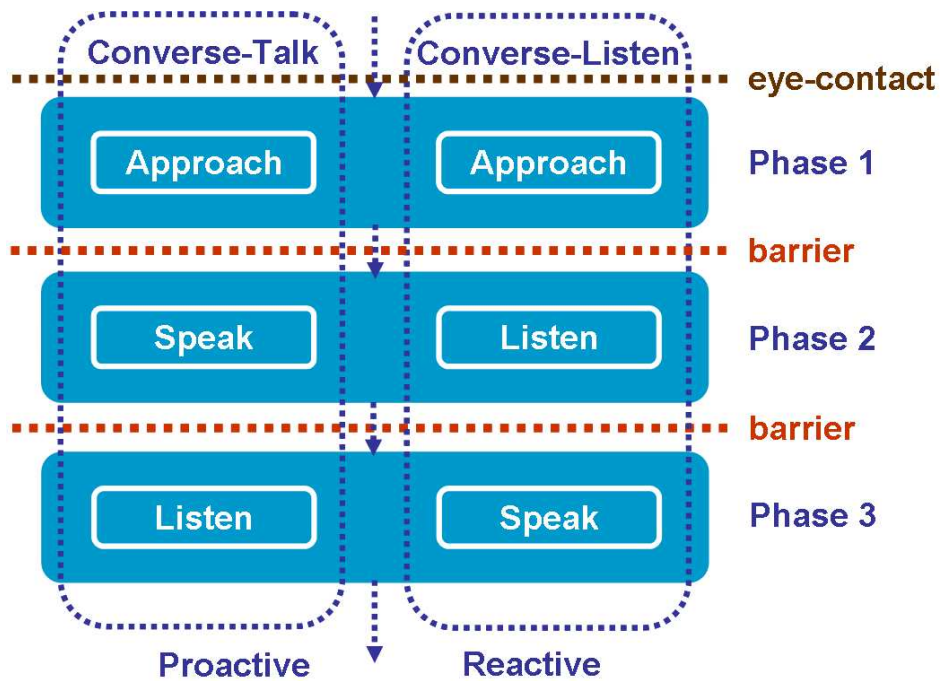


Figure 3.16: Each pair of tasks in a protocol completes successfully before a new pair of tasks can be executed.

For example, a collaborative behaviour between a guard and a friendly nearby NPC can be established if the guard has a proactive behaviour (e.g., **Converse-Talk**) on a topic (e.g., “weather”) and the collaborator (friendly NPC) has a reactive behaviour (e.g., **Converse-Listen**) on the same topic. Once the collaboration begins, both NPCs start executing pairs of tasks that comprise their respective behaviours simultaneously. For example, in this case, **Approach** is the common first task of both the **Converse-Talk** and the **Converse-Listen** behaviours. These tasks are performed concurrently. Once both tasks are completed, the collaboration en-

ters the next phase. The second task of the **Converse-Talk** behaviour is **Speak**, while the second task of the **Converse-Listen** behaviour is **Listen**, which means that the first collaborator talks while the second collaborator listens. If an author uses a collaborative behaviour, scripting code that synchronizes these activities is generated automatically. For example, if the first collaborator completes the **Speak** task before the second collaborator completes the **Listen** task, the first NPC will wait (*barrier*, as shown in Figure 3.16) for the second NPC to finish before both NPCs proceed to the next task pair.

For maximum flexibility, as long as an NPC has a proactive or latent collaborative behaviour on a topic, any other NPC that has a reactive behaviour on the same topic can engage in a collaborative behaviour with that NPC. The author can attach the initiator side of a topic and the reactor side of a topic to NPCs independently. For example, the initiator (i.e., **Converse-Talk**) of a conversation can be attached to the guard NPC and the reactor (i.e., **Converse-Listen**) can be attached to any guard-friendly NPC. In this case, the guard can approach any of the guard-friendly NPCs and start a conversation. Similarly, the initiator of an **Order-fetch** behaviour in a tavern can be attached to patron NPCs and the reactor can be attached to tavern servers and the owner. Note that the same NPC can be both an initiator and a reactor on the same topic. For example, a guard can proactively initiate a conversation with a fellow guard and reactively respond to other guards. The NPCs will be able to collaborate with each other at run-time without knowing their collaborators at compile-time. Any basic behaviour can be used to initiate a collaborative behaviour or to react to a collaborative behaviour. For any two basic behaviours to work together, one as an initiator and the other as a reactor, the initiator is used as a proactive or a latent behaviour, the reactor is used as a reactive behaviour, and the same topic name is used in both behaviour cues.

The initiator does not have to know the reactor until run-time when an NPC that is registered as a reacting partner for this topic is selected from all registered NPCs. If an NPC's behaviours include a proactive collaborative behaviour, then at spin time, the collaborative model looks for any NPC that includes a reactive behaviour on the same topic. Similarly, if an NPC's latent behaviours include a latent collab-

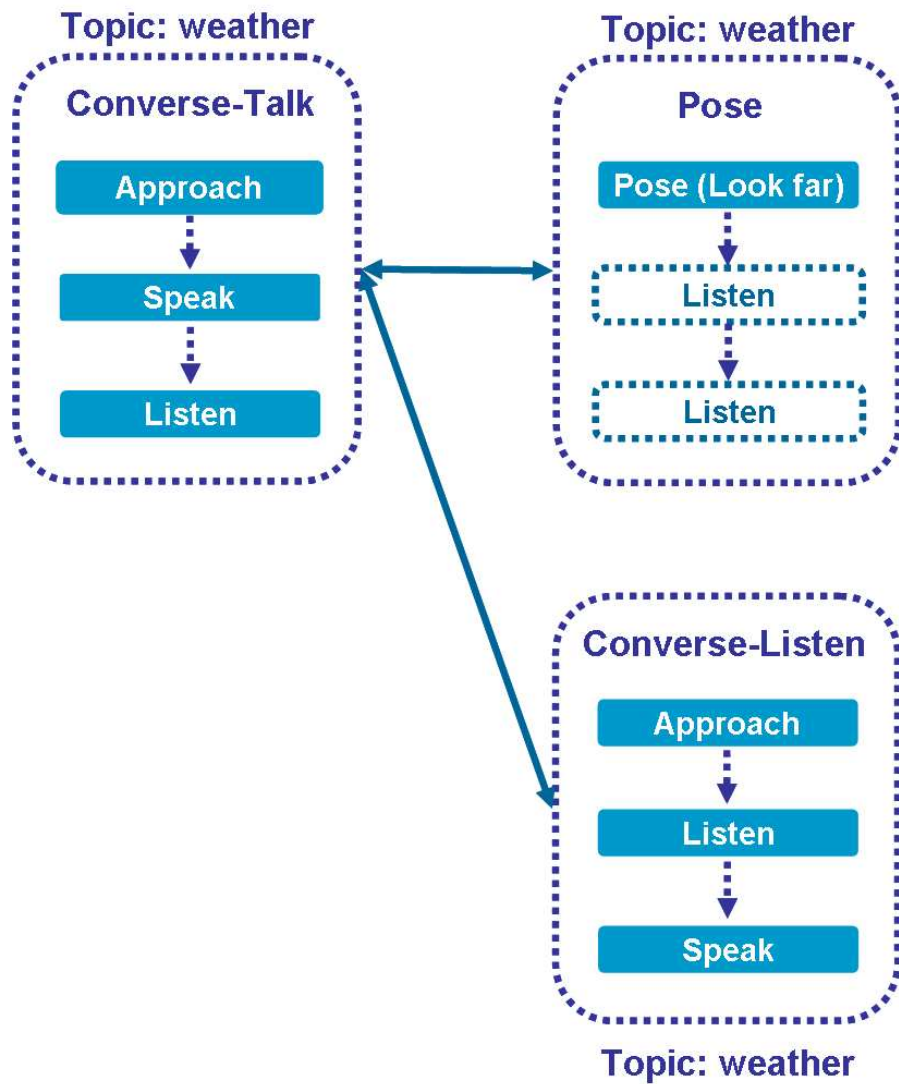


Figure 3.17: The proactive-reactive pairs **Converse-Talk** and **Pose**, as well as **Converse-Talk** and **Converse-Listen** collaborative behaviours may have different lengths. At run-time, **Listen** tasks are added to the shorter chain (**Pose**) until the chains are identical in length.

orative behaviour and the latent behaviour is triggered, a reactor NPC on the same topic is also sought. The potential reactor NPCs are filtered to include only those who satisfy two requirements. First, the reactor must make *eye-contact* with the initiator. Second, any conditions included in the initiator's collaborative behaviour must be satisfied. In this case, the proactive behaviour is considered *possible* for that spin. The *eye-contact* mechanism checks that the reactive collaborator is not currently involved in another collaborative or latent behaviour. However, eye-contact can be made if the potential reactor is involved in an independent behaviour. In that case, the independent behaviour is interrupted and it resumes after the collaborative behaviour is completed.

Since any basic behaviour can be used to construct either side of a collaboration, collaborative behaviours can have different lengths. Therefore, an initiator's behaviour and a reactor's behaviour on the same topic may contain a different number of tasks. In this case, if one of the collaborators has fewer tasks, then a **Pose** task with a default "Listen" animation (i.e., the **Listen** task) is added at run-time to the shorter chain until the length of the initiator chain is identical with the length of the reactor chain, as illustrated in Figure 3.17. This procedure has to take place at run-time, when the eye-contact for a collaborator is made, since the same chain can interact with other chains of variable lengths that are attached to other NPCs. An NPC can interact with another NPC on the same topic regardless of its actual behaviour. For example, if the initiator has two proactive behaviours **Converse-Beseech** (if the collaborator is within a certain distance of the NPC, then the NPC walks near the collaborator, it speaks, and then it listens) and **Converse-Talk** (the NPC approaches the collaborator, it speaks, and then it listens) on the "weather" topic and the reactor has two reactive behaviours on the same "weather" topic, then these two NPCs can collaborate on that topic, regardless of which particular proactive/reactive behaviour they choose. Figure 3.18 shows that the initiator can start a **Converse-Beseech** behaviour to which the reactor can respond with either a **Converse-Listen** (the NPC is approaching the collaborator, it listens, and then it speaks) or a **Converse-Beseech** behaviour on the same "weather" topic. Note that, although the **Converse-Beseech** and **Converse-Listen** behaviours constitute a more reasonable conversation match

than **Converse-Beseech** and **Converse-Beseech**, story authors often create such behaviour-mirroring situations to add more dramatism to certain scenes. In addition, for more flexibility, the initiator can become the reactor if the cues are interchanged. For example, if the **Converse-Beseech** and **Converse-Talk** behaviours are triggered by a reactive cue and the **Converse-Listen** and **Converse-Beseech** are triggered by a proactive cue, the initiator picks one of the latter behaviours and the reactor responds with one of the former behaviours, as long as they share the same topic.

As part of choosing the collaborator, the dispatcher finds the nearest or a random creature that satisfies the requirements of the collaboration. The prospective collaborator has to be available to respond to the collaborative behaviour, it has to include a response to that behaviour, and it has to satisfy any conditions imposed by the particular scenario of the collaboration. For example, in a tavern scene, a tavern server will offer a drink to a patron only under the following circumstances: the patron has a reactive behaviour whose topic is “offer”, the patron is near the tavern server, and the patron is not already involved in a collaborative or a latent behaviour.

We defined a semantics for the collaborative behaviour interruption that allows an NPC involved in a collaborative behaviour to execute independent behaviours while waiting for the collaborator to complete a latent behaviour. More generally, our semantics applies to the situation in which an NPC waits for the collaborator to complete a lengthy behaviour as part of their collaborative behaviour, even if their collaboration is not interrupted by a latent behaviour. For example, a tavern patron may decide to perform independent behaviours while a tavern server fetches a drink for this patron. This model ensures no starvation, since an NPC always executes behaviours, as discussed in Section 3.7.

The collaborators execute their actions simultaneously. An NPC can perform an independent behaviour, if one is available, or a latent behaviour while waiting for a collaborator. If one of the NPCs finishes its side of the current collaborative pair of tasks first, it waits a short amount of time (short-duration) for the collaborator to finish. If that collaborator takes too much time (medium-duration) to complete its

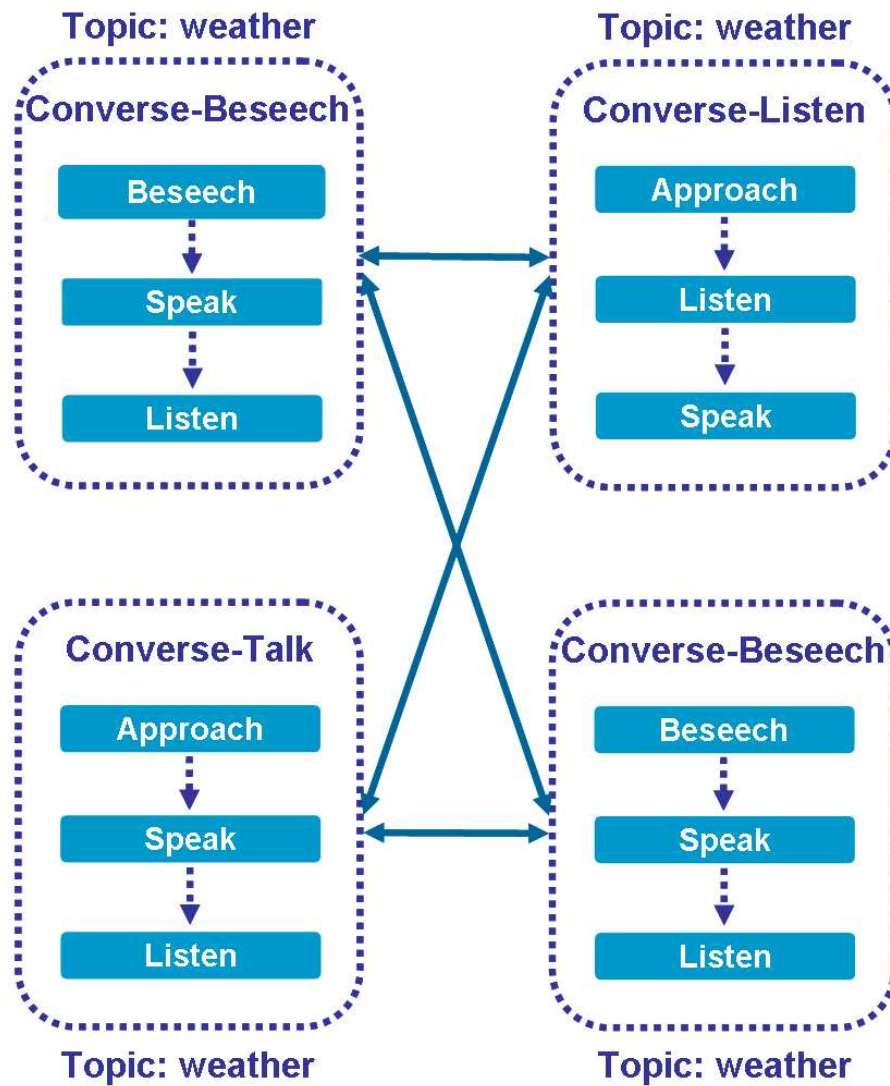


Figure 3.18: The lengths of reactive behaviours cannot be computed at compile-time due to multiple roles on each side of a collaboration and due to multiple collaborators.

part of the current pair of tasks, the NPC chooses one of its currently interrupted behaviours or it spins for a new proactive behaviour, if no interrupted behaviours exist. If the collaborator is still not finished (long-duration), the waiting NPC cancels the collaboration, since the collaborator may be blocked. This ensures no deadlock or indefinite postponement for NPCs. The concurrency control mechanism embedded into our behaviour patterns is described in detail in Section 3.7. Both NPCs re-spin, therefore they have a chance to execute the same collaboration together again.

3.6 Behaviour Dispatch and Implementation

The main obstacle in achieving better character AI, especially in commercial computer games, is the computational cost of implementing complex behaviours. The key to our behaviour model is dispatching the appropriate behaviour at any time and remembering what behaviour to return to when a behaviour is interrupted. Figure 3.19 illustrates behaviour dispatch in our model. When an NPC is created in the game, an *OnSpawn* event is generated on the NPC. ScriptEase generates code for the *OnSpawn* event that fires a custom user-defined event that we call a *behaviour event* on the NPC. Specifically, each user-defined event has a numerical parameter that can be checked in the script attached to the *OnUserDefined* event. In our case, we reserve a specific number, such as 31415, to represent our behaviour event. Computational shortcuts are needed to minimize the overhead of having behaviour scripts on NPCs. Only the NPCs that have behaviour patterns attached are registered with the behaviour event.

We must ensure that behaviour events are continuously generated to allow the NPC to quickly change behaviours in response to the game environment. NWScript supports a statement called `DelayCommand` that executes the contents of the script first, then waits a certain time and executes the specific delayed command. Our behaviour script uses a delayed command to generate a new behaviour event on the same creature, so that behaviours occur continuously. Figure 3.20 illustrates the predefined *NWN* game engine events defined for all creatures: *OnBlocked*, *OnCombatRoundEnd*, *OnConversation*, *OnDamaged*, *OnDeath*, *OnDisturbed*, *On-*

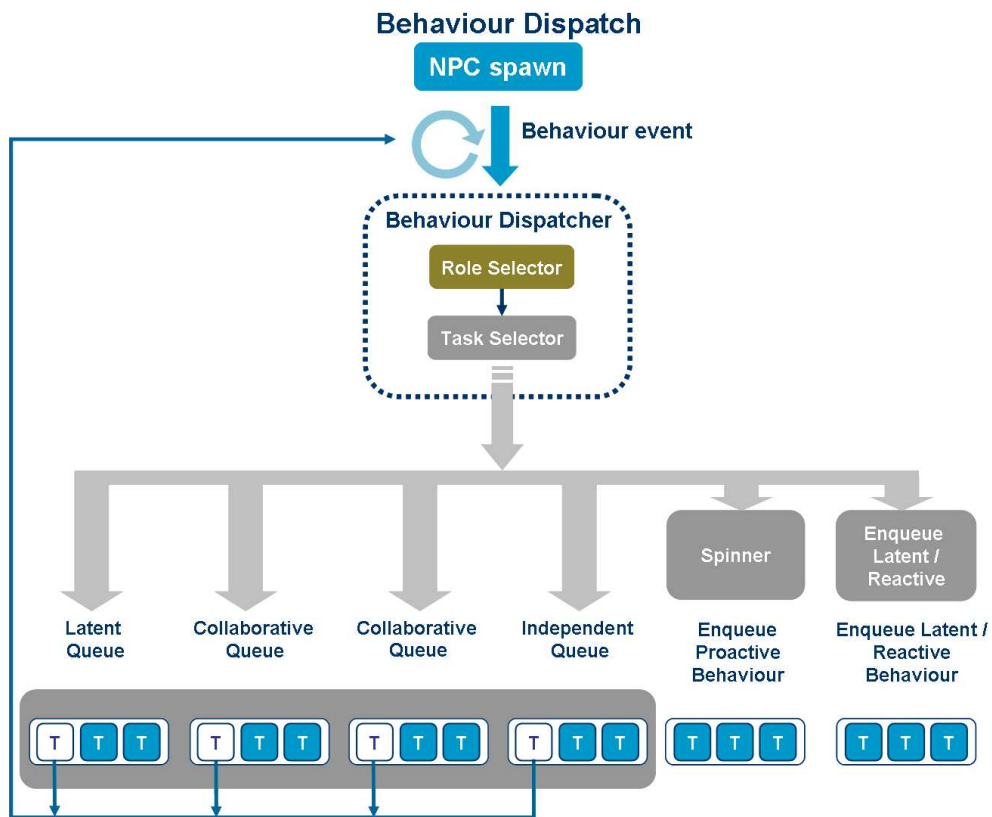


Figure 3.19: Behaviour dispatch of the proactive, reactive, and latent behaviours. If no queued task is available, the dispatcher enqueues a new behaviour.

Heartbeat, *OnPerception*, *OnPhysicalAttacked*, *OnRested*, *OnSpawn*, *OnSpellCastAt*, and *OnUserDefined*. The two events of interest for our model are *OnSpawn*, triggered when the NPC is created in the game and *OnUserDefined*, triggered when any custom event is fired on the NPC. In our case, the *OnUserDefined* event corresponds to one custom event, the behaviour event fired by the *OnSpawn* script. A creature’s event loop in *NWN* is illustrated in Figure 3.21. For example, the *OnHeartbeat* event is triggered every six seconds on the NPC and the *OnDeath* event is triggered when the NPC dies.

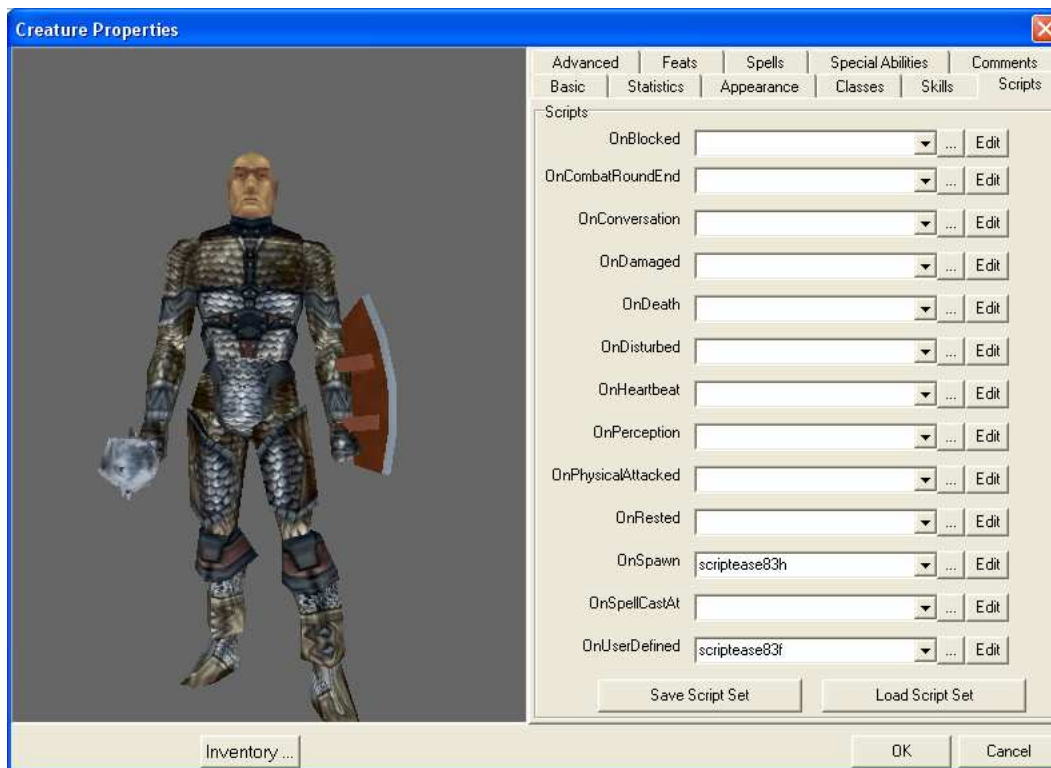


Figure 3.20: The NWScript events attached by ScriptEase to an NPC with behaviour patterns, shown as they appear in the Aurora Toolset.

In addition to firing the initial behaviour event on the NPC, the *OnSpawn* script sets a role variable (“current role name”) to be “spawn”. It also sets the NPC’s initial motivation attribute values specified by the pattern designer. Finally, it records the NPC’s original location and facing that are used when the NPC moves to its original position during a **Return** behaviour. All the actions included in the *OnSpawn* script are executed when the creature is first created in the game. If the game is saved and

reloaded, the *OnSpawn* event is not fired again, therefore a behaviour event is not generated unnecessarily.

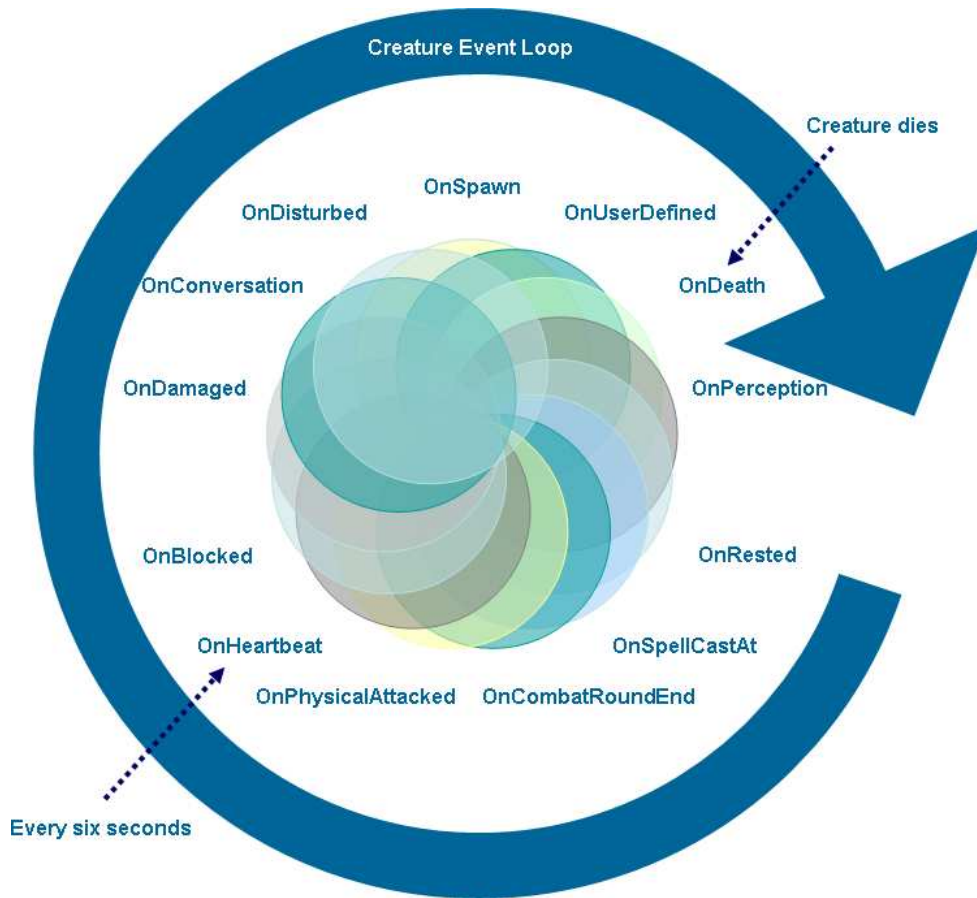


Figure 3.21: *NWN* event loop for NPCs.

The script that is triggered by the behaviour event (called *behaviour dispatcher*) contains a call to a *role selector* that chooses the appropriate NPC role based on a role variable stored in the NPC. Each role has a role cue that activates that role. When a creature is spawned in the game, it is guaranteed to have a role since the creature’s role variable is set to “spawn”. When a role is created, the default role cue is a spawn cue that simply checks to see if the role variable is set to “spawn”. If more than one role has a spawn cue for the same NPC, then the last role is activated, since it overrides the previous roles. When a cue changes the active role, it simply sets the role variable to the new role name. The next time the role selector is invoked, the new role is selected. This ensures that, whenever a role is active,

only basic behaviours within the role are selected. For example, if a guard NPC is performing a **Patron** role and an intruder approaches the guarded container, the guard will not respond with a **Warn** latent behaviour, since this latent behaviour is part of the **Guard** role that is no longer active.

Once a role is selected, it invokes a *task selector* that may select a proactive behaviour based on the role's motivation. However, the *task selector* does not always select a new proactive behaviour. At any time, an NPC can perform one of the following types of basic behaviours: a latent (independent or collaborative) behaviour, a proactive (independent or collaborative) behaviour, or a reactive behaviour. A proactive (independent or collaborative) behaviour is triggered by a spin-based behaviour cue (a probability-based or a motivation-based cue). A reactive behaviour is activated by a reactive cue that is triggered by a spin-based cue or a latent cue on the collaborator. A latent behaviour is triggered by a latent-based (non-proactive and non-reactive) behaviour cue. The selected basic behaviour triggers a set of tasks composed of the lower level actions that implement the behaviour, as illustrated in Figure 3.22.

Each unfinished behaviour is stored on a queue as shown in Figure 3.19. Unfinished behaviours of all types are possible. If an NPC is performing a collaborative behaviour and, while waiting for the collaborator to finish the current task, it decides to perform a proactive (independent or collaborative) behaviour, then an unfinished collaborative behaviour exists. Suppose that the NPC decided to perform a proactive independent behaviour. As a result, the independent queue has unfinished tasks. A latent cue may then interrupt this behaviour, so that a latent behaviour can be performed. At this point, there is an unfinished latent behaviour on the latent queue, so three queues contain tasks. If the latent behaviour currently enqueued is a collaborative behaviour and the collaborator takes a long time to perform its current task, the NPC will return to its interrupted task. This is a natural form of behaviour mixing that mimics human behaviour. However, if all enqueued behaviours are blocked for some reason, the NPC will decide to enqueue yet another behaviour. Since the only available queue is one of the two collaborative queues, the NPC may enqueue such a behaviour. There can be at most one independent (proactive), two

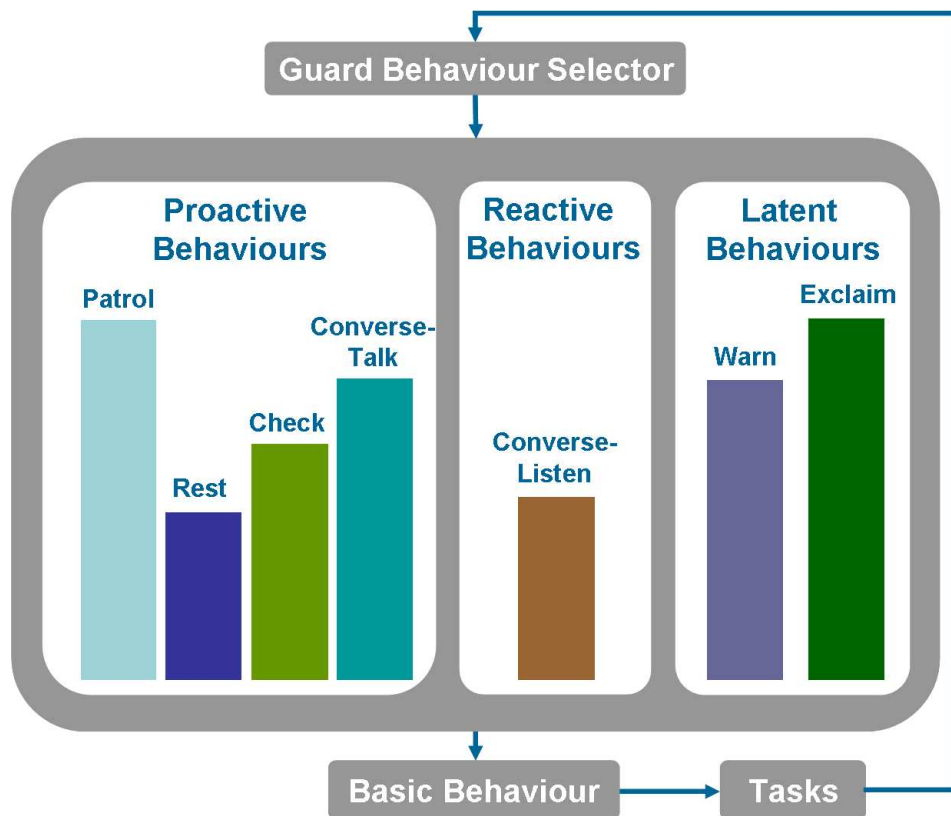


Figure 3.22: The behaviour selector for a guard NPC in our behaviour model.

collaborative (proactive or reactive), and one latent behaviour on each queue at any time. A second latent cue may trigger a latent behaviour with a higher priority than the first latent behaviour, in which case the second latent behaviour replaces the current latent behaviour.

For example, a server NPC in a tavern can offer to fetch a drink for a patron from the bar (**Offer-fetch** proactive collaborative behaviour). If the patron accepts the offer (**Receive** reactive behaviour), the tavern server walks to the bar to fetch the drink and the patron waits (in our system, for a short-duration, or less than twenty seconds) for the server to return with the drink. While the patron is waiting (in our system, for a medium-duration, or between twenty seconds and less than forty seconds), it can start executing one of the possible behaviours. For example, the dispatcher executes a proactive independent behaviour, such as walking to a random patron (**Approach**). If the patron is interrupted during the walk by a latent behaviour, such as the PC approaching the patron, then the patron executes one of the appropriate latent behaviours, e.g., a **Converse-Talk** latent collaborative behaviour with a nearby patron, to reveal a clue for the PC during this conversation. If the PC walks even closer to the patron, then the patron clears the previous latent behaviour and it executes an **Exclaim** latent independent behaviour to greet the PC, only if the priority of the **Exclaim** behaviour is higher than the priority of the **Converse-Talk** latent behaviour. Otherwise, the patron ignores the **Exclaim** behaviour and completes the **Converse-Talk** behaviour. After the latent behaviour is completed, the patron resumes its proactive independent behaviour if the server has not returned with the drink yet. If an NPC waits for the collaborator to finish a task for a long time (in our system, for a long-duration, i.e., forty seconds or more), it cancels the collaboration by clearing both NPCs' respective collaborative queues to prevent deadlock, as described later in this chapter. If the collaborator (tavern server) does return within forty seconds, the patron executes the next task of the reactive **Offer-fetch** behaviour (e.g., **Receive** that simulates receiving the drink).

Our behaviour multi-queue architecture supports four queues, one for latent (independent or collaborative) behaviours, two for collaborative (proactive or reactive) behaviours, and one for proactive independent behaviours, as illustrated in Figure

3.19. Each queue holds one basic behaviour's set of tasks that have not yet been completed. Two collaborative queues provide increased flexibility and realism for behaviours.

When it is called, the behaviour dispatcher tries to execute a task from one of the queues in the following priority order: latent, collaborative, and independent. If all the queues are empty or blocked waiting, the behaviour dispatcher selects an appropriate behaviour, adds its tasks in the appropriate empty queue, and restarts (i.e., it calls the behaviour event). For example, the tavern server's behaviour dispatcher may select an **Offer-fetch** basic behaviour based on motivations. Since this is a proactive collaborative behaviour and the tavern server's collaborative queues are empty, the dispatcher adds the tasks that compose this basic behaviour (e.g., **Approach, Speak, Listen, Fetch, Exchange, Listen, Speak**) into the server's collaborative queue and restarts, as illustrated in Figure 3.23. The **Offer-fetch** and **Receive** collaborative behaviours of the server-patron scenario are illustrated in Figure 3.23 where the server and patron have just finished their **Approach** tasks and are about to **Speak/Listen**. A detail of the dispatch mechanism for one of these types of queues is illustrated in Figure 3.24.

If one of the queues is not empty and a new behaviour of that type is generated, the dispatcher waits until the current basic behaviour is executed (i.e., until that queue is empty again) before it can enqueue another behaviour of that type. This happens unless the new behaviour has a higher priority (e.g., a latent behaviour) than the existing behaviour, in which case the queue is cleared and the new behaviour is enqueued. At the next spin, the dispatcher will start executing this behaviour.

In general, the behaviour dispatcher tries to select and perform a task from one of the queues in this priority order: latent, collaborative, independent. After the task has been successfully completed, it is dequeued from its queue and the behaviour dispatch is restarted by creating a new behaviour event, as illustrated in Figure 3.19. Dequeuing tasks after their successful execution, rather than before their completion, ensures that the behaviours are flexible and robust. If an interruption occurs, the interrupted task is eventually re-executed since it remains at the front of its

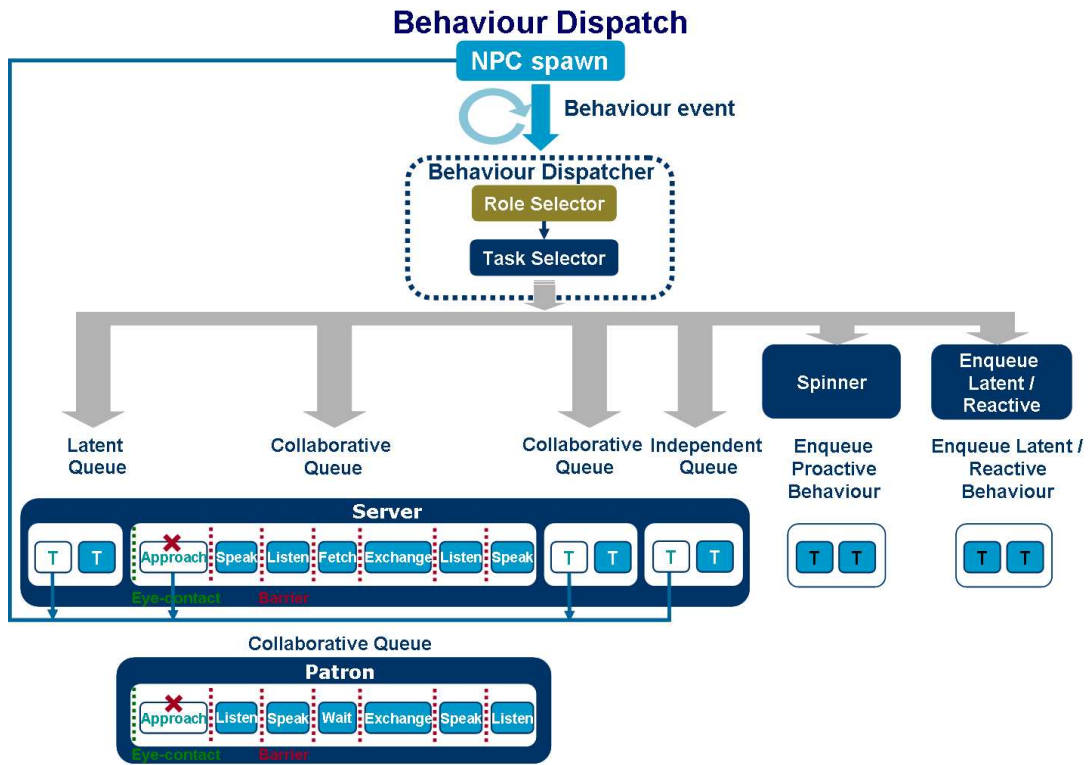


Figure 3.23: A tavern server and a patron performing a collaborative behaviour: the server initiates a proactive **Offer-fetch** behaviour and the patron responds with a reactive **Receive** behaviour.

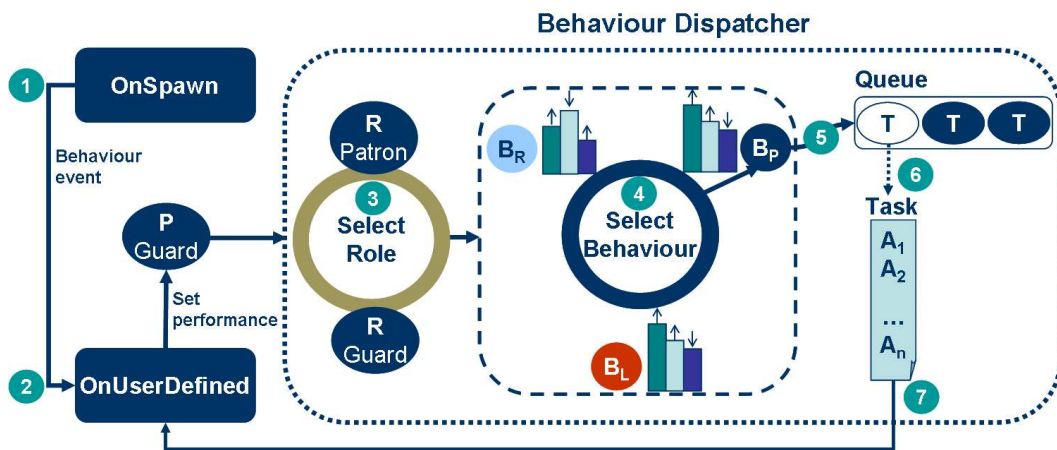


Figure 3.24: Behaviour dispatch detail for an NPC with two roles, **Guard** and **Patron**.

queue. This ensures that if the NPC is interrupted in the middle of a behaviour, it will resume the execution of that behaviour by re-executing the interrupted task. When trying to execute a task from the collaborative queue, the dispatcher first checks if the collaborator has signalled that the next task can be started. If not, the dispatcher waits for a short time and checks again. If it still has not been signalled, it proceeds to the next queue. Only if no task can be selected does the spinner select a new proactive behaviour. In addition, when a task from the collaborative queue is finished and it is dequeued, the dispatcher also signals the collaborator that it can proceed to the next task of the collaborative behaviour. When a role cue changes roles, it sets the current role variable on the NPC. The existing queues may still contain tasks from basic behaviours in the previous role, but new tasks are only added from basic behaviours in the new role.

Our system includes a memory of where the NPC was located when an interruption occurred, so that the NPC can return to that position before it proceeds. For example, consider a basic behaviour that is composed of two tasks, **Approach** a target object and **Pose** facing that object. Suppose that the **Pose** task were interrupted after facing the object, but before performing the animation. If the interruption caused the NPC to move away from the target object, then when the interrupted task is restarted, the NPC performs the animation near the wrong object. Since our system remembers the location of the NPC at the time of the interruption, the NPC walks near that location and then starts executing the **Pose** task. This can be accomplished by adding a hidden **Walk** task on the head of the interrupted queue. This feature has been designed, but not yet implemented.

More generally, we consider a proactive independent behaviour to have priority zero, a proactive collaborative behaviour to have priority one, a latent behaviour (independent or collaborative) to have priority two or larger, and a reactive behaviour to have the priority of the behaviour of the initiating collaborator. The behaviour dispatcher selects behaviours in the descending order of priorities. If a new latent behaviour meets the selection criteria and its priority is higher than the priority of the enqueued latent behaviour, then the latent queue is cleared and the new latent behaviour is enqueued; otherwise, the new latent behaviour is ignored. A latent

behaviour does not clear itself from the latent queue. For example, if a **Warn** behaviour is enqueued on the latent queue and the cue conditions are met again, the new **Warn** latent behaviour is ignored until the current behaviour is completed. Since proactive behaviours of the same type (independent or collaborative) have the same priority, they do not preempt each other. The dispatcher spins for a new proactive behaviour only if the latent queue and both collaborative queues are empty or blocked waiting for a collaborator, and the proactive independent queue is empty.

In summary, at any time, the behaviour dispatcher executes a pending task from one of the queues. If no task is available, the dispatcher selects a new basic behaviour, enqueues the component tasks in an appropriate queue, and restarts, as illustrated in Figure 3.25.

```
1.  if (latent queue not empty)
2.      execute  $T$ , task on the top of the queue;
3.      if ( $T$  completed) dequeue  $T$ ;
4.      return;
5.  if (first collaborative queue not empty)
6.      execute  $T$ , task on the top of the queue;
7.      if ( $T$  completed) dequeue  $T$ ;
8.      return;
9.  if (second collaborative queue not empty)
10.     execute  $T$ , task on the top of the queue;
11.     if ( $T$  completed) dequeue  $T$ ;
12.     return;
13. if (proactive independent queue not empty)
14.     execute  $T$ , task on the top of the queue;
15.     if ( $T$  completed) dequeue  $T$ ;
16.     return;
17. if (can enqueue selected  $B_b$ )
18.     enqueue  $B_b$ 's tasks on the respective queue;
19.     return;
20. select  $B_b$  proactive;
21. return;
```

Figure 3.25: Behaviour dispatch algorithm.

3.6.1 Latent Queue

Latent behaviours may have different priorities, depending on the author's story concept. For example, to create a more responsive guard, the **Exclaim** latent behaviour should preempt the **Warn** latent behaviour. Thus, when an intruder opens the chest, the **Exclaim** behaviour should interrupt any proactive behaviours and cancel the **Warn** latent behaviour enqueued in the latent queue. If the new latent behaviour (**Exclaim**) had the same priority as the enqueued latent behaviour (**Warn**), this scenario would fail. The intruder would execute the latent behaviour in its latent queue (**Warn**) to completion and would ignore the other latent behaviour (**Exclaim**).

On the other hand, if we always allow a new latent behaviour to clear the latent queue, a sequence **Exclaim** followed by **Warn** will only warn the intruder, since a **Warn** latent behaviour will clear a previous **Exclaim** latent behaviour before the exclamation is completed. To solve this problem, we attribute priorities to latent behaviours. The pattern designer assigns a lower priority to the **Warn** behaviour than to the **Exclaim** behaviour to obtain the desired semantics.

If a new latent behaviour has an equal or lower priority than the latent behaviour on the latent queue, then the new latent behaviour is ignored. After the completion of the enqueued latent behaviour, if a new latent behaviour is triggered, it is enqueued in the latent queue regardless of priority. If a new latent behaviour has a higher priority than the behaviour in the latent queue, then if the enqueued behaviour is independent, the latent queue is cleared. If the enqueued behaviour is collaborative, the latent queues of both this NPC and the collaborator are cleared. Then, the new latent behaviour is added to the latent queue. The latent behaviours for a single NPC are totally ordered.

We support one latent queue that can hold a latent independent or a latent collaborative behaviour, due to the highly responsive nature of latent behaviours. One queue suffices for independent, as well as collaborative latent behaviours, since latent behaviours should happen immediately. If many latent queues were supported, conditions that were verified when the latent behaviour was enqueued may not be satisfied when the latent behaviour is finally executed. For example, assume multi-

ple latent queues existed. Assume a high priority latent behaviour, **Take**, removes an item from a chest and a lower priority latent behaviour, **Give**, places the same item into an NPC's inventory. If the conditions for both **Take** (the item is available) and **Give** (the giver has the item) are satisfied, these behaviours are triggered and the behaviour dispatcher enqueues them on two separate latent queues. The behaviour dispatcher starts executing the **Take** behaviour, based on priorities. Then, when the dispatcher starts executing the **Give** behaviour, since the conditions (e.g., the giver has the item) are not checked again, the behaviour dispatcher still executes all the tasks that compose the **Give** behaviour, although the giver does not possess the item. Therefore, an NPC should execute at most one latent behaviour at any time, immediately after the behaviour cue conditions are met. This approach can be used to simulate a situation where the author wants a condition to be continuously tested for a behaviour to continue. In our case, instead of continuously checking the satisfiability of a condition, we can add a second latent cue that checks the non-satisfiability of the condition and triggers a latent behaviour with a higher priority that cancels the current latent behaviour. Alternately, we could push the conditions from the behaviour level to the task level (in our model, tasks can have definitions, conditions, and actions) and execute tasks from the latent queue until a task with unsatisfied conditions is met. In this case, we could dequeue all the remaining tasks of the latent queue.

We do not need more than one latent queue, since if a latent behaviour is already on the queue, it can never be interrupted by another latent behaviour: it can only be cleared or allowed to execute to completion. We enqueue reactive behaviours that are triggered by a latent behaviour on the latent queue, but an alternative model that enqueues these behaviours on one of the collaborative queues can be considered as well.

3.6.2 Collaborative Queue

When the behaviour dispatcher tries to select a task (T) from the collaborative queue, the behaviour dispatcher first checks if the NPC's collaborator has signaled that the next task can be started. Recall that an NPC cannot perform the next task in

the collaborative behaviour chain until the collaborator signals that it has finished its side of the current task pair.

When a task from the collaborative queue is finished, the behaviour dispatcher must signal the collaborator that it can proceed to the next task in its collaborative behaviour. When a collaborator has received signals (from itself and its collaborator), it can dequeue the top element from its collaborative queue. After an NPC executes a task to completion, if the collaborator has not completed its corresponding task, i.e., no signal has been received, the NPC first waits for twenty seconds (short-duration). Then, if the collaborator still has not signaled (medium-duration), it blocks the queue and it re-dispatches. This process happens until the collaborator sends a signal to proceed to the next task pair that unblocks the queue. If the collaborative and proactive independent queues are empty, the behaviour dispatcher spins for a new proactive behaviour and adds its tasks to the respective queue. If no signal is sent by the collaborator after a certain time (in our system, for more than forty seconds, or a long-duration timeout), the collaboration is aborted and these collaborative queues for both NPCs are cleared. This is necessary because the collaborator may be unable to return to the collaboration. For example, the collaboration may be destroyed by the PC or one of the collaborators' path may be obstructed by a game object.

When a collaborative behaviour is considered in the spin process, the following actions are performed in a loop by the behaviour dispatcher until a collaborator is found:

- Pick a random NPC in the area.
- Check if the selected NPC can collaborate (i.e., it includes a reactive behaviour on the same topic).
- Check if the behaviour conditions for the collaboration are met.
- Attempt to establish *eye-contact* with the collaborator (who is not performing a latent or a proactive collaborative behaviour).

Within a selected role, the dispatcher performs the first appropriate step for the initiator NPC from the following list:

- If there is a pending latent task in the latent queue, execute it.
- If the first available collaborative queue is not empty and the signal to proceed is set, then execute the first task in this queue. After the task is successfully executed, set the signal on the collaborator and yourself. Then each collaborator dequeues its task. If this is the last pair of tasks for this collaborative behaviour, set both collaborators' eye-contacts to available.
- If the proactive independent queue is not empty, execute the first task in this queue.
- Spin to select a new basic behaviour and enqueue it in its respective queue, depending on the nature of the basic behaviour. If the selected behaviour is a collaborative behaviour, set both collaborators' eye-contacts to busy.

An NPC can perform at most two collaborative behaviours at the same time, with at most two NPCs.

3.6.3 Independent Queue

The proactive independent queue has the lowest priority, hence it can be interrupted by both latent and collaborative behaviours. However, when a collaborative behaviour executes and the collaborator takes more time to complete its current task, an NPC may start executing independent behaviours, if they are available, or it can enqueue new independent behaviours and execute them at a later time.

3.6.4 Behaviour Dispatch Summary

On each NPC, the registering behaviour event triggers a behaviour dispatcher that selects a single basic behaviour as a result of a probabilistic or motivational choice among all the proactive behaviours that the NPC could initiate. For example, the behaviour dispatcher “spins” to select a collaborative proactive behaviour (e.g., **Offer-fetch**) from a server NPC's potential proactive behaviours and then it generates a

new behaviour event. In response to the new behaviour event, the dispatcher adds this behaviour's component tasks to the appropriate queue (one of the collaborative queues in this example) and then it generates another behaviour event. For this event, the dispatcher starts executing the task on the top of the appropriate queue and it generates another behaviour event. Each task removes itself from the queue if it is successfully completed. Eventually, all the tasks on the collaborative queue will be completed, if the collaboration is not cancelled. This process continues until the NPC role is changed and then it restarts for the new role.

Proactive Behaviour Dispatch

Suppose that the dispatcher spins and selects a proactive independent behaviour. Usually, the behaviour dispatcher does not spin for a new proactive behaviour until the current behaviour (proactive, reactive, or latent) is executed to completion. There is an exception to this situation if the NPC is blocked. An NPC is *blocked* if at least one of its queues is not empty and there is no change in any queue for a long time (e.g., ≥ 40 seconds) or if a collaborative queue is not empty and has not changed for a medium time (e.g., ≥ 20 seconds and < 40 seconds). The first case usually occurs when a behaviour is being executed and the NPC cannot finish a task. For example, during an **Approach** behaviour, the PC blocks the NPC's path for a long time. The second case usually occurs when an NPC's collaborator takes too long to complete its current task of the collaboration. For example, if a server who is fetching supplies for an owner does not return quickly, the owner is blocked.

A blocked behaviour causes the dispatcher to spin for a new proactive behaviour (independent or collaborative, depending on the NPC's motivations). For example, the initiator (owner) sends the responder (server) to the storeroom to fetch supplies. While waiting, the owner may start a proactive independent behaviour (**Exclaim** to greet the nearest patron) or a proactive collaborative behaviour (**Offer-drink** to the nearest patron), based on motivations. Note that since we support two collaborative queues, we can have two collaborative behaviours in progress at the same time if one of them is blocked.

In summary, when an NPC is created in the game, the behaviour dispatcher

spins for a proactive behaviour for that NPC. If the proactive behaviour selected is an *independent* behaviour and the proactive independent queue is empty, the dispatcher enqueues this behaviour on the proactive independent queue. If the proactive behaviour selected is a *collaborative* behaviour, the dispatcher enqueues this behaviour on the first available collaborative queue, if the conditions of the eye-contact with the collaborator are met. If one of the tasks of the collaborative behaviour takes too much time to complete, the dispatcher first selects tasks from one of the collaborative or proactive independent queues. If these queues are empty, the dispatcher spins for another proactive behaviour. If this is another proactive collaborative behaviour, the dispatcher enqueues it on the second collaborative queue. If this is a proactive independent behaviour, then the new proactive behaviour is enqueued on the proactive independent queue.

When executing collaborative behaviours, the dispatcher always selects tasks, in order, from the available collaborative queues. A queue is *available* if it is not empty and not blocked. If there are no more tasks on the first collaborative queue (cq1) before the second collaborative queue (cq2) becomes empty, then the dispatcher continues selecting tasks from cq2 until cq2 is blocked (e.g., the NPC is waiting for the collaborator). Only then will the dispatcher fill cq1 and start executing from cq1. Once cq2 is available, the control switches to cq2 until it is completed. The dispatcher selects tasks from the first collaborative queue if it is available. Then, the dispatcher selects tasks from the second collaborative queue, if it is available. Finally, the dispatcher selects tasks from the proactive independent queue, if the queue is available. While cq2 is blocked, if cq1 also becomes blocked, the dispatcher selects tasks from the proactive independent queue. If this queue is empty, the dispatcher spins for a proactive behaviour, but it ignores a collaborative behaviour, since both collaborative queues are not empty. Once a proactive independent behaviour is selected, it is added to the proactive independent queue. The dispatcher selects tasks from this queue until one of the collaborative behaviours can continue.

When the dispatcher is looking for non-empty queues, it may encounter tasks that have been inserted for *latent* or *reactive* behaviours.

Latent Behaviour Dispatch

A *latent* behaviour may be added to the latent queue when a behaviour cue triggers it. If the latent queue is empty, the new latent basic behaviour is added by the dispatcher on the NPC's latent queue. If the latent queue is not empty, the new latent behaviour should be executed if its priority is higher. In this case, the latent queue is cleared and the new latent basic behaviour fills in the latent queue. The semantics of the latent (independent or collaborative) behaviour execution is the following:

- If the latent queue is empty, then the dispatcher enqueues the latent queue with the first latent behaviour whose cue is activated.
- If the latent queue contains a latent behaviour with a lower priority than the new latent behaviour, then the dispatcher clears the latent queue and enqueues the new latent behaviour. In this case, if the latent queue contains a latent collaborative behaviour, the reactor's latent queue is cleared as well.
- If the latent queue contains a behaviour with priority equal or higher than the new latent behaviour, then the dispatcher ignores the new latent behaviour.
- If the latent queue contains a reactive behaviour that was triggered by a collaborator's latent collaborative behaviour, then the dispatcher always enqueues the new latent behaviour. There is a lack of total ordering among the priority schemes of different NPCs and this policy ensures that the NPCs are more responsive to their own latent behaviours.

For example, when the PC approaches a patron, the patron walks to the nearest NPC and starts a conversation (latent collaborative behaviour **Overhear-Talk**). If the PC walks very close to this NPC, the patron may respond with a higher priority latent behaviour, **Exclaim**. As a result, the patron cancels its latent collaborative behaviour by clearing its own latent queue, as well as the collaborator's latent queue, and it enqueues the new latent independent behaviour, **Exclaim**. After the patron greets the PC (latent independent behaviour **Exclaim**), if the cue conditions for the

Overhear-Talk latent collaborative behaviour are satisfied, then the patron walks again to the closest NPC and starts a conversation. In the guard NPC example, while the guard is resting, if an intruder (PC or NPC) is noticed near the guarded chest, the guard interrupts its rest, warns the intruder, and then resumes its rest, as illustrated in Figure 3.26.

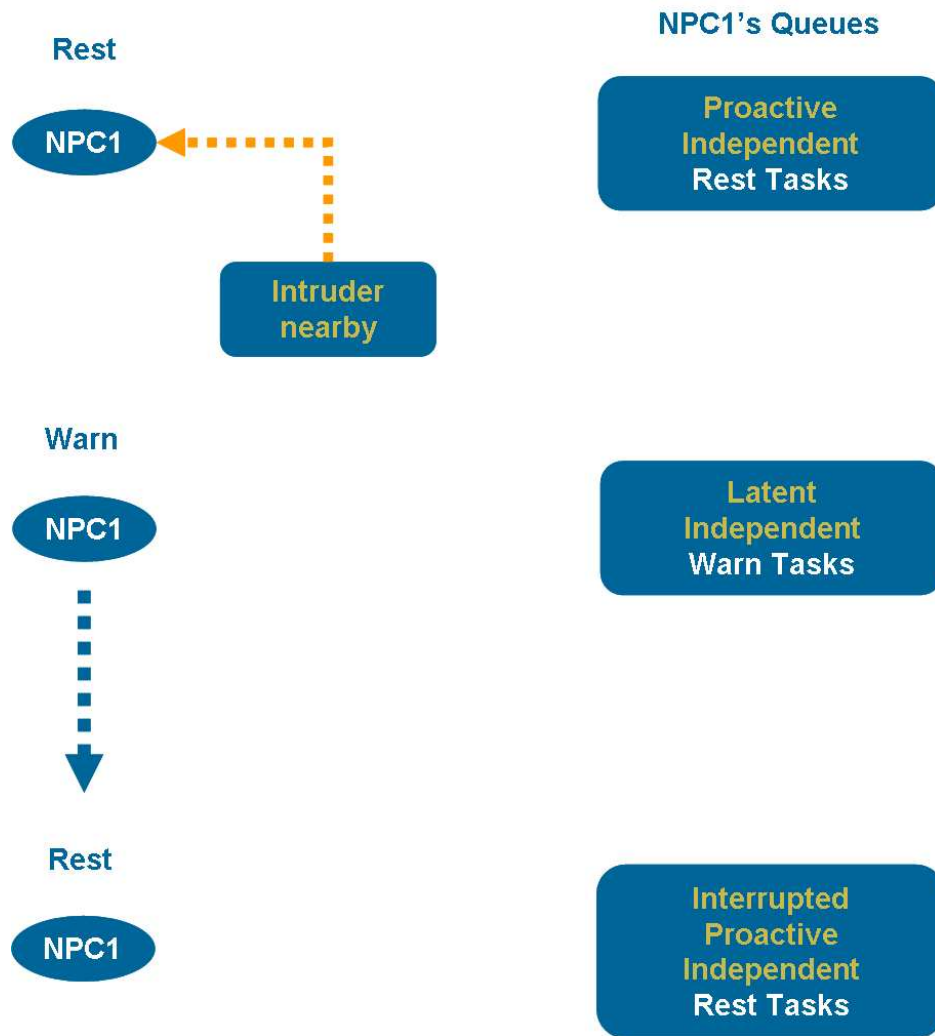


Figure 3.26: The **Warn** latent behaviour interrupts the **Rest** proactive independent behaviour of a guard NPC when an intruder is near the guarded chest.

Reactive Behaviour Dispatch

A *reactive* behaviour is added to one of the collaborative queues of an NPC by the initiator of the collaboration, if eye-contact is made. For example, if a tavern server

initiates an **Offer-fetch** collaborative behaviour and the patron is available for collaboration, then the tavern server adds a proactive collaborative basic behaviour's tasks to its first empty collaborative queue and, at the same time, it adds a reactive basic behaviour's tasks to the responder's first empty collaborative queue. The tav-

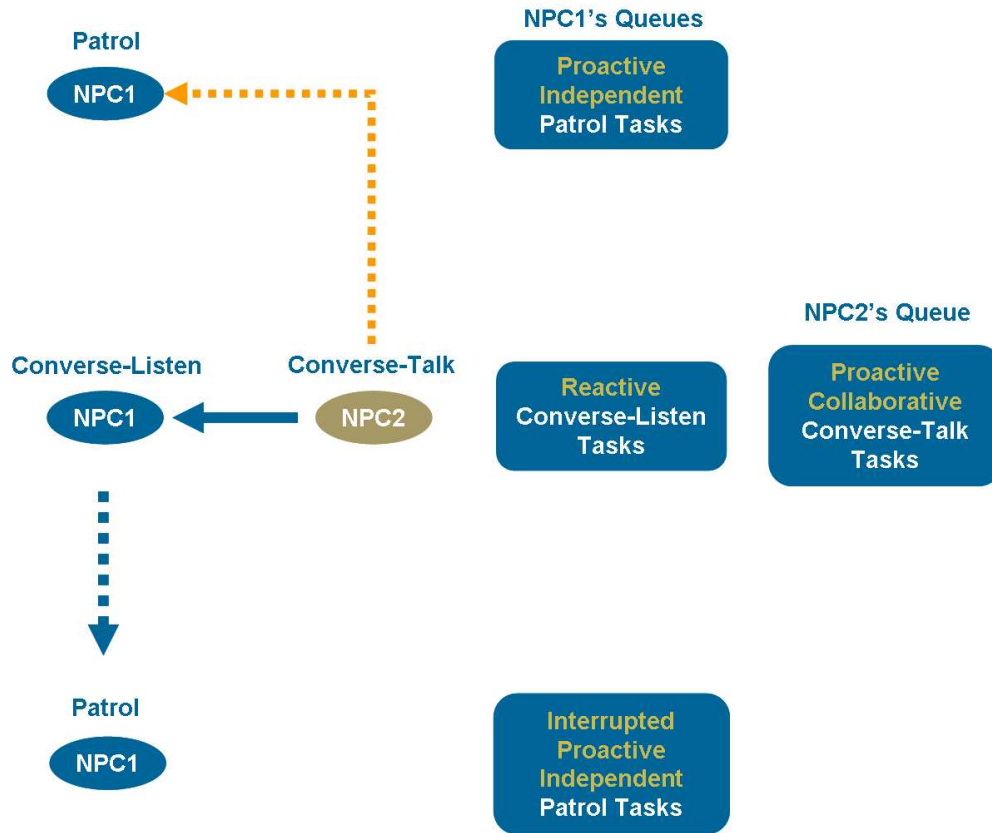


Figure 3.27: The **Converse-Listen** reactive behaviour interrupts the **Patrol** proactive independent behaviour, since a proactive independent behaviour has a lower priority than a reactive behaviour.

ern server's proactive collaborative behaviour **Offer-fetch** interrupts a patron's **Approach** (the patron walks to the tavern's bar) proactive independent behaviour that is resumed after the collaboration finishes. Figure 3.27 illustrates a different scenario in which a guard-friendly NPC initiates a **Converse-Talk** proactive collaborative behaviour with a guard NPC that performs a **Patrol** proactive independent behaviour. The proactive **Converse-Talk** behaviour of the guard-friendly NPC causes the guard NPC to perform a reactive **Converse-Listen** behaviour. As a result, the guard's proactive independent behaviour **Patrol** is interrupted by the guard's reac-

tive behaviour **Converse-Listen** that has a higher priority (1) than the proactive independent behaviour (0). After completing its reactive **Converse-Listen** behaviour, the guard resumes its interrupted proactive **Patrol** behaviour. A reactive behaviour triggered by a collaborator’s latent collaborative behaviour is added to the latent queue, but another model in which it is added on one of the collaborative queues is also possible.

Behaviour Dispatch Example: “Bob-Sally”

The following more complex “Bob-Sally” example includes many different kinds of behaviours. This example illustrates the interactivity, alertness, and variety of the behaviour model.

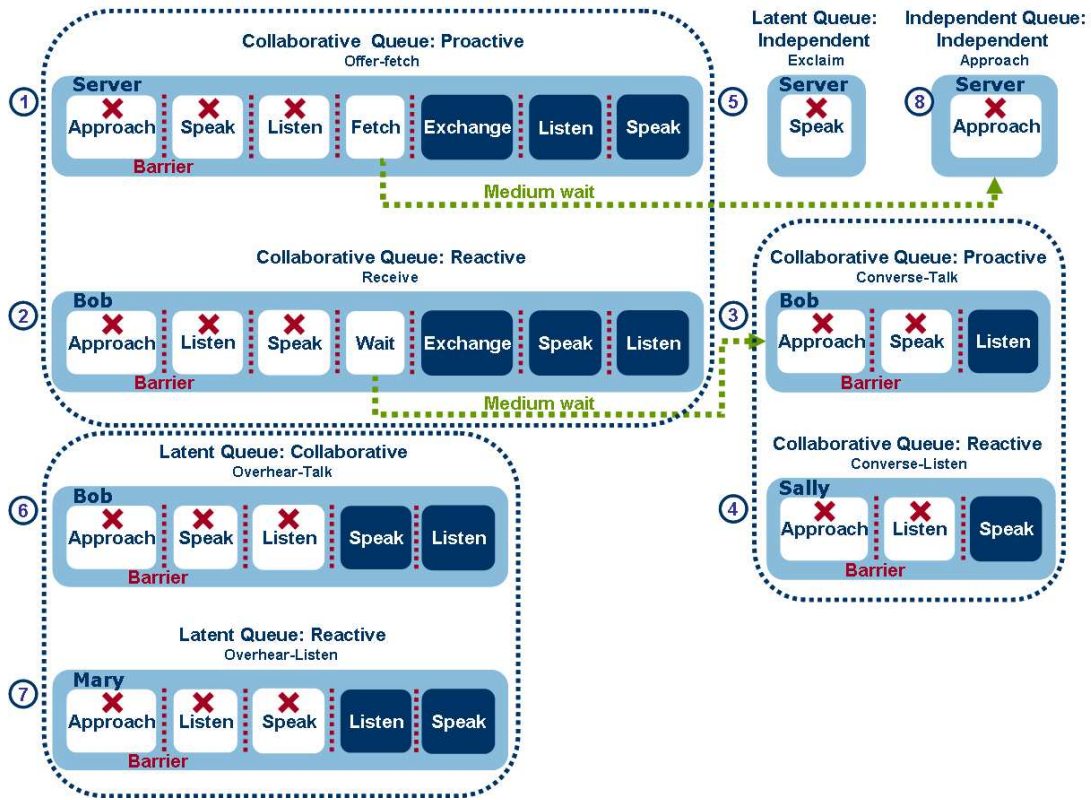


Figure 3.28: After Bob and Mary finish their latent collaborative behaviour, Bob and the server resume their proactive collaborative behaviour. Then, Bob and Sally resume their interrupted proactive collaborative behaviour.

Suppose that when a server NPC is created in a tavern, its behaviour dispatcher

spins for a proactive behaviour and chooses the **Offer-fetch** proactive collaborative behaviour with a random patron, Bob, as illustrated by the circled steps 1 and 2 of Figure 3.28. While the server walks to the bar to fetch a drink, Bob performs a simple animation and then waits. If Bob's wait exceeds twenty seconds, Bob spins for another proactive behaviour to execute while waiting for the server to return. If the selected proactive behaviour is collaborative (e.g., **Converse-Talk** with the nearest patron), then Bob walks to the nearest patron, Sally, and starts a conversation (circled steps 3 and 4). While the server is returning, a PC walks close to the server. A range cue triggers a latent independent behaviour (e.g., **Exclaim**) on the server who faces and greets the PC (circled step 5). While Bob is talking to Sally (waiting for the server to return), a latent collaborative **Overhear-Talk** behaviour is triggered by a range cue between Bob and the PC, as illustrated in Figure 3.29. As a consequence, Bob walks to the currently nearest patron, Mary, and starts a conversation with her that reveals information to the PC (circled steps 6 and 7). When the server returns to Bob, since Bob is performing a latent behaviour, the server must wait. The server spins for a proactive behaviour (**Approach**, i.e., it approaches a random patron) while waiting for Bob to finish his current task (circled step 8). The queues for this scenario at this particular point are illustrated in Figure 3.28, where the empty boxes show tasks that are completed (red cross) or in progress (no red cross) and the filled boxes show tasks that have not started yet. When Bob finishes his latent behaviour, Bob and the server finish their collaboration (the server finally gives the drink to Bob) and Bob resumes his interrupted proactive collaborative behaviour (conversing with Sally).

If a latent collaborative behaviour is blocked, the dispatcher chooses the first behaviour available to execute in the meantime. For example, if the conversation between Bob and Mary is blocked, Bob tries to resume its collaborative behaviour with the server. However, if the server has not yet returned, Bob will resume his collaboration with Sally. If there is a behaviour with the same collaborator as the latent collaborative behaviour, then this behaviour is set as unavailable, since there is no reason to resume a behaviour with a blocked collaborator.

The semantics of our model can be easily adapted at the implementation level

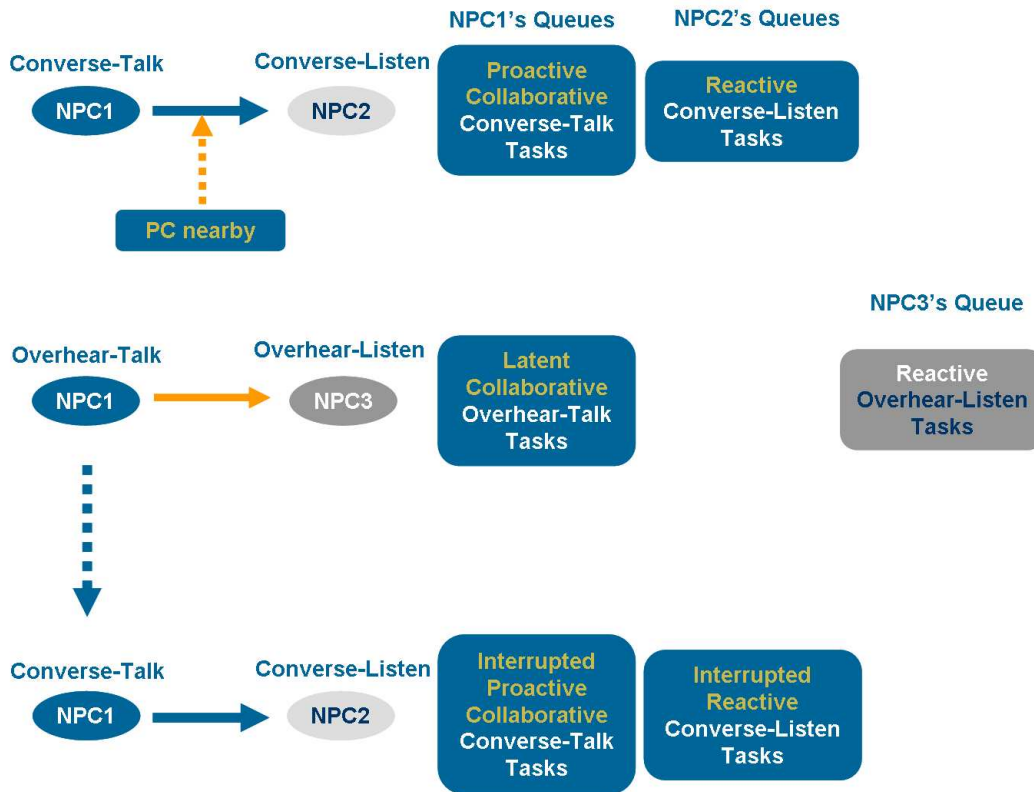


Figure 3.29: The **Overhear-Talk** latent collaborative behaviour interrupts the **Converse-Talk** proactive collaborative behaviour when the PC walks near the NPCs, since a latent behaviour has a higher priority than a proactive behaviour.

for different designs, providing additional flexibility. In some situations, it would be enough to allow an NPC to initiate at most one collaborative behaviour at any time. More specifically, an NPC may use only the first collaborative queue to store proactive collaborative behaviours, the second collaborative queue being reserved for reactive behaviours. For example, while waiting for a server to bring supplies from the storeroom in a collaborative behaviour **Ask-fetch**, the owner cannot initiate any new proactive collaborative behaviours. However, the owner can respond to a proactive collaborative behaviour initiated by another NPC. For instance, the owner can respond to a patron who orders a drink. Note that it may be possible for the server to initiate a collaborative behaviour with the owner who sent it to the storeroom, if the server is not currently executing a proactive collaborative behaviour. This situation is useful if the server decides to converse with the owner on a different topic, which provides a more realistic interaction.

3.7 Concurrency Control Model

Concurrency models have been studied extensively for general-purpose computing. Our behaviour system must satisfy the same requirements that exist for general parallel computing systems. The issues of synchronization, deadlock, and indefinite postponement must be addressed to ensure correct and robust execution of behaviours for NPCs. Each of these requirements poses a challenge in the CRPG behaviour domain. We provide a solution to each of these problems.

Our model allows NPC behaviours to be interrupted and resumed. It also supports NPC-PC interactions in addition to NPC-NPC collaborations. Our concurrency control mechanism is invisible to the story author and it is only partially visible to the designer of behaviour patterns. We present a description of the difficulties we experienced in building a concurrency model for interacting NPCs, as well as our solutions.

3.7.1 Synchronization

Synchronization among NPCs is essential so that an NPC completes all of the actions of a behaviour before the next behaviour is fired. For example, in an **Offer-fetch** proactive collaborative behaviour, the tavern server should not fetch a drink (server's task) for a patron before the patron has decided whether to order a drink or not (patron's task). In addition, synchronization among the behaviours of a single NPC is crucial. For example, a tavern server should manage the simultaneous execution of its own collaborative behaviours (proactive or reactive) seamlessly.

Synchronization of a Single NPC's Behaviours

The queue structure ensures synchronization for *a single NPC* (proactive basic behaviours). The control model based on queues with different priorities ensures synchronization among an NPC's basic behaviours by preventing an NPC from starting a behaviour of the same type before the previous behaviour is done, except for two particular cases. First, if the server is executing a collaborative behaviour, such as **Offer-fetch**, it cannot select a new collaborative behaviour, such as **Converse-Talk**, until the current collaborative behaviour is completed. However, if the initiator (e.g., the server NPC) of a collaborative behaviour (**Offer-fetch**) takes too much time to finish the current task, the reactor (e.g., the patron NPC) may spin for a new proactive behaviour (collaborative or independent) to execute in the meantime. As soon as the initiator finishes the lengthy task of the collaborative behaviour, the reactor interrupts the new collaborative behaviour (**Converse-Talk**) and both NPCs continue their original collaborative behaviour (**Offer-fetch**) until this behaviour completes or one of the NPCs takes too much time to finish a task. In this case, the collaborator selects tasks from the collaborative or proactive independent queues or, if these queues are empty or blocked, it spins for a proactive behaviour. Without synchronization, the tavern server will receive tasks from both behaviours (**Offer-fetch** and **Converse-Talk**) in an interleaved manner that may violate synchronization. Note that this process is symmetrical for both collaborators: the initiator can also spin for a proactive behaviour if the reactor takes more time to complete a task of their common collaborative behaviour.

Second, a latent behaviour (independent or collaborative) can clear any other latent behaviour with a lower priority from the latent queue and can interrupt any proactive behaviour. When a latent behaviour is interrupted, it is cancelled rather than interrupted and resumed. For example, a tavern server can perform latent behaviours at any time during the execution of a proactive or reactive behaviour, resuming the interrupted behaviours after a latent behaviour is completed. An interrupted collaborative behaviour can interrupt another independent or collaborative behaviour when the interrupted behaviour is resumed.

Synchronization Among NPCs

The queue mechanism cannot prevent synchronization problems among NPCs during collaborative behaviours. Each object in the game executes actions from its personal action queue in order. However, without proper synchronization among action queues, the actions of several NPCs may interleave in an unexpected manner. For example, while performing an **Ask-fetch** collaborative behaviour with a patron, a tavern server should not fetch a drink before the patron orders it.

We need another mechanism to solve this problem. To ensure synchronization for *multiple NPCs* (proactive and reactive behaviours) during collaborative behaviours, we introduce two mechanisms. The first mechanism is an *eye-contact* protocol ensuring that both NPCs agree to participate in a collaborative behaviour before the behaviour is started. The eye-contact mechanism works as follows. If NPC1 identifies NPC2 as a potential collaborator, then NPC1 tries to make eye-contact with NPC2. If NPC2 is involved in a latent or collaborative behaviour, NPC2 denies eye-contact. If NPC2 is not involved in a latent or collaborative behaviour, NPC2 accepts the eye-contact and sends a signal to NPC1 to start the appropriate behaviour. It is possible for an NPC to have eye-contact with more than one NPC at the same time. This can happen only when the collaborator in a collaborative behaviour takes too much time to complete a task. In this case, the waiting NPC may make eye-contact with a second NPC to start a second collaborative behaviour. We implement the eye-contact mechanism using state variables of the NPCs, so that their values are updated instantly. Note that when a game is saved

and then loaded, all the values stored on the NPC as state variables, as well as the values stored on the module as global variables, are preserved.

The second mechanism is a *barrier* between collaborative pairs. A collaborator cannot pass the barrier until both collaborators reach it. Together with the eye-contact mechanism, the barrier mechanism ensures that the tasks on the queues of different NPCs are synchronized, as illustrated in Figure 3.30 for the **Converse-Talk** and **Converse-Listen** behaviours.

The synchronization model used by our behaviour patterns is scalable to more complex NPC interactions.

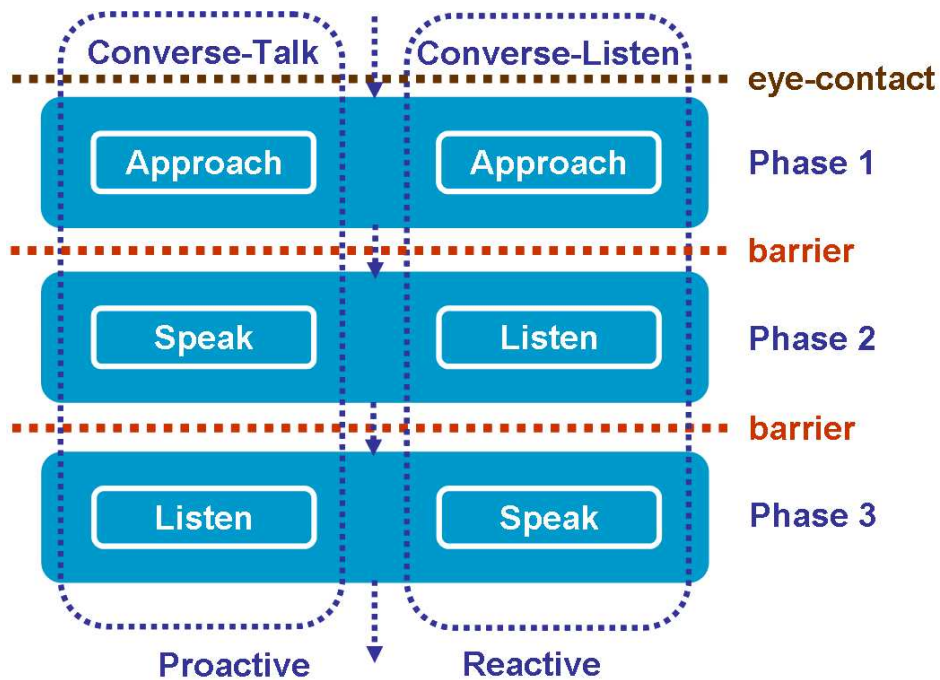


Figure 3.30: The barrier and eye-contact mechanisms that ensure behaviour synchronization.

3.7.2 Deadlock

Deadlock is a problem that occurs when two or more entities in a group need a resource to advance, but they will never acquire that resource since it is held by another entity in that group and will not be released. Entities may be granted access to resources such that a group of two or more competing entities is unable to proceed,

because each entity holds resources and is also waiting for the release of resources currently held by others in that group [14]. In our behaviour system, deadlock must be avoided so that neither NPC in a collaborative behaviour waits forever for the other NPC to perform a task as part of their collaborative protocol.

A state of *deadlock* arises when all of the following four conditions are satisfied [14]:

1. *Mutual exclusion*: the NPCs claim exclusive control of the resources they require.
2. *Hold and Wait*: the NPCs hold resources already allocated to them while waiting for additional resources.
3. *No preemption*: resources cannot be forcibly removed from the NPCs holding them until the resources are used to completion (scripts are successfully executed to completion).
4. *Circular wait*: a circular chain of NPCs exists, such that each NPC holds one or more resources that are requested by the next NPC in the chain.

For example, consider a tavern server who offers to fetch a drink for a patron. If the patron accepts the offer, then the server walks to the bar to fetch a drink. The game engine executes one script at a time. When one of the NPCs starts executing a script, no other task is executed until the completion of the current script (*mutual exclusion*). The patron waits for the server to return with the drink. If the server's entire walk to the bar were represented by a single script and if this script did not execute to completion (e.g., the server's path is blocked by a game object), then the patron would wait forever for the server to finish (*hold and wait*). If the tavern server's script must execute to completion (*no preemption*), the tavern server will be blocked forever. The patron is also blocked since it must wait for the server to serve the drink before its collaborative behaviour ends (*circular wait*).

However, this is not how actions work in *NWN*, otherwise deadlock would frequently occur. In *NWN*, to allow events in the game to interleave, the engine does not complete the walk action during the execution of the script. A walk action is

placed on the action queue (described in Section 3.4.8) and the script ends. This allows other NPCs and objects to perform actions. This means that the patron's script could execute as soon as the server starts walking (finishes the tavern server script). While the *NWN* game engine solves deadlock using action queues, our behaviour system introduces problems due to our synchronization model and has to provide appropriate solutions. To ensure synchronization, the behaviour dispatcher does not allow the patron to proceed to the next task. This *lock* mechanism could potentially cause a deadlock. In our system, when a task is executing, it has a *lock* that prevents other tasks for that NPC from executing (*mutual exclusion*). This lock is returned to the behaviour dispatcher when a task is completed so that it can be given to the next task. If a tavern server fails to complete its **Approach** task (walking to the tavern's bar), then the patron will wait forever for the tavern server to finish (*hold and wait*). The tavern server holds the lock that cannot be relinquished until the task finishes (*no preemption*). Again, the patron must wait for the tavern server to deliver the drink before their collaborative behaviour ends (*circular wait*).

We could prevent deadlock caused by our synchronization mechanism if one or more of the above conditions were violated. Our system detects deadlock, i.e., determines at run-time if deadlock has occurred and identifies the entities and resources involved. Our behaviour system recovers from deadlock by clearing the detected deadlock so that the entities can complete execution and free their resources. To detect and recover from deadlock, we use timeout mechanisms. An NPC waits for the collaborator to finish a task. Meanwhile, a medium-duration timeout can allow the NPC to perform behaviours (*hold and wait*). This prevents the deadlock but causes the other collaborator to starve. However, if the collaborator takes too much time (long-duration timeout) to finish the current task, the NPC cancels the collaboration for both collaborators (*hold and wait* exception for the canceller and *no preemption* for the collaborator). Cancelling the tasks for both collaborators ensures that neither collaborator is performing a behaviour that requires a missing synchronization. This eliminates the deadlock and frees the tavern server from starvation.

3.7.3 Indefinite Postponement

Indefinite postponement (or starvation) is a problem related to deadlock. In this situation, an entity never receives the resources that it needs. In our behaviour system, indefinite postponement can occur when an NPC must wait forever to perform a behaviour. There are three potential ways for indefinite postponement to be caused in our behaviour system.

The first cause for indefinite postponement can occur in a deadlock situation, in which both NPCs suffer indefinite postponement. The timeout mechanisms discussed in the previous section prevent indefinite postponement from occurring in such situations.

The second potential cause for indefinite postponement can arise when a collaborating NPC has its action queue cleared. In this case, the collaborator will suffer indefinite postponement. There are several situations in which an NPC's action queue can be cleared. First, the player may start a conversation with the NPC by clicking on that NPC. This scenario triggers an *OnConversation* event that in turn determines the execution of a script, if available. If the NPC has an *OnConversation* script that contains a `BeginConversation` command, then a `ClearAllActions` command is executed instantly and it clears the NPC's action queue, clearing the call to the spinner implicitly, hence the NPC's behaviours. If there is no *OnConversation* script attached to the NPC or if the *OnConversation* script does not contain a `ClearAllActions` command, then our behaviour model is not affected. Second, this problem arises when another game object (not necessarily a creature) has a script attached that assigns a `ClearAllActions` command to an NPC with behaviours. The NPC's action queue is cleared and its behaviours cancelled. We need a mechanism to protect the NPC from indefinite postponement since the NPC will never finish the current task and, consequently, it will not signal the behaviour dispatcher to start a new task.

The third situation in which indefinite postponement can occur is when an NPC's path is blocked by an obstacle. For example, while a tavern server is approaching a patron or the bar, if an obstacle obstructs the server's path, then the walk action is blocked, which leads to failure to complete the entire behaviour.

Our solution to indefinite postponement mirrors our solution to deadlock: we use a timeout mechanism. The behaviour event (fired continuously using a delayed command) increments a counter by one for every NPC. The dispatcher checks whether the counter has reached a specific threshold value (forty seconds in our system). If the counter reaches the threshold value, the NPC's behaviour event is restarted. The counter is set to zero every time the NPC performs a behaviour, so as long as the NPC is performing behaviours, no restart will occur. This timeout mechanism ensures that the NPC is not blocked and that the behaviour dispatcher always starts an NPC behaviour. If no behaviour execution is recorded within this time interval, then a behaviour event is triggered on the NPC.

3.8 Chapter Summary

In this chapter, we introduced our behaviour model, we presented behaviour patterns, their structure, use, and implementation. We illustrated how each of the major components of the model (performance, cue, role, motivation, behaviour, and task) can be represented and manipulated using new ScriptEase constructs. With these components, an author can generate proactive, reactive, and latent NPC behaviours that can be independent or collaborative. Learning can be seamlessly added to this model through behaviour cues to replace the motivational or probabilistic selection of behaviours within a role. Behaviour patterns generate complex and non-repetitive NPC scripts automatically for interactive story authors who are not programmers. Patterns are reusable and they hide the level of complexity necessary to create a realistic interactive story. We concluded with an outline of the problems and solutions posed by our concurrency control model for interacting NPCs.

Chapter 4

Reinforcement Learning in ScriptEase

In this chapter, we introduce ALeRT (Action-dependent Learning Rates with Trends), an algorithm that makes two modifications to the learning rate and one change to the exploration rate of traditional reinforcement learning algorithms. Our learning rates are action-dependent and increase or decrease based on trends in reward sequences. Our exploration rate decreases when the agent learns successfully and increases otherwise. These improvements result in faster learning. We implemented this algorithm in NWScript with the goal of improving the behaviours of game agents so that they react more intelligently to game events. Our primary goal is to provide an agent with the ability to (1) *discover* favourable strategies in a multi-agent computer role-playing game situation and (2) *adapt* to sudden changes in the environment. Our secondary goal is to investigate whether story authors can generate adaptive behaviours quickly and reliably *without programming*. The aim of this chapter is to investigate the viability of reinforcement learning in creating adaptive NPC behaviours in a commercial CRPG.

4.1 Introduction

An enticing game story relies on non-player characters (NPCs or agents) acting in a believable manner and adapting to ever-increasing demands of players. Since the best interactive stories have many agents with different purposes, creating an engaging complex story is challenging. Most games have NPCs with manually scripted

actions that lead to repetitive and predictable behaviours. We extend the behaviour model of Chapter 3 to support reinforcement learning without the need for manual scripting. In Chapter 3, the model selects an NPC behaviour based on motivations and perceptions. The model's implementation generates scripting code for *Neverwinter Nights (NWN)* from a set of *behaviour patterns* built using ScriptEase [70]. The generated code is attached to NPCs to define their behaviours. Although the model described in Chapter 3 supports motivations to select behaviours, a more versatile mechanism is needed to generate *adaptive* behaviours.

An author describes a behaviour motivation in ScriptEase by enumerating motivational attributes and providing them with initial values. Behaviours are selected probabilistically, based on a linear combination of the attribute values. After a behaviour is selected to be performed, the NPC's motivational attributes are updated to express the consequences of the behaviour. For example, the motivation of a guard relies on the **Duty**, **Tiredness**, and **Threat** motivational attributes that control the selection of the patrol, rest, and check behaviours. When patrol is selected, the value of the **Duty** motivational attribute is decreased and the values of the **Tiredness** and **Threat** motivational attributes are increased. An agent that selects behaviours based only on motivations is not able to quickly discover a successful strategy in a rapidly changing environment. Motivations provide limited memory of past actions and lack information about the order or outcomes of actions.

We use reinforcement learning (RL), an unsupervised learning technique, to augment ScriptEase behaviour motivations. An agent learns how to *adapt* to unexpected changes in the environment through experience, by mapping percepts acquired from the environment to actions in order to maximize a numerical reward signal [79]. Without learning, a fighter or a wizard NPC would choose possible behaviours from their **Fighter** or **Wizard** roles randomly or based on the character's motivations. With the help of a learning system, the NPC can learn to choose the actions that result in the best outcome (i.e., actions that maximize reward) for its particular class. This way, the NPC learns what is the best course of action in each situation through interaction with the environment, regardless of the situation. The agent constantly changes its behaviours through experience to improve its ability

to achieve a goal or accumulate long-term reward. Our approach provides NPCs with the ability to learn to adapt online fast, reliably, and effectively [74]. NPCs are immune to mistakes resulting from poor AI implementation, since the AI is pre-tested in the behaviour catalogue and no custom AI is constructed by the author. The learning algorithm allows NPCs to respond creatively to new situations by selecting new behaviours in response to previously unconsidered situations [73].

Our extension of the behaviour model to include learning provides agents with a mechanism to *adapt* to unforeseen changes in the environment by learning. The learning task is complicated by the fact that the agent’s optimal strategy at any time depends on the strategies of the other agents, which creates “a situation of learning a moving target” [6]. More specifically, the learning task is challenging for the following reasons:

1. the game environment changes while the agent is learning (other agents may also change the environment),
2. the other story agents and the player character may also learn,
3. the other agents may not use or seek optimal strategies,
4. the agent must learn in real-time, making decisions quickly, especially to recover from adverse situations, since the system targets a real-time CRPG,
5. the agent must learn and act efficiently, since in most games there are hundreds or thousands of agents. RL techniques are not used in commercial games due to fears that agents can learn unexpected (or wrong) behaviours and because of experience with algorithms that converge too slowly to be useful [66].

The paucity of RL research in commercial CRPGs makes RL in this domain an interesting area of research. In addition, it is not always possible to define an optimal behaviour for a particular agent in a game situation. For example, a guard NPC may not have an optimal patrol behaviour. Moreover, for realism, NPCs should operate under the constraint of partial observability. For example, an NPC should

only perceive and be perceived by nearby creatures. The actions of an agent (NPC or PC) should not be immediately apparent to agents in a different area.

We introduce a variation of a single-agent online RL algorithm, Sarsa(λ) [79], as an additional layer to behaviour patterns with the goal of learning NPC behaviours. To evaluate our approach, we constructed experiments to evaluate learning rates and adaptability to new situations in a changing game world. Although our goal is to learn general behaviours (such as the guard described previously), combat offers an objective environment for preliminary testing, since it is easy to construct an objective evaluation mechanism. Spronck [77] has provided a pre-built arena combat module for *NWN* that is publicly available and he has created learning agents that can be used to evaluate the quality of new learning agents. We evaluate our learning algorithm using this module.

Our experiments show that traditional RL techniques with static or decaying RL parameters do not perform well in this dynamic environment. We identified three key problems using traditional RL techniques in the computer games domain. First, a fixed exploration rate is not suitable for dynamic environments. Second, with action-independent learning rates, the actions that are rarely selected early may be discounted and not “re-discovered” when the environment changes to be more favourable for those actions. Third, fixed learning rates, or learning rates that decay monotonically, learn too slowly when the environment changes. These key problems motivated our changes to traditional RL techniques that resulted in the ALeRT algorithm.

We modify traditional RL techniques in three ways. First, we adjust the exploration rate based on the learning success of the agent. Second, we support action-dependent learning rates. Third, we identify “safe” opportunities to learn fast. In our approach, the agent learns about the effect of actions at different rates as the agent is exposed to situations in which these actions occur. This situation mirrors nature, where organisms learn the utility of actions when stimuli/experiences produce these actions as opposed to learning the utility of all actions at a global rate that is either fixed, decaying at a fixed rate, or established by the most frequently performed actions.

This chapter makes the following contributions:

1. It introduces a mechanism for decreasing the exploration rate when the agent learns successfully and increasing it otherwise.
2. It introduces action-dependent learning rates in RL.
3. It provides a mechanism for increasing the learning rate (i.e., the *step-size*) in RL when prompted by significant changes in the environment.
4. It evaluates an implementation of an RL algorithm with these enhancements through a series of experiments in the demanding environment of commercial computer games (*NWN*) where it outperforms Spronck's dynamic rule-based approach [76] (Spronck's learning method 1) for adaptation speed.
5. It smoothly integrates this RL algorithm into the ScriptEase behaviour code generation.

4.2 Related Work

Several efforts that attempt to improve the behaviours of NPCs have appeared in the literature. Many games use AI techniques, such as decision trees, neural nets, genetic algorithms, and probabilistic methods (e.g., *Creatures* and *Black & White*) only when they are needed and in combination with deterministic techniques [4]. Although successful in these particular games, evolutionary methods are not popular in commercial games, since it is slow to search the space of policies when there are multiple combinations of game states. Moreover, evolutionary methods have not been suitable for handling rarely visited states [37].

RL techniques applied in commercial games are quite rare, since in general it is not trivial to construct a feature vector and the agents adapt too slowly for online game environments [75]. However, in massively multiplayer online role-playing games (MMORPGs), a motivated reinforcement learning (MRL) algorithm generates NPCs that can evolve and adapt in a dynamic environment [54]. The

algorithm (tabular approach) is based on Q-Learning [79] and it uses a context-free grammar instead of a feature vector to represent the environment in which states and actions are dynamically added when certain conditions are met. The agent has only been evaluated against a static opponent. Other researchers [86] have applied RL to learn winning policies in a domination game within a team FPS game that uses the *Unreal Tournament* [84] game engine. Their Q-Learning based algorithm eliminates discount rates and is evaluated against both a static and a dynamic opponent.

The Sarsa algorithm with a linear action-value function approximator was applied to learn the actions of fighter NPCs in a fighting simulation game [32]. The Sarsa(λ) algorithm (tabular approach) was successfully applied to FPS bots to learn the tasks of combat and navigation while collecting items [52]. In the navigation experiment, the *eligibility trace* (discussed later in this chapter) needed to be small, since the best solutions did not require significant planning, while in the combat experiment the eligibility trace needed to be large to facilitate more planning. However, the experiments were performed against a static state machine controlled bot.

Dynamic scripting [76] is a learning technique that combines rule-based scripting with RL. To the best of our knowledge, this is the only RL technique applied to *NWN*, apart from our work. The strategy is updated by extracting rules (e.g., attack) from a rule-base according to their weights and the value function is updated when the effect of an entire sequence of actions can be measured, not after each action. Rules with a positive contribution are rewarded by increasing their weights, while rules with a negative contribution are punished by decreasing their weights. States are encoded in the conditions of the rules in the rule-base. Each type of NPC has a set of rules particular to that type. For example, for a fighter NPC, the size of a script is set to five rules selected from a rule-base of 21 rules. If no rule can be activated, a call to the default game AI is added. However, the rules in the rule-base have to be ordered [83], the agent cannot discover other rules that were not included in the rule-base [74], and, as in the previous cases, the agent has only been evaluated against a static opponent.

Traditionally, estimator variance which leads to learning difficulties is addressed

by tuning the learning parameters, which is a time-consuming process. To address the trade-off between learning rate and estimator variance, researchers [62] proposed a method (the *ccBeta* algorithm based on Q-Learning) for varying the step-size parameter (called β in their work) by an online statistical analysis of the estimate error. Our approach differs in that we introduce action-dependent step-sizes, we modify each step-size parameter, *alpha*, by a fixed amount, and we do not always increase or decrease alpha. Instead, we identify cases in which a trend is not significant and, in these cases, we do not modify the value of alpha. We use the Delta-Bar-Delta approach [78] to detect a trend. In contrast with the Delta-Bar-Delta approach, we introduce action-dependent step-sizes and we detect *significant* trends using a standard deviation factor (f).

4.3 Algorithm

We introduce a step-size updating mechanism that speeds up learning, a variable action-dependent learning rate, and a mechanism that adjusts the exploration rate in RL. We demonstrate this idea using the Sarsa(λ) algorithm.

4.3.1 Sarsa(λ)

Popular RL algorithms include TD, Q-Learning, and Sarsa [79]. The Sarsa algorithm was used for multi-agent systems in computer games [7][57][86]. In Sarsa(λ), the *agent* (a combination of perception, reasoning, and action) learns a *strategy* that indicates what *action* it should take for every *state* by defining an *immediate reward* function, r , the mapping of a state-action pair to a numerical value obtained after performing the action. The *value function*, $Q(s, a)$, defines the value of action a in state s as the total reward an agent will accumulate in the future starting from that state-action. This value function, $Q(s, a)$, must be learned so that a strategy that picks the best action can be found.

At each step, the agent maintains an estimate of Q . At the beginning of the learning process, the estimate of $Q(s, a)$ is arbitrarily initialized. At the start of each episode step, the agent determines the state s and selects some action a to be

taken using a selection policy. For example, an $\epsilon - greedy$ policy selects the action with the largest estimated $Q(s, a)$ with probability $1 - \epsilon$ and it selects randomly from all actions with probability ϵ .

At a step of an episode in state s , the selected action a is performed and the consequences of the action are observed: the immediate reward, r , and the next state s' . The algorithm selects the next action, a' , and updates the estimate of $Q(s, a)$ for all s and a using the Sarsa(λ) algorithm shown in Figure 4.1. The figure shows the linear gradient-descent Sarsa(λ) algorithm with binary features and $\epsilon - greedy$ policy.

The error δ can be used to evaluate the action selected in the current state. If δ is positive, it indicates that the action value for this state-action pair should be strengthened for the future, otherwise it should be weakened. Note that δ is reduced by taking a step (α) toward the target. The step-size parameter, α , reflects the learning rate, therefore, a larger α value has a bigger effect on the state-action value. The algorithm is called Sarsa because the update is based on: s, a, r, s', a' . Each episode ends when a terminal state is reached and $Q(s, a)$ is zero on the terminal step, because the value of the reward from the final state to the end must be zero.

To speed up the estimation of Q , Sarsa(λ) uses *eligibility traces*. Each update depends on the current error combined with traces of past events. Eligibility traces provide a mechanism that assigns positive or negative rewards to past eligible states and actions when future rewards are assigned. When an estimate error occurs, only the eligible states and actions are assigned credit or blame for this error. For each state, Sarsa(λ) maintains a memory variable called an *eligibility trace*. The eligibility trace for state-action pair (s, a) at any step is a real number denoted $e(s, a)$. When an episode starts, $e(s, a)$ is set to zero for all s and a . At each step, the eligibility trace for the state-action pair that actually occurs is incremented by $\frac{1}{n}$, where n represents the number of active features for that state. The eligibility traces for all states decay by $\gamma\lambda$, where γ is the discount rate and λ is the trace decay parameter. The value of the discount rate, γ , is one for episodes that are guaranteed to end in a finite number of steps [79], which is true in our combat scenario. For example, assume that $\gamma = 1$, $\lambda = 0.5$, and the melee action is taken in state s at some step.

```

 $\vec{e}$  accumulating traces
 $s$  state,  $a$  action
 $i$  feature
 $r$  immediate reward
 $\delta$  error in the estimate
 $\vec{\theta}$  feature vector
 $\gamma$  discount rate
 $Q_a$  state-action values
 $\alpha$  step-size parameter
 $\lambda$  trace decay parameter
Initialize  $\vec{\theta}$  arbitrarily
Repeat (for each episode):
     $\vec{e} = \vec{0}$ 
     $s, a \leftarrow$  initial state and action of episode
     $F_a \leftarrow$  set of features present in  $s, a$ 
    Repeat (for each step of episode):
        For all  $i \in F_a$ :
             $e(i) \leftarrow e(i) + 1$  (accumulating traces)
            Take action  $a$ , observe reward  $r$  and
            next state  $s$ 
             $\delta \leftarrow r - \sum_{i \in F_a} \theta(i)$ 
            With probability  $1 - \epsilon$ :
                 $\forall a \in A(s)$ :
                     $F_a \leftarrow$  set of features present
                    in  $s, a$ 
                     $Q_a \leftarrow \sum_{i \in F_a} \theta(i)$ 
                     $a \leftarrow \operatorname{argmax}_a Q_a$ 
            or with probability  $\epsilon$ :
                 $a \leftarrow$  a random action  $\in A(s)$ 
                 $F_a \leftarrow$  set of features present
                in  $s, a$ 
                 $Q_a \leftarrow \sum_{i \in F_a} \theta(i)$ 
             $\delta \leftarrow \delta + \gamma Q_a$ 
             $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$ 
             $\vec{e} \leftarrow \gamma \lambda \vec{e}$ 
    until  $s$  is terminal

```

Figure 4.1: The Sarsa(λ) linear gradient-descent algorithm.

Assume that the state has three active binary features, then $e(s, melee) = \frac{1}{3}$ and the eligibility traces for the rest of the actions in state s are zero. In the next step, if $(s, melee)$ does not occur again, the eligibility trace decays to $e(s, melee) = \frac{1}{6}$ (since $\lambda = 0.5$) and in the next step it further decays to $e(s, melee) = \frac{1}{12}$.

As shown in Section 4.7, the Sarsa(λ) RL algorithm used by a fighter does not converge to the optimal fighter strategy. For example, we experimented with $\alpha = \frac{0.01}{\ln|GS|}$ and variations of $\alpha = \frac{1}{n}$, where GS constitutes the number of features in the game state space and n represents the number of episodes. We also experimented with $\alpha = \frac{\alpha_0}{1 + \frac{steps}{100}}$, where $steps$ is the number of steps in which a behaviour was taken and α_0 is a constant (e.g., 0.1 and 0.3). We experimented with static or decaying (e.g., $\epsilon = \epsilon_0 \frac{(-\ln 2)^* \frac{\epsilon - \epsilon_0}{\epsilon_{halflife}}}{\epsilon_{halflife}}$ and $\epsilon = \epsilon_0 e^{\frac{(-\ln 2)episode}{halflife}}$) ϵ values, where ϵ_0 and $halflife$ are constants (e.g., 0.1 and 25, respectively) and $episode$ is the current episode number. While reducing the value of the learning rate ensures convergence in stationary environments [44], the slow adaptation to changing environments is impractical for our purpose. Moreover, the agent that learns using Sarsa(λ) adapts slower to changes in the environment than the agent that learns using Spronck’s dynamic scripting (Spronck’s learning method 1). Therefore, we designed the enhancements to Sarsa(λ) to create the ALeRT algorithm.

4.3.2 ALeRT

As we mentioned earlier in this chapter, we identified three problems with traditional RL techniques in the CRPG domain. First, a fixed exploration rate is not suitable for dynamic environments. Second, with a global learning rate, the actions that are rarely selected may be poorly represented, especially when the environment changes. Third, traditional learning rates learn too slowly when the environment changes.

Therefore, the ALeRT algorithm, illustrated in Figure 4.2, modifies traditional RL techniques in three ways. First, it adjusts the exploration rate based on the learning success of the agent. Second, it supports action-dependent learning rates. Third, it identifies opportunities to learn fast.

```

Initialize  $\vec{\theta}$  arbitrarily
** Initialize  $\alpha(a) \leftarrow \alpha_{max}$  for all  $a$ 
Repeat (for each episode):
   $\vec{e} = \vec{0}$ 
   $s, a \leftarrow$  initial state and action of episode
   $F_a \leftarrow$  set of features present in  $s, a$ 
  Repeat (for each step of episode):
    For all  $i \in F_a$ :
      **  $e(i) \leftarrow e(i) + \frac{1}{\|F_a\|}$  (normalized accumulating
      traces)
    Take action  $a$ , observe reward  $r$  and
    next state  $s$ 
     $\delta(a) \leftarrow r - \sum_{i \in F_a} \theta(i)$ 
    With probability  $1 - \epsilon$ :
       $\forall a \in A(s)$ :
         $F_a \leftarrow$  set of features present
        in  $s, a$ 
         $Q_a \leftarrow \sum_{i \in F_a} \theta(i)$ 
         $a \leftarrow \operatorname{argmax}_a Q_a$ 
      or with probability  $\epsilon$ :
         $a \leftarrow$  a random action  $\in A(s)$ 
         $F_a \leftarrow$  set of features present
        in  $s, a$ 
         $Q_a \leftarrow \sum_{i \in F_a} \theta(i)$ 
     $\delta(a) \leftarrow \delta(a) + \gamma Q_a$ 
    **  $\vec{\theta} \leftarrow \vec{\theta} + \alpha(a) \delta(a) \vec{e}$ 
     $\vec{e} \leftarrow \gamma \lambda \vec{e}$ 
    **  $\Delta\alpha \leftarrow \frac{\alpha_{max} - \alpha_{min}}{\alpha_{steps}}$ 
    ** if  $\overline{\delta(a)} \delta(a) > 0 \wedge |\overline{\delta(a)} - \mu_{\overline{\delta(a)}}| > f \sigma_{\overline{\delta(a)}}$ 
      **  $\alpha(a) \leftarrow \alpha(a) + \Delta\alpha$ 
    ** else if  $\overline{\delta(a)} \delta(a) \leq 0$ 
      **  $\alpha(a) \leftarrow \alpha(a) - \Delta\alpha$ 

  end of step
until  $s$  is terminal
**  $\Delta\epsilon = \frac{\epsilon_{max} - \epsilon_{min}}{\epsilon_{steps}}$ 
** if  $\sum_{step} r_{step} = 1$ 
  **  $\epsilon \leftarrow \epsilon - \Delta\epsilon$ 
** else
  **  $\epsilon \leftarrow \epsilon + \Delta\epsilon$ 
end of episode

```

Figure 4.2: The ALERT algorithm.

Dynamic Exploration Rate

We use the WoLF principle of “learn quickly while losing, slowly while winning” [5] in our first change to traditional RL techniques to ensure that we explore quickly when we start losing. Recall that the exploration parameter (ϵ) in our $\epsilon - greedy$ strategy is the probability of exploring (selecting randomly from the non-optimal actions). Traditionally [78], ϵ ranges from 0.01 to 0.1, corresponding to exploration between 1% and 10%. We vary our exploration parameter (ϵ) in fixed steps ($\epsilon_{steps} = 15$) between a minimum ($\epsilon_{min} = 0.005$) and maximum ($\epsilon_{max} = 0.02$) value. We selected $\epsilon_{steps} = 15$, since we wanted to be able to move from minimum to maximum exploration rate in a small number of combat episodes. We selected a maximum exploration rate of 2%, since more exploration results in too many lost combat episodes due to an excess number of random actions. Initially, $\epsilon = \epsilon_{max}$, since we would like to explore substantially at the beginning. In general, the value of ϵ increases after a loss, when we need to explore more to find a winning strategy. The value of ϵ decreases after a win, when we do not need to explore as much. However, a minimum value for ϵ is necessary, since occasional exploration is required to discover the optimal strategy in dynamic environments. Figure 4.3 illustrates the ϵ values immediately after a phase change from the Melee to the Ranged equipment configurations (explained later). A phase represents 500 consecutive episodes in which the NPC uses a specific equipment configuration (e.g., Melee). We selected 500 episodes to allow all of the algorithms to converge to the best strategy they could find. When a change in the environment occurs, we need to explore to find a new winning strategy. At the beginning of the second phase, as the agent started losing using the previous best strategy (e.g., speed-melee), the ϵ value increased. As a result, the NPC explored the ranged action twice (episodes 515 and 517) before it finally started exploiting the ranged action (episode 523). At this point, the NPC discovered the new best strategy (e.g., speed-ranged) and started winning consistently. Consequently, the ϵ values started decreasing to the minimum ϵ value, ϵ_{min} .

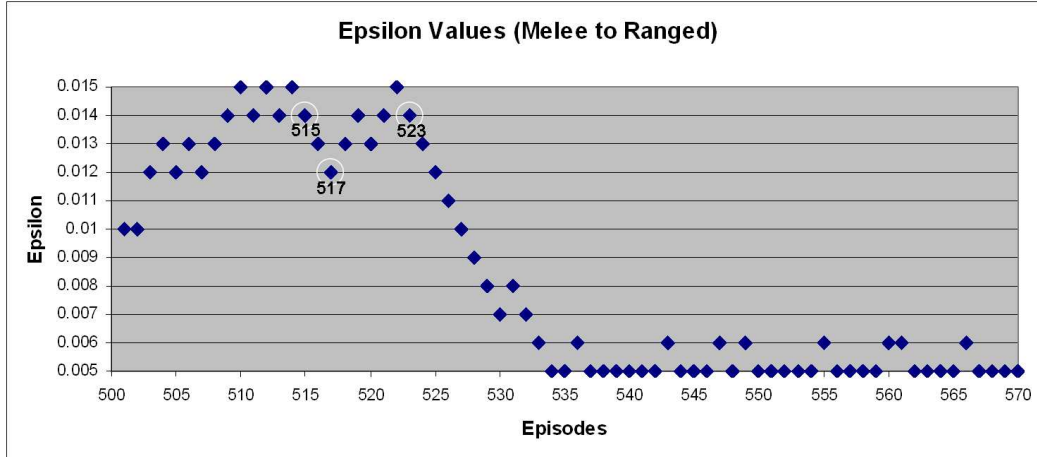


Figure 4.3: Exploration/exploitation (epsilon) values when the phase changes from a Melee to a Ranged equipment configuration. The x-axis shows episodes starting with episode 500 and the y-axis shows epsilon values between $\epsilon_{min} = 0.005$ and $\epsilon_{max} = 0.015$.

Action-dependent Learning Rates

Traditionally, the step-size parameter, α , has either been set to a fixed value to learn in a dynamic context or decayed at a rate that guarantees convergence in an application that tries to learn an optimal static strategy. As stated previously, one of the two main issues with using RL in computer games is the slow learning rate in a dynamic environment [66]. In a game environment, the learning algorithm should not converge to an optimal static strategy because the changing environment can deem this static strategy obsolete (i.e., not optimal any longer). Moreover, a global learning rate does not perform well against a dynamic opponent. We would like learning rates for rare actions (heal or speed) to be different than learning rates for frequent actions (melee or ranged). When the same learning rate is used for all four actions, with some equipment configurations, the NPC does not learn to avoid taking the heal potion even by the end of the first phase (500 episodes). In addition, the NPC forgets to use a speed potion after the transition to the second phase, even though taking the speed potion is still optimal. Varying the exploration rate, ϵ , does not solve these problems. Action-dependent learning rates for taking the heal potion and the speed potion were essential to solve these two problems.

Therefore, the second change we make to traditional RL techniques is to introduce a separate step-size $\alpha(a)$ for each action a . This allows the learning algorithm to accommodate situations in which a new best action for a particular state replaces an old best action for that state, while a different action for a different state remains unaffected. For example, at the beginning of a phase change when an agent in *NWN* acquires a better ranged weapon, the agent will learn quickly to take a ranged action in the second step of the combat instead of taking a melee action, because the $\alpha(\text{ranged})$ value is high. However, the agent will still correctly take a speed potion in the first step of a combat episode, since the $\alpha(\text{speed})$ parameter is unaffected by the change to the $\alpha(\text{ranged})$ values. We expect $\alpha(a)$ to converge to a small value when the agent has settled on a best strategy and the environment does not change. Otherwise, $\alpha(a)$ will be elevated, so that the agent learns fast. An elevated α value often indicates a rare action (e.g., heal or speed) whose infrequent use has not yet decayed its step-size from the high value at the start of training. The agent's memory has faded with regards to the effect of this action. When this rare action is used, its high α value serves to recall that little is known about this action and the current utility is judged on its immediate merit. This situation is reminiscent of the start of training when no bias has been introduced for any action.

Trend Detection

Although the introduction of action-dependent learning rates solved the heal and speed problem, it did not solve another issue: what to do after a string of losses. We encountered several situations, for both a static and a changing environment, in which our agent could not determine if the environment had changed or not. The problem was that we could not distinguish whether a sequence of losses was due to an environment change, a change in the opponent's strategy, or bad luck due to stochasticity. We needed a mechanism to differentiate among an unlucky sequence of losses (even during a winning strategy), changes in the environment (e.g., new equipment configuration), ambiguous scenarios where no clear best strategy can be identified, or combinations of these situations. We tried to reset the learning rate when we detected a series of losses, but this approach was not successful. Our

solution was to rapidly decrease/increase the learning rate only under certain conditions. This solution is more flexible than identifying a fixed window of consecutive losses, since a sequence of losses followed by a single win and another sequence of losses would not be detected by the window approach.

Therefore, the third change we make to traditional RL algorithms is to speed up the learning rate when there is a recognizable trend in the environment and slow it down when the environment is stable. This supports fast learning when necessary, but reduces variance due to non-determinism in stable situations. Since α is the learning rate, the problem reduces to one of determining a good time to increase or decrease α . When the environment changes enough to perturb the best strategy, the estimator of Q for the new best action in a given state will change its value. The RL algorithm will adapt by generating a sequence of positive δ values, as the estimator of Q continually underestimates the reward for the new best action until the estimate of Q has been modified enough to identify the new best action. Conversely, when the environment is stable and the policy has already determined the best action, then there is no new best action, so the sequence of δ values will have random signs, indicating that no trend exists.

When a trend for a specific action is detected, α increases to revise the estimate of Q faster. When there is no trend, α decreases to reduce the variance in a stable environment. We recognize the trend using a technique based on Delta-Bar-Delta [78]. We compute *delta-bar*, the average value of δ over a window of previous steps that used that action and then compute the product of the current δ with *delta-bar*. We combine action-dependent *alphas* with trend-based *alphas* in that there is a separate *delta-bar* for each action. When the product of δ with *delta-bar* is negative, the opposite TD-error signs indicate an oscillation of the TD-error values around a mean value, so α is decreased to lower variance. When the product is positive (as illustrated in Figure 4.4), there is a positive correlation between the current δ and the δ trend, so α is increased to learn the new strategy faster. The x-axis of Figure 4.4 shows 22 consecutive episodes in which the speed potion was taken, starting with episode 408. The y-axis shows the parameter values for δ and *delta-bar-delta*. However, we modified this approach using significant trends to ensure that variance

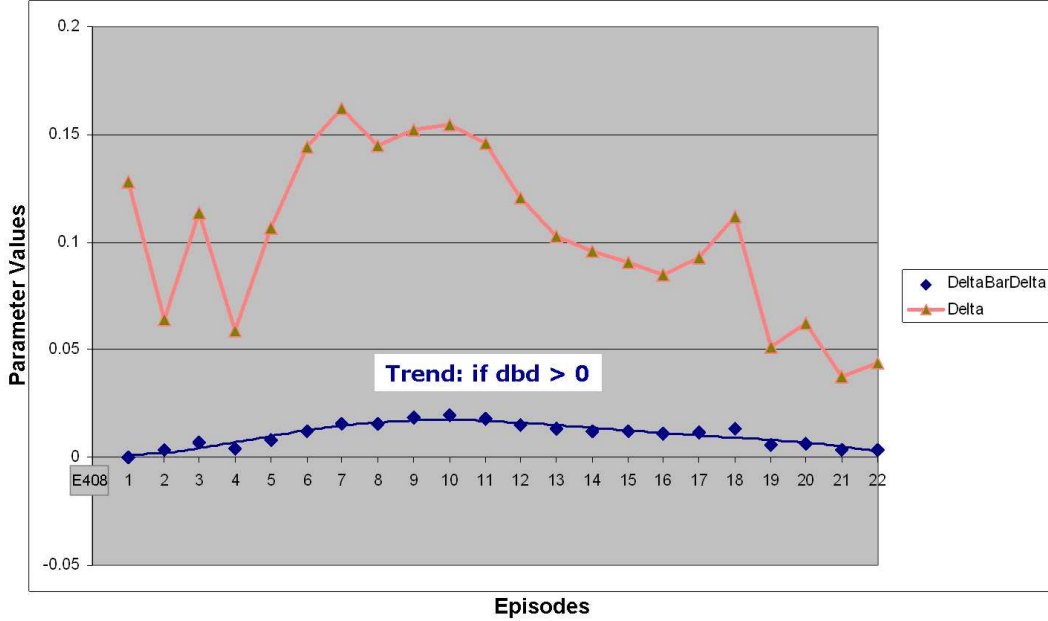


Figure 4.4: A trend is detected for the speed action starting with episode 408 of a Melee configuration. The upper trace represents the δ values for the speed action, while the lower trace represents the delta-bar-delta (dbd) values for speed immediately after the NPC drank a speed potion.

remains low and to accommodate unfair situations for our agent, where the best strategy may not be able to attain a tie. Figure 4.5 shows the variables we compute to determine whether the trend detected in Figure 4.4 is significant.

We detect a *significant trend* for an action when delta-bar (the average of the individual δ values for the action over a fixed window of δ values) differs from the *average* delta-bar, $\mu_{\bar{\delta}}$, computed from the beginning of the game for that action by more than a factor (f) times the *standard deviation* of delta-bar, $\sigma_{\bar{\delta}}$, also computed from the beginning of the game, as illustrated in Figure 4.6.

Although delta-bar is computed over a fixed window of the latest δ values for a particular action a , the *average* and the *standard deviation* of delta-bar are computed over the entire set of episodes. Initially, $\alpha = \alpha_{max}$, since the agent needs to use a high step-size when the best strategy is unknown.

If there is a significant trend ($\bar{\delta}\delta > 0 \wedge |\bar{\delta} - \mu_{\bar{\delta}}| > f\sigma_{\bar{\delta}}$), we increase α to learn faster, as illustrated in Figure 4.7. We change α in fixed size steps ($\alpha_{steps} = 20$) between α_{min} and α_{max} , although step sizes proportional to delta-bar are

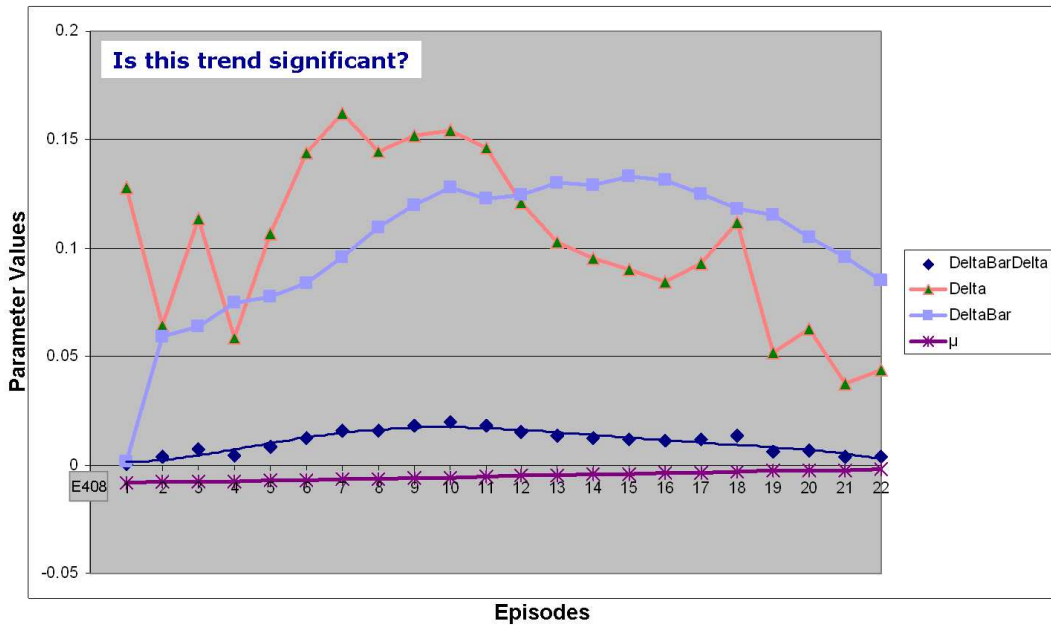


Figure 4.5: We compute the delta-bar and the average delta-bar, $\mu_{\bar{\delta}}$, for the speed action to determine whether the trend for the speed action is significant.

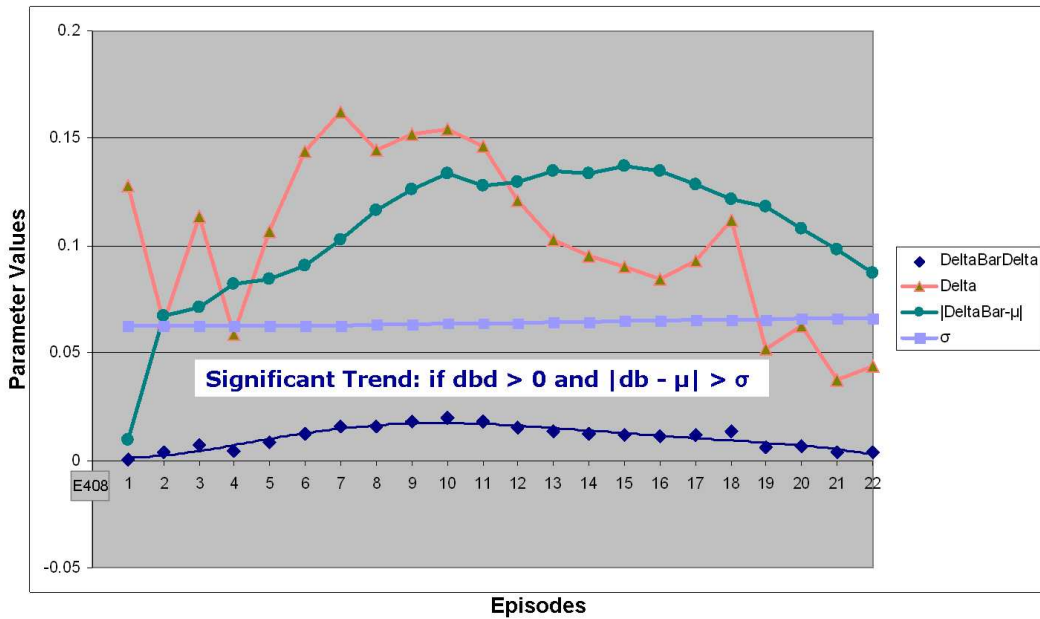


Figure 4.6: We identify a significant trend for the speed action: the current variation of delta-bar from the average delta-bar, $\mu_{\bar{\delta}}$, exceeds the standard deviation ($f = 1$) of delta-bar, $\sigma_{\bar{\delta}}$, for the speed action, while delta-bar-delta is positive.

reasonable. The value 20 was selected empirically and we found that a slightly slower (20) change rate for the learning rate was more effective than the faster (15) change rate for the exploration rate.

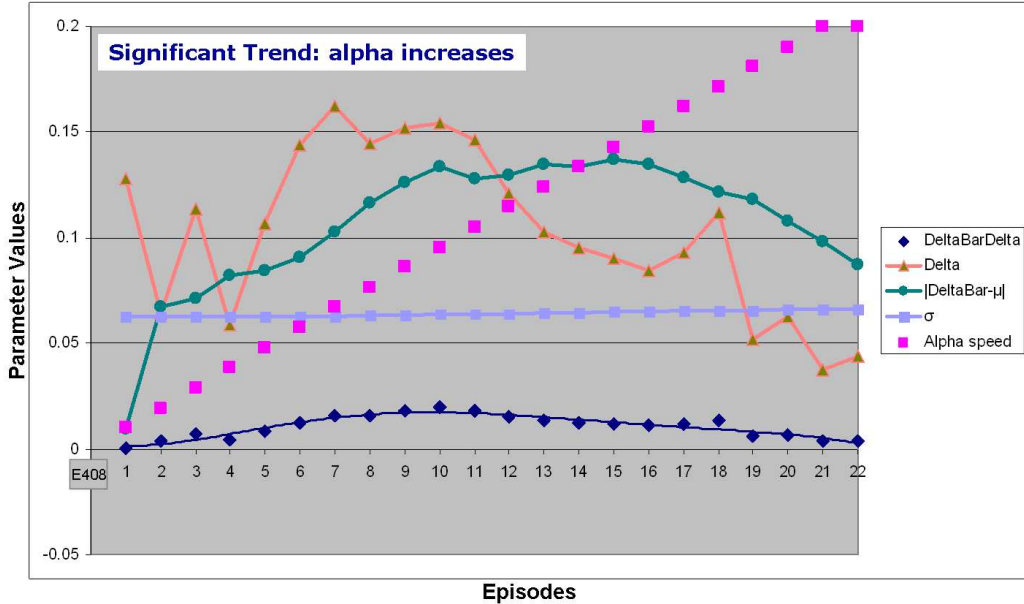


Figure 4.7: When a significant trend for the speed action is detected, the value of the alpha parameter for the speed action is increased, so that the NPC learns faster.

If we identify a trend that is not significant ($\bar{\delta}\delta > 0 \wedge |\bar{\delta} - \mu_{\bar{\delta}}| \leq f\sigma_{\bar{\delta}}$), we do not change α .

In the stable case in which no trend is detected ($\bar{\delta}\delta \leq 0$), α decreases to reduce variance so that the estimate of Q does not change significantly, as illustrated in Figure 4.8 for a melee action. However, α is inferior bounded by α_{min} , because the agent needs to respond to future changes in the environment that may alter the best strategy.

For example, if a new powerful ranged weapon has been obtained, then the best new action in a combat situation may be a ranged attack instead of a melee attack. However, the damage done or taken each round (i.e., a game unit of time that lasts six seconds) varies due to non-determinism, therefore the values of the δ for the new best action (e.g., ranged) may vary as well. Figure 4.9 shows the α parameter values (y-axis) for the ranged and the melee actions starting with episode 501, i.e., after

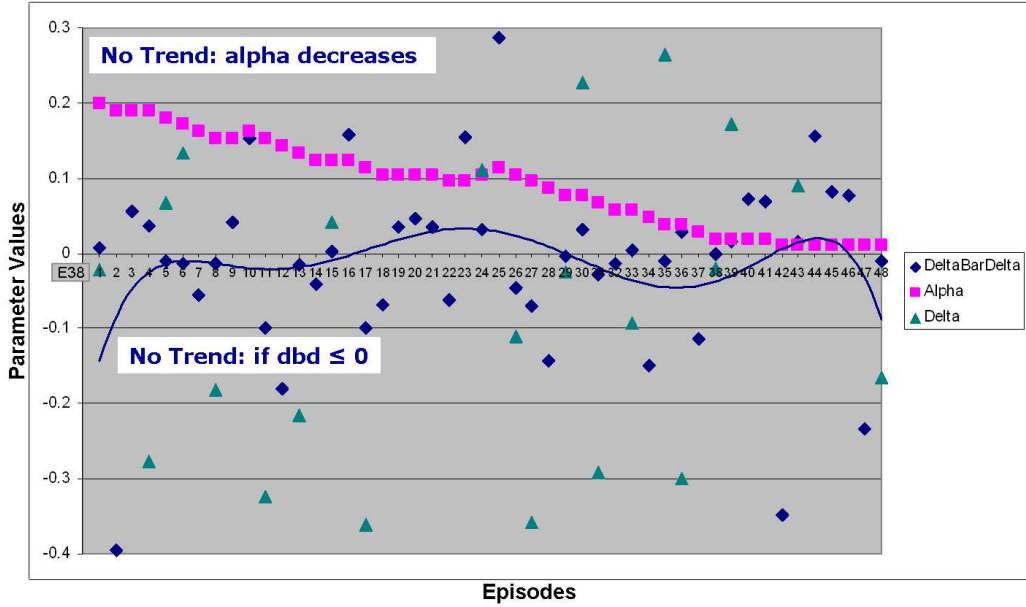


Figure 4.8: No trend is detected for the melee action at episode 38 of a Melee configuration, therefore the value of the alpha parameter for the melee action is decreased.

a phase change from Melee to Ranged. The x-axis shows the steps in which these actions were taken. In the $\alpha(\text{ranged})$ trace, only the steps where the range action was taken are shown, since other actions do not affect the value of $\alpha(\text{ranged})$. Similarly for the $\alpha(\text{melee})$ trace. This is the reason why there is a different number of steps in each episode of each trace. An episode is not included in a trace if the appropriate action was not taken during that episode. For example, in episode 501, the ranged action was taken 14 times and the next time the ranged action was selected was in episode 515 where it was selected only once. The figure indicates how the α values for the ranged action change from episode 501 to episode 535 by which the NPC has learned the new winning strategy. It also shows the α values for the melee action from episode 501 to episode 509. The episode numbers in which the actions are selected are marked on the α traces for the two actions. The delta-bar-delta trends for the two actions are illustrated in Figure 4.9 as well. Notice that since the x-axis represents consecutive steps in which an action is taken and melee is taken more frequently than ranged (at least near episode 500), the episodes proceed more slowly for melee.

After the phase change from Melee to Ranged (episode 501), the ranged action is not selected until episode 515, since speed-melee was the winning strategy at the end of the first phase. In this example, the ranged action was rarely selected in the first phase. The last exploited ranged action occurred in episode 44, while the last explored ranged action occurred in episode 278. Therefore, $\alpha(\text{ranged})$ remained high (0.18). When the ranged action is selected in the second phase (through exploration in episodes 515 and 517), the resulting delta-bar-delta values are negative, indicating that the new values are not part of a trend. Therefore, the $\alpha(\text{ranged})$ values decrease. When the NPC discovers that the new action yields good results (through exploitation in episode 523), a trend for the ranged action emerges (episodes 523 to 526, steps 17 to 25 in Figure 4.9). As a result, the α values for the ranged action increase, reaching the maximum α value, $\alpha_{max} = 0.2$. In episodes 527 and 528, steps 30 to 35, a trend for the ranged action is detected, but it is not significant. Therefore, the α values for ranged remain unchanged. By episode 535, the α values for ranged steadily decrease (no trend is detected) to the minimum α value, $\alpha_{min} = 0.01$. A small $\alpha(\text{ranged})$ value shows that the NPC learned to properly use the ranged action, in this case to replace the melee action with the ranged action. The overall delta-bar-delta values for ranged are positive at the beginning, as the NPC progressively learns about the consequences of the ranged action, and they have random signs as the NPC finally discovers that ranged is the best new action.

Before the phase change, the winning strategy was speed-melee, so $\alpha(\text{melee})$ starts low (0.05). The new δ values for melee indicate no trend (as they did at the end of the first phase), therefore in episode 501 the $\alpha(\text{melee})$ values decrease. A trend is detected in episode 502, steps 12 to 16, but it is not a significant trend, therefore the $\alpha(\text{melee})$ values remain the same. A significant trend for melee occurs in episodes 502 and 503, steps 18 to 21, as well as in episode 505, steps 35 and 36, therefore the $\alpha(\text{melee})$ values increase. However, this is due to non-determinism, as the NPCs have not changed their strategies yet. This situation is reflected in the random signs of the delta-bar-delta values for melee. After episode 515 when the ranged action starts to be selected, the melee action is rarely selected, but since $\alpha(\text{melee})$ already reached α_{min} , the $\alpha(\text{melee})$ values remain low until the end of the

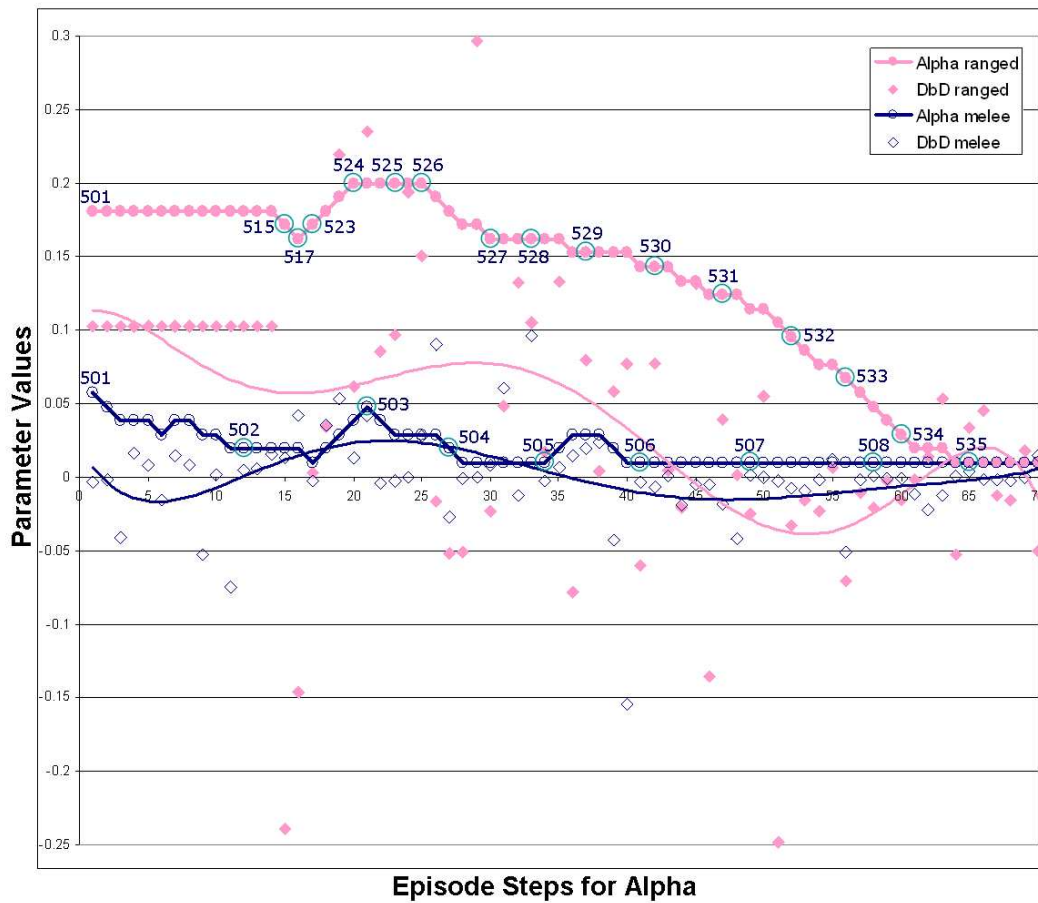


Figure 4.9: Phase change from a Melee to a Ranged configuration: after the NPC learns to favour the ranged action, the values of the alpha parameters for both the melee and the ranged actions decrease.

second phase. As the previous best action (e.g., *melee*) is replaced with the new best action (e.g., *ranged*), the $\alpha(\text{melee})$ values decrease to α_{min} . In this case, the NPC learned that the *melee* action is not a winning action anymore.

The ALeRT algorithm is shown in Figure 4.2, in which we mark with two asterisks (**) our changes to the Sarsa(λ) algorithm. We are using the generalization approach in our algorithm, rather than the other common type of policy representation, the tabular approach. We use a function estimator to generalize the mapping of a state to an action, whereas the tabular approach uses a lookup table to store values of an action in a particular state. For simplicity, $Q(s, a)$ denotes the estimator of Q .

4.4 Implementing ALeRT in *NWN*

We define each NPC to be an *agent* in the *environment* (*NWN* game), controlled by the computer, whereas the PC (player character) is controlled by the player. The NPCs respond to a set of game events, such as *OnCombatRoundEnd* (CRE), *OnDeath*, *OnPerception*, etc. If an event is triggered and the NPC has a script for that event, then that script is executed. *NWN* combat is a zero-sum game (one agent's losses are the opponent's wins). We define an *episode* as a fight between two NPCs that *starts* when the NPCs are spawned and perceive each other as enemies (having maximum hit points) and *ends* as soon as one of the agents dies (has zero or less hit points). We do not consider draws, since there is always one NPC who dies first. We define a *step* as a combat round during which the NPC must decide what action to select and then execute the action. A combat round is defined in the game by the CRE event that fires every six seconds in the game.

We define a *policy* as a set of rules that tells the agent what action to take for every state of the game. It is a mapping from perceived states of the environment to actions taken when in those states. We use an ϵ -*greedy* policy to select an action to perform in each state according to the current Q_a values. We estimate our Q_a value using a feature vector $\vec{\theta}$ that has one feature for each state-action pair. The *value* of a state represents the total long-term amount of reward that the agent is expected to acquire in that episode, starting from that state. The value of a state takes into

account the potential following states and their associated rewards. Therefore, the agent favours actions that result in states of maximum value not maximum reward, since these actions produce the greatest amount of reward in the long run. For example, the speed action may always yield a low immediate reward if the opponent chooses a melee or a ranged action, but it can still have a high value because it is regularly followed by other states, such as melee, that yield high rewards.

The *state space* S illustrated in Figure 4.10 consists of five Boolean features:

1. *HSL*: the agent's HP are lower than half of its initial HP and the agent has a potion of heal available,
2. *EA*: the agent has an enhancement potion available,
3. *EO*: the agent has an enhancement potion active, since approximately six rounds after drinking the enhancement potion, this potion ceases to be active,
4. *DM*: the distance between the NPCs is in the melee interval (the NPCs are close to each other), and
5. *CONSTANT*: a constant with value 1 that expresses the importance of an action in comparison with the rest of the actions, especially when none of the other features is active.

The *action space* A consists of four actions: melee, ranged, heal, and speed. Therefore, there are 20 features in the $\vec{\theta}$ feature vector, one for each state-action pair. For example, if features 1, 3 and 5 are active and a melee action is taken, three components of \vec{e} will be updated in this step: 1-melee, 3-melee, and 5-melee. These updates will influence the same components of $\vec{\theta}$ and will affect the estimator of Q_{melee} in future steps. The action space is illustrated in Figure 4.11.

When the agent is in a particular game state, the action to take is chosen based on the estimated values of Q_a using an $\epsilon - greedy$ policy. When we exploit (choose the action with the maximum estimated Q_a for the current state) and there is a tie for the maximum value, we randomly select among these actions. We *explore* using a uniformly random approach (irrespective of the estimated Q_a values). Using

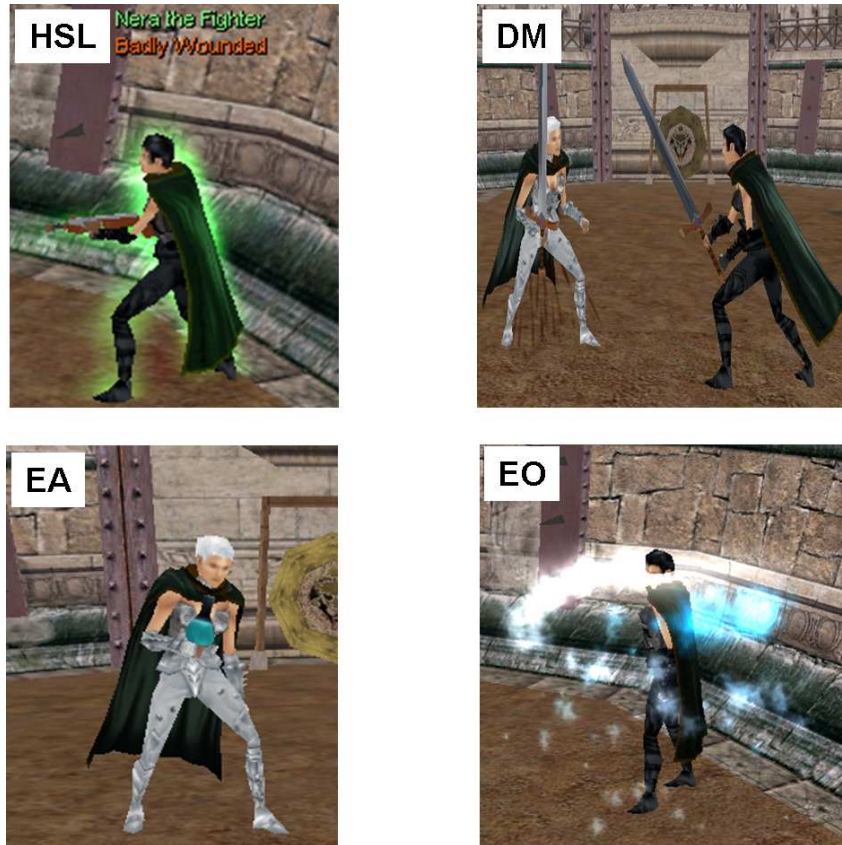


Figure 4.10: Four of the states in the state space for the **Fighter** role: *HSL* (the NPC is injured), *EA* (the NPC has the speed enhancement potion in hand), *EO* (the NPC has just drunk the speed enhancement potion), and *DM* (the NPCs are within melee distance). The constant state is not represented in this figure.



Figure 4.11: The action space for the **Fighter** role.

exploration, we can select actions that lead to states that we may never select otherwise.

We define the *score* of a single episode as 1 if the agent wins the episode and -1 if it loses. The agent’s goal is to win a fight consisting of many consecutive episodes, i.e., to maximize the total reward in the long run. One way to measure the agent’s success is to assess the difference in hit points (HP) lost from one step to another between the opponent and the agent. We define the *immediate reward*, r , at the end of each step of each episode in formula 4.1:

$$r = 2 * \left(\frac{HP_{s'}}{HP_{s'} + HP_{o'}} - \frac{HP_s}{HP_s + HP_o} \right), \quad (4.1)$$

where the subscript s represents the agent (self), the subscript o represents the opponent, a prime ($'$) denotes a value after the action and a non-prime represents a value before the action. As a validation of this formula, the sum of all the immediate rewards during an episode amounts to 1 when our agent wins and -1 when our agent loses. Moreover, the sum of all the rewards throughout the game amounts to the difference in episode wins and losses during the game.

A *standard action* [60] is a type of action that may be executed during a *combat round*, such as a melee or a ranged attack, casting a spell, or using a magic item. A combat round is the interval between two consecutive *OnCombatRoundEnd* events that occur every six seconds in the game. An agent can execute a standard action and a *move action* during a combat round, however, it cannot execute another standard action in place of the move action.

We consider an *episode* to be the interval between the perception of an enemy, which triggers the game’s *OnPerception* event, and the death of one of the NPCs, which triggers the game’s *OnDeath* event. The definition of an episode may be extended from individuals to teams. More specifically, for individual combats (teams of one NPC), the death of either of the two NPCs marks a terminal state for our NPC. For teams, the death of all the members of either team marks a terminal state for our algorithm. In the combat framework that we used for our experiments [77], when either team dies, the remaining NPCs from the other team are destroyed in preparation for the next episode. Any necessary updates of the data structures are

performed before the team is destroyed.

Some actions, such as drinking a speed or a heal potion, do not require the entire combat round duration to execute. If our NPC is attacked immediately after drinking a heal or a speed potion, then a script is executed, so that the NPC has the opportunity and time to *select* an action to execute for the next combat round before this combat round ends. We call this action an *early action*. However, the NPC cannot legally *execute* this early action, since two actions must not be performed during the same combat round.

Both the original *NWN* AI and Spronck's AI enqueue actions immediately when any one of the following four events occurs: attack (*OnPhysicalAttacked*), damage (*OnDamaged*), cast a spell (*OnSpellCastAt*), or disturb (*OnDisturbed*). We have already given an example using the *OnPhysicalAttacked* event. The other three events provide similar opportunities to select an early action.

We discovered the necessity of early actions by observing an unfair number of actions performed by the opponent NPC during early experiments. In our initial implementation, we cleared all actions of our NPC's action queue at the end of every CRE and we did not enqueue the next action as soon as a previous action was complete. Instead, we always waited until after the next CRE. As a result, in cases in which an early action should have been selected, we found that the opponent performed more action strikes than our NPC. Effectively, we ignored the opportunity to select an early action when one of the four early action triggering events occurred. Since neither the default *NWN* strategy nor Spronck's strategy ignore early events, we modified our code to support early actions to ensure a fair number of actions for each NPC.

For example, assume that the first step of an episode starts with the selection and then the execution of a speed or heal action. If the NPC is attacked, then an early action is triggered and the feature vector and action probabilities are updated, and a new action is selected (but not executed). We cannot update the immediate reward and the data structures, since the number of hit points of each NPC may change before the CRE occurs and these changes must be reflected in the immediate reward. However, when a CRE is reached, the feature vector and action probabilities are

updated. If there is no early action selected, a new action is selected. In either case, we also compute the immediate reward for the previous action and update the data structures. The reason we do not wait until the CRE to select our new action is because this can result in fewer actions for our NPC. As indicated earlier, the full effect of an action (e.g., current hit points of each NPC after the execution of an action) can only be known after a CRE. There is also an issue caused by the lack of synchronization between the NPCs' CREs. An *attack of opportunity* is a single extra melee attack that is performed when an enemy performs actions such as drinking a potion, casting a spell, or using a ranged weapon. The effect of an attack of opportunity by the enemy NPC is recorded during the enemy's action, which may occur during, before, or after our NPC's combat round. Therefore, we only apply the update of the main data structures used in the ALERT algorithm after a CRE is reached.

4.5 Using RL in ScriptEase

We developed a mechanism that enables story authors to automatically gain the benefits of a learning system by using an enhanced behaviour model. We can distinguish three categories of story designers that use ScriptEase. The first category, authors, encompasses designers who use ScriptEase patterns with only small adaptations (e.g., setting options) and who only use the ScriptEase *Story Designer* tool. The second category, pattern designers, consists of designers who create their own patterns using the ScriptEase *Pattern Designer* tool. The third category, programmers, consists of designers who create new atoms that will be used for patterns, using the ScriptEase *Pattern Designer* tool.

Our goal is to show that ScriptEase is flexible and it can be easily modified to support learning for a real game environment to facilitate NPC adaptation to new situations. Moreover, this learning system can be used by game story authors without programming.

4.5.1 Using an RL Performance

An author uses our learning system by instantiating a performance (e.g., **RL-Combat** performance illustrated in Figure 4.12). The author sets the only option of this performance, **ACTOR**, to a specific NPC. Then, the author sets the options of all roles that are not activated by explicit role cues, sets the role cue options, expands all the role cues, and sets the role options. An RL performance contains two roles: **RL-Listener-Name** and **RL-Name**, where *Name* can be substituted by the name of the particular learning role (e.g., *Combat*) as shown in Figure 4.12.

For example, to use an NPC with combat learning, the author may instantiate the **RL-Combat** performance illustrated in Figure 4.12. For brevity, we do not show the definition atoms included in the *RL* role cue in this figure. This performance contains both the **RL-Combat** role and the **RL-Listener-Combat** role. The author has added the **Guard** role to this performance. Alternately, the author can start with a **Guard** performance and add the **RL-Combat** and the **RL-Listener-Combat** roles. The **RL-Combat** role is activated by the *RL* cue, i.e., by the presence or actions of an enemy.



Figure 4.12: An RL performance in ScriptEase that contains the **RL-Combat** and **RL-Listener-Combat** roles.

The author does not have to modify the RL components (e.g., RL parameters) to use a behaviour. We set default values for behaviour patterns from the moment the patterns are created. Our behaviour system provides added value for the author who takes advantage not only of the power of behaviour patterns, but also of the convenience of a transparent learning system.

4.5.2 RL Cue Components

The components of the *RL* role cue can be inspected to understand how RL is represented in ScriptEase. However, an author does not need to see these components in order to use an RL performance. A pattern designer must assemble these components to create a new RL pattern.

The **RL-Listener-Name** role contains a behaviour cue for each event that triggers a single learning step. These behaviour cues contain an action atom, **Activate RL cue**, that triggers the start step and each continuation step of the learning process by activating an *RL* role cue. The *RL* role cue activates the **RL-Name** role. The **RL-Name** role contains the default *Start-RL* behaviour cue that selects a single behaviour for each learning step as soon as the role becomes active. The *Start-RL* behaviour cue is a refinement of the normal *Start* behaviour cue that adds support for RL data structure updates. Therefore, all the RL basic behaviours are triggered by the *Start-RL* behaviour cue of the **RL-Name** role. The **Activate RL cue** action activates the *RL* role cue and, at the same time, it fires the user-defined event that corresponds to the *Start-RL* behaviour cue of the activated role. Once the **RL-Combat** role is activated, it remains active until it is changed to a different role. Therefore, we need to ensure that the basic behaviours are triggered by the *Start-RL* behaviour cue to prevent them from being performed continuously while the role is active, without complying with the combat round boundaries.

Since the **RL-Listener-Name** role is not activated by a role cue, it is inherited by all roles in the performance, including the **RL-Name** role. Therefore, while the **RL-Name** role is active, it can respond to all the behaviour cues included in the **RL-Listener-Name** role, in addition to its own behaviour cues. The end of the learning process is triggered by behaviour cues in the **RL-Name** role. Thus, when the learning ends, the NPC's role may be set to the role used before learning started. The role **RL-Name** (e.g., **RL-Combat** shown in Figure 4.13) contains at least two behaviour cues, one to select the appropriate behaviour in an RL step (*Start-RL* behaviour cue) and at least one to end a learning episode (either *Creature death* or *No enemy perception* behaviour cues). A behaviour cue that ends a learning episode can also optionally restore the role. The end of a learning episode is triggered by



Figure 4.13: The **Activate RL cue** action starts a generic RL step, the **Activate RL early cue** action starts an early RL combat step, and the **End episode** action ends a generic RL episode.

specific events defined by the author depending on the particular learning situation. For example, in a combat situation, the end of an episode may be marked by either the *Creature death* (implemented using an *OnDeath* event) or the *No enemy perception* cue (implemented using as an *OnHeartbeat* event that polls for enemies in the area), as illustrated in Figure 4.13. As in the previous figure, we do not show the definition atoms included in the *RL* role cue for brevity. The behaviour cues that trigger the end of an episode contain two ScriptEase actions, **End episode** and **Activate previous role**. The **End episode** action updates the data structures involved in the learning process. The author may specify a scenario that happens in the last step of each episode using the behaviour cues that end an episode. The **Activate previous role** action changes an NPC's current role to the NPC's previous role and re-spins. For example, when an NPC who performs a **Guard** role perceives an enemy, the *Enemy perception* role cue changes the NPC's role from **Guard** to **RL-Combat**. More specifically, the *Enemy perception* role cue activates the *RL* role cue that activates the **RL-Combat** role. The *Start-RL* behaviour cue is activated immediately when the role is activated. As a consequence, the NPC selects one of the basic behaviours triggered by the *Start-RL* behaviour cue. When the fight ends, one of the two behaviour cues, *Creature death* or *No enemy perception*, changes the NPC's role from **RL-Combat** to the NPC's previous role, **Guard**. The author may delete the optional **Activate previous role** action that restores an NPC's role to the NPC's previous role, if this does not reflect the author's intent.

If an author modifies the structure of an existing behaviour pattern, then the author must modify the *state space* if necessary. Note that the author does not need to provide the *action space*, since ScriptEase automatically generates the action space from the behaviour structure. Each action corresponds to a basic behaviour included in a role. For example, if an author adds a new behaviour to a role, the action space is automatically updated, but the state space may need to be modified. Figure 4.14 shows the components of the *RL* role cue for the combat RL example from Figure 4.13.



Figure 4.14: The components of the *RL* role cue. The default parameter values were determined empirically by the pattern designer.

Reward Function

The reward function is represented in ScriptEase using a definition atom. This definition, named *SE_RL_REWARD*, is included at the *RL* role cue level as illustrated in Figure 4.14. The RL system recognizes this particularly named definition as being the RL reward. If the author needs to modify the reward function, a new definition with this name can be added to replace the default reward.

RL Parameters

The RL parameters are represented in ScriptEase using definitions. These definitions are included at the *RL* role cue level as illustrated in Figure 4.14. An author may adjust an RL parameter value (e.g., set the *SE_RL_LAMBDA* parameter to 0.5) or it can replace an RL definition (e.g., *SE_RL_EPSILON*), but it cannot change the name of the definition.

State Space

The state features are represented in ScriptEase using binary definitions, as illustrated in Figure 4.14. Definitions for each of the binary features are included at the *RL* role cue level. A pattern designer who desires to change the state space may modify, add, or delete the binary definitions. The labels of the binary definitions that constitute the state space are marked with the *SE_RL_* prefix. For example, if a new binary state, *DSO* (*defensive spell on*), is added to the state space, the author may add as many auxiliary definitions necessary for the computation of the new state, followed by a binary definition with the label *SE_RL_DSO* for the new state.

Action Space

The actions comprising the action space are automatically generated from the behaviours contained in a role. No definitions are required to express the action space. For example, for the **Guard** role, the action space consists of four actions, **melee**, **ranged**, **heal**, and **speed** that correspond to the four basic behaviours **Melee**, **Ranged**, **Heal**, and **Speed**.

Auxiliary Definitions

These definitions are used to compute the state space features. They retrieve auxiliary objects and values needed to create the binary definitions for the state space. The auxiliary definitions are added at the *RL* role cue level, after the definitions necessary for the RL parameters, i.e., after the definition of the δ window size illustrated in Figure 4.14. For example, an auxiliary definition, *Opponent*, is used to compute the distance between our agent and its closest enemy. The computed state, *DM* (distance melee), is represented as a ScriptEase binary definition.

4.5.3 Creating and Adding RL Behaviours

Suppose that a pattern designer needs to add a fifth behaviour to a **Fighter** role, which causes the NPC to move a certain distance away from the enemy. This behaviour provides the fighter NPC with the ability to evade the enemy, drink an available potion of speed or heal if necessary, and then resume the combat. To accomplish this, the pattern designer needs to select the **Move away** behaviour from the library of available behaviours. If this behaviour is not available in the library, the pattern designer can easily create it by adapting the **Approach** task of the **Approach** behaviour, in which the **Move** action atom (a call to the NWScript function `ActionMoveToObject`) is replaced with the new **Move away** action atom (a call to the NWScript function `ActionMoveAwayFromObject`).

When the **Fighter** role is compiled in ScriptEase, the new **Move away** behaviour is automatically added to the action space as the *move away* action. No change is necessary to the state space. However, if the pattern designer needs to create an NPC that learns to retreat only when it has very low hit points, the concept of “very low hit points” must be added to the state space. For example, the pattern designer may define a new state that indicates when the NPC’s hit points reach 10% of their initial value. The pattern designer simply selects an already available binary ScriptEase definition to express this binary game state. In general, the pattern designer adds binary ScriptEase definitions for each of the state features. Sometimes auxiliary definitions are needed to create the state definitions. For ex-

ample, the opponent is set whenever an event activates the *RL* role cue (e.g., the attacker in the *OnPhysicalAttacked* event). The auxiliary definition, *Opponent*, in Figure 4.14 retrieves this opponent. The opponent is needed to compute the binary *DM* (distance melee) state definition that is added to the *RL* role cue. In a similar manner, binary definitions for *HSL*, *EO*, *EA* are added, as illustrated in Figure 4.14. The *Constant* feature is implicit to the *RL* role cue, thus it is not added by the pattern designer.

```

//Compute Immediate Reward
float SCEZ_RL_UpdateReward(object self, object opponent, string behaviour) {
    //Current hit points self
    float currentHPs = IntToFloat(GetCurrentHitPoints(self));

    //Current hit points opponent
    float currentHPo = 0.0;
    //If the HPs drop below 0, we set them to 0.
    if ( !SCEZ_RL_IsStateTerminal(self, opponent) ) {
        currentHPo = IntToFloat(GetCurrentHitPoints(opponent));
        if (currentHPs < 0.0) currentHPs = 0.0;
    }

    SCEZ_RL_PrintString(" CurrentHPs " + FloatToString(currentHPs));
    SCEZ_RL_PrintString(" CurrentHPo " + FloatToString(currentHPo));

    //Previous hit points
    float prevHPs = SCEZ_RL_GetPreviousHPs(self);
    float prevHPo = SCEZ_RL_GetPreviousHPo(self);

    float currentDelta = SCEZ_RL_DivideFloats(currentHPs, currentHPs + currentHPo);
    float previousDelta = SCEZ_RL_DivideFloats(prevHPs, prevHPs + prevHPo);

    //Immediate reward at the end of each step of an episode
    float reward = 2.0 * (currentDelta - previousDelta);
    SCEZ_RL_UpdatePreviousHP(self, currentHPs, currentHPo);
    return reward;
}

```

Figure 4.15: NWScript code for the update of the reward function.

4.5.4 Creating and Adding RL Roles

Many times, a pattern designer would like to create new roles based on existing roles or apply the RL framework to non-combat learning situations.

Other Combat RL Roles

For example, a pattern designer may want to create a different combat role (e.g., a **Sorcerer** role). The sorcerer NPC performs five behaviours: casts a spell, drinks

a speed potion, drinks a heal potion, fights with a melee weapon, and fights with a ranged weapon. Behaviour patterns for these behaviours exist in ScriptEase.

The pattern designer needs to first create a new role in ScriptEase and add any necessary options. The `ACTOR` option is available by default when the role is created. Then, the pattern designer instantiates each of these behaviour patterns in the **Sorcerer** role. Since the game state space is not defined, the pattern designer needs to add binary ScriptEase definitions representing the state features. Since the **Fighter** and the **Sorcerer** roles are combat roles, they may share the same game state space. For this reason, the easiest way to create a **Sorcerer** role is to adapt the **Fighter** role by replacing its behaviours with sorcerer-specific behaviours. In this case, no RL changes are necessary, because the action space is generated automatically by ScriptEase. However, the pattern designer may decide to further edit this role by adding a few new states to the existing state space. For example, the designer may add a binary state indicating whether the sorcerer is currently affected by a defensive spell. In addition, if a new reward function is desired for the new role, the programmer can create a new definition atom in ScriptEase, using `NWScript` code. The current fighter combat reward function is illustrated in Figure 4.15.

Generic RL Roles

We can apply our RL algorithm, ALeRT, to non-combat situations. For example, a guard NPC can decide when to rest, check, or patrol using learning. We plan to develop a more generic collection of *RL* role cues. A guard *RL* role cue will help create a more effective guard NPC. A challenge posed by non-combat situations is the need to define the episode boundaries, i.e., we need to generalize the concept of a round-end. We are considering time-based episodes, such as an hour or a time interval between several *OnHeartbeat* events (approximately six seconds each).

4.6 Integrating RL in ScriptEase

When a creature with a performance spawns in the game, the *OnSpawn* script attached to that NPC initializes the creature in preparation for the behaviours it will

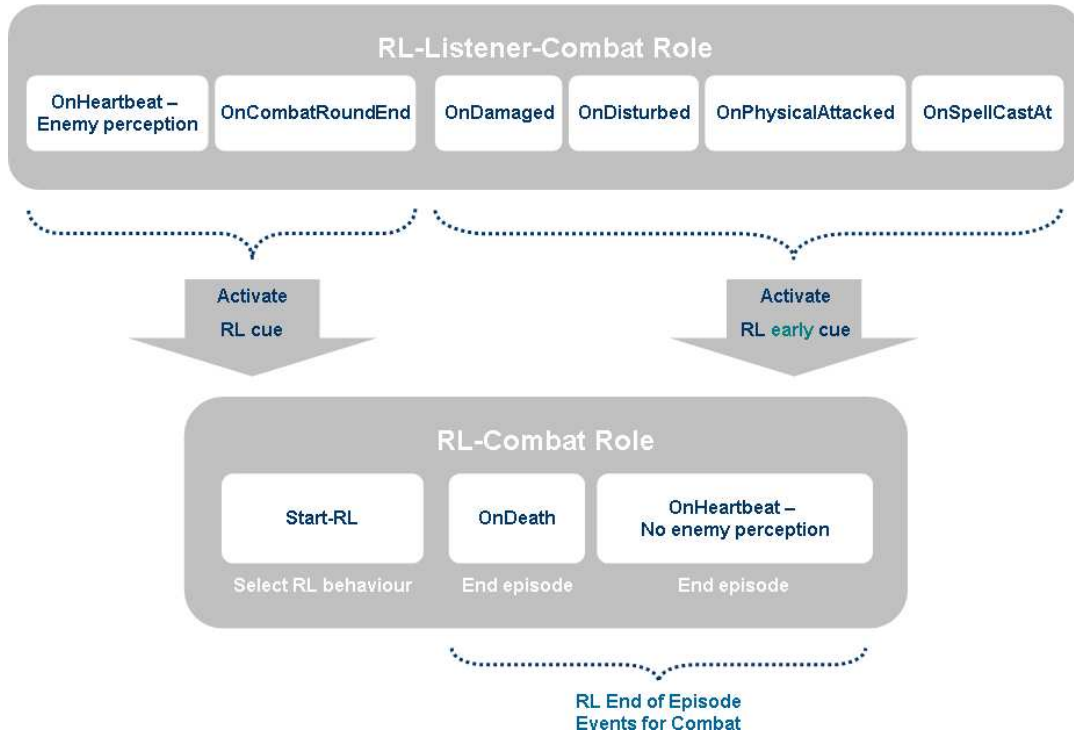


Figure 4.16: The events that activate the *RL* role cue for combat.

perform (i.e., it resets the latent priority and it starts the behaviour event). The behaviour event is a user-defined event fired on the NPC that causes the *OnUserDefined* script to execute. As a result, the performance (e.g., **Wanderer**) starts by activating the default **Start** role cue. Any roles included in a performance whose role cues are triggered become activated and one of these roles is selected randomly. If the selected role has any basic behaviours that can be selected (i.e., their conditions do not prevent them from selection), then the NPC executes one of these behaviours in concordance with their priority scheme.

The latent role cue, *RL*, triggers the beginning of the combat RL (e.g., if the NPC detects a hostile creature on an *OnHeartbeat* event). For example, in the case of a guard NPC, if the guarded item is stolen from a chest, the thief is set to be hostile to the guard NPC. This activates the *RL* role cue. As a second example, the **Wanderer** performance illustrated in Figure 4.17 contains a **Fighter** role activated by an *RL* role cue and a **Wanderer** role activated by the *Start* role cue. When the NPC perceives an enemy, the **Fighter** role becomes active and it selects

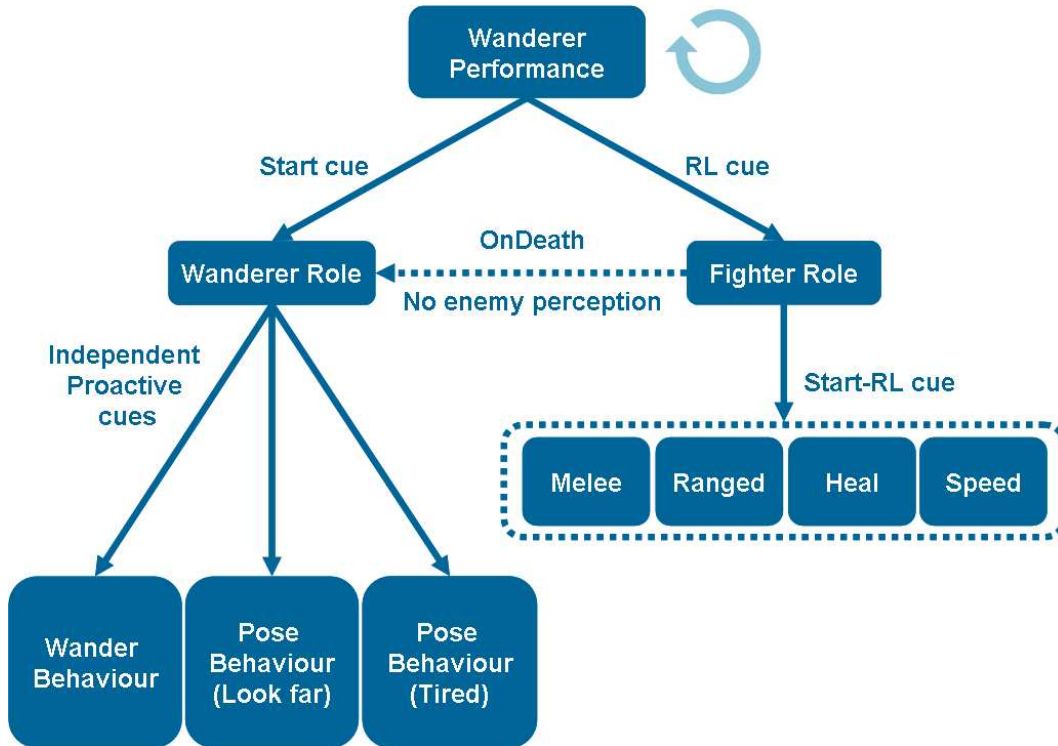


Figure 4.17: The cues used in combat RL for a fighter NPC.

among one of its four behaviours (melee, ranged, speed, and heal) using our learning algorithm. The fight stops after the first combat round if no other role cue or action reactivates it. As long as there is an opponent in sight, the fight goes on (the *OnCombatRoundEnd* event fires to continue the fight). In addition to picking an action only when a combat round occurs, there is one exception in which we trigger an *RL* role cue before a combat round is complete. We need to select an action right away when our NPC is attacked during a round in which it performs non-attack actions (e.g., heal or speed). These non-attack actions are performed faster than attack actions. The default *NWN* scripts call the `DetermineCombatRound` function when the NPC is not engaged in combat but has to start combat when it is attacked by an enemy, by firing one of the following events: *OnPhysicalAttacked*, *OnDamaged*, *OnDisturbed*, and *OnSpellCastAt*. To ensure a fair number of actions for both NPCs, we also activate the *RL* role cue and fire an event that activates the *Start-RL* behaviour cue when these four events occur, by performing the **Activate RL early cue** action atom. For a general RL situation, only the **Activate RL cue**

action atom is necessary to activate the *RL* role cue. The role activated by the *RL* role cue triggers its *Start-RL* behaviour cue that selects a basic behaviour based on learning. The *Start-RL* behaviour cue has an *early* variable that is set to true when the **Activate RL early cue** action atom is performed and it is set to false when the **Activate RL cue** action atom is performed. If the value of the *early* variable is true, the *Start-RL* behaviour cue selects and enqueues a behaviour to perform in the next step (it does not perform that behaviour in the current step). If the *early* variable is false, the *Start-RL* behaviour cue selects, enqueues, and performs a behaviour.

Finally, the fight stops when our learning NPC dies or when the last hostile NPC dies. When our NPC dies, the *OnDeath* event is triggered on our NPC, activating the *Creature death* cue and causing an update of the learning data structures. However, the combat may also end when our NPC ceases to perceive an enemy, i.e., when the *OnHeartbeat - No enemy perceived* event is fired, activating the *No enemy perceived* cue. In this case, the latent queue empties, since the role is no longer active. The behaviour cues that determine the end of an episode (e.g., end of combat in a combat RL scenario) include an action atom that sets the role to the previous role (i.e., the role used before the combat started), as illustrated in Figure 4.17. For example, if all the enemies leave the area, the **End of episode** action atom causes our NPC to return to its interrupted behaviours or to its previous role.

In general, to create a new RL performance, a user needs to provide behaviour cues that start, continue, and end an episode. An RL performance must contain an **RL-Listener-Name** role that consists of several behaviour cues. One such cue (e.g., *Enemy perception*) activates the *RL* role cue that starts the learning process. Another behaviour cue (e.g., *OnCombatRoundEnd*) activates every learning step. The role itself, **RL-Name**, should provide at least a behaviour cue that ends a learning episode (e.g., *Creature death*), so that the role can be restored to a previous role when the learning ends. The extra behaviour cues included for the combat example are specific to combat and are not necessary in a general learning process. In addition, the user needs to insert a *Start-RL* behaviour cue into the role to trigger the basic behaviours that will be learned.

4.6.1 The RL Auxiliary File

The parameters needed for the ALERT algorithm outlined in Figure 4.2 are the following: ϵ the exploration vs. exploitation ratio, which is the ratio between using a random policy to using a predefined policy (a *greedy* policy, in the case of our implementation of Sarsa), α the learning rate, λ the trace decay, and γ the discount rate. The estimator of the value function for action a , Q_a , is initialized with zero and it is represented by the sum of the θ values corresponding to action a for the active states.

In addition to the traditional RL parameter values: ϵ , α , λ , γ , ALERT adds new parameter values: ϵ_{min} , ϵ_{max} , ϵ_{steps} , α_{min} , α_{max} , α_{steps} , δ_{window} , and f , the standard deviation factor used by the trend mechanism. Note that the ϵ and α parameters values are automatically computed from the values of the ϵ_{min} , ϵ_{max} , ϵ_{steps} , and α_{min} , α_{max} , α_{steps} respectively, using formulas represented by ScriptEase definitions.

An author does not have to change or view parameter values. A pattern designer who wishes to tune the traditional RL parameters can assign new values to the definitions found at the *RL* role cue level. For example, if a pattern designer wishes to modify the value of the *lambda* parameter, the designer expands the *RL* role cue and changes the default value that defines the *lambda* parameter to a desired value in the corresponding definition, as illustrated in Figure 4.14. If a pattern designer wishes to change the values of the *epsilon* or *alpha* parameters, regardless of the default formula provided by our algorithm, the designer needs to add definition atoms for the *SE_RL_EPSILON* and *SE_RL_ALPHA* variables respectively to define the new formulas. A programmer may directly modify our RL implementation, for instance to tune the traditional RL parameter values or to change the learning algorithm. In this case, the programmer's modifications will be localized to the `i_se_rl` auxiliary file that contains the RL code. The `i_se_rl` auxiliary file is automatically inserted in any module compiled in ScriptEase. In general, the changes necessary to use our learning algorithm for other situations (combat or non-combat scenarios) are localized to the RL auxiliary file.

4.6.2 Changes to Spronck’s Arena Module

There were several small changes that we made to the combat framework to accommodate our experiments. The most significant changes were the order of spawning of the NPCs, support for dynamically changing the equipment configuration between phases, and destruction delay of our NPC after the opponent NPC died until after our NPC reaches a CRE.

We alternate the creation of the NPCs for every episode, so that no NPC has an unfair advantage by being spawned first and attacking first, possibly with an extra attack of opportunity. In all experiments, we consider an even number of episodes and in each episode we spawn the NPCs alternately.

For a switch in configuration, we created a copy of each fighter NPC with a different blueprint resref and a different inventory configuration (equipment). To integrate this change into Spronck’s code, we created a sign and a lever for each of the new fighters to ensure that learning is used by Spronck’s NPC after the switch in configuration as well. We modified four files in Spronck’s module v.3 to address fairness (the order of NPC spawning) and the switch in configuration (spawning NPCs from a different blueprint resref after episode 500). Changes in the modified files are marked using a bold font and are presented in Appendix C. We have created an additional file, `i_se_rl_modific`, for the implementation of these changes. This file allows for potentially other additions to the system that involve the smooth integration of our algorithm with other learning systems.

The end of an episode occurs when either of the NPCs dies. If our NPC dies and the opponent has not died yet, our data structures are updated when the *OnDeath* event occurs. No combat framework change was necessary in this case. If the other NPC dies and our NPC has not died yet, Spronck’s combat framework destroys our NPC a few seconds later on the *OnHeartbeat* event of a third NPC (Igor, the referee). If no CRE for our NPC has occurred before our NPC’s destruction, the data structures will not be updated. Therefore, we marked our NPC as non-destroyable until after the occurrence of our NPC’s CRE event. We did not have to change Spronck’s framework code to accommodate this change. Instead, setting a non-destroyable flag on our NPC simply changes the effect of the framework code.

4.7 Experiments and Evaluation

We used Spronck’s *NWN* combat module (illustrated in Figure 4.18) to run experiments between two competing agents. Each agent was scripted with one strategy from a set of seven strategies.



Figure 4.18: Two fighters in Spronck’s arena performing a ranged (left) and a melee (right) action, respectively.

- *NWN* is the default *NWN* agent, a rule-based probabilistic strategy that suffers from several flaws. For example, if an agent starts with a sword, it only selects between melee and heal, never from ranged or speed.
- *RL*₀, *RL*₃, and *RL*₅ are traditional Sarsa(λ) dynamic learning agents with $\alpha = 0.1$, $\epsilon = 0.01$, $\gamma = 1$, and with $\lambda = 0$, $\lambda = 0.3$, and $\lambda = 0.5$ respectively.
- *ALeRT* is the agent that uses our new strategy with action-dependent learning rates that vary based on trends, with the parameters initially set to $\alpha = \alpha_{max} = 0.2$, $\epsilon = \epsilon_{max} = 0.02$, $\lambda = 0$ (fixed), $\gamma = 1$, and with $\alpha_{min} = 0.01$, $\alpha_{max} = 0.2$, $\alpha_{steps} = 20$, $\epsilon_{min} = 0.005$, $\epsilon_{max} = 0.02$, $\epsilon_{steps} = 15$, $\delta_{window} = 10$, and $f = 1$, where f is the number of standard deviations that delta-bar must be away from the average delta-bar to identify a significant trend. These values were determined empirically.
- *M1* is Spronck’s dynamic scripting agent (Spronck’s learning method 1), a rule-based strategy inspired by RL, called dynamic scripting.

- OPT is a hand coded optimal strategy that we created based on the available equipment. For example, for the Melee equipment configuration detailed in Table 4.1, the OPT agent drinks a speed potion and then performs melee actions until the end of the episode. We refer to this sequence of actions as S-M*. The optimal action sequence for the Ranged configuration is S-R* (speed followed by repeated ranged actions).

Each experiment consisted of 50 trials and each trial consisted of either one or two phases of 500 episodes. At the start of each phase, the agent was equipped with a specific configuration of equipment. We created the phases by changing each agent’s equipment configuration at the phase boundary. In the first phase, we evaluated how quickly and how well an agent was able to *learn* a winning strategy without prior knowledge. In the second phase, we evaluated how quickly and how well an agent could *adapt* to sudden changes in the environment and discover a new strategy. In essence, the agent must overcome a bias towards one strategy to learn the new strategy. Each equipment configuration (Melee, Ranged, Heal) has an optimal action sequence. For example, the melee weapon in the Melee configuration does much more damage than the ranged weapon, therefore, the optimal strategy uses the melee weapon rather than ranged. The optimal strategy for the Melee configuration is speed followed by repeated melee actions until the episode finishes. Similarly, the Heal configuration has a potion that heals a greater amount of HP than the healing potions in the other three configurations, because it completely restores the agent’s hit points. Therefore, the optimal strategy for the Heal configuration is speed, repeated melee actions until the agent’s hit points are less than half of the initial value, heal, and repeated melee actions until the end of the episode. The optimal action sequence for each configuration is shown in Table 4.1.

Since we could not disable the graphics without affecting the rules of combat and since the duration of a combat round is six seconds, each experiment ran for at least ten hours. In all experiments, the two competing agents had exactly the same equipment configuration and game statistics. However, the agents had different scripts that controlled their behaviours. In the experiments illustrated in Figure 4.19 and Figure 4.20, we ran only a single phase (500 episodes).

Config.	Melee Weapon	Ranged Weapon	Healing Potion	Enhancement Potion	Optimal Strategy
Melee	GS +1	HC +1, B	Cure Serious	Speed	S-M*
Ranged	LS	LB +7, A +5	Cure Light	Speed	S-R*
Heal	LS +3	HC +1, B	Heal	Speed	S-M*-H-M*

Table 4.1: Agent equipment configurations and optimal strategies.

Legend: GS: Great Sword; HC, B: Heavy Crossbow and bolts; LS: Longsword; LB, A: Longbow and arrows; S-M*: speed followed by repeated melee actions; S-R*: speed followed by repeated ranged actions; S-M*-H-M*: speed followed by repeated melee actions, heal, and more melee actions.

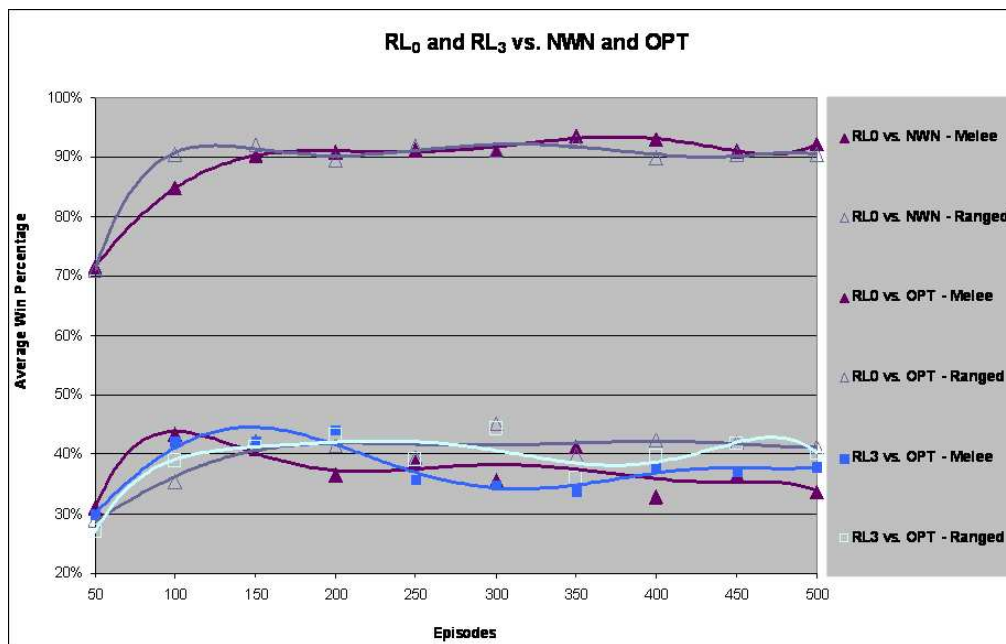


Figure 4.19: RL_0 and RL_3 vs. NWN and OPT.

Each data point in a graph represents an agent’s winning percentage against the opponent after each group of fifty consecutive episodes averaged over all trials. The x-axis indicates the episode group and the y-axis indicates the agent’s average winning percentage during that episode group. For example, the data point at $x=200$ represents the average winning percentage of the group of episodes between episode 151 and episode 200 over all trials for that experiment.

4.7.1 Motivation - ALeRT, M1, and RL vs. Static Opponents

In this set of experiments, we compare the combat results of the ALeRT, M1, and RL (i.e., Sarsa(λ) with different λ values) agents respectively against two static agents: NWN, the default NWN strategy and OPT, the optimal strategy for each configuration.

RL vs. NWN and OPT

Our first set of experiments compares RL vs. NWN and OPT. Originally, we thought that traditional RL could be used for agents in NWN . In fact, RL_0 defeated NWN by a range of 70% to 93%, as shown in Figure 4.19 for both Melee and Ranged configurations. RL agents with other RL parameter values also defeated NWN (not shown in Figure 4.19). NWN performs poorly since if an agent starts with a sword equipped, it only selects from melee and heal, never from ranged or speed actions. However, although we experimented with many different RL parameter values, RL could not converge to the optimal strategy against OPT. For example, Figure 4.19 shows that RL_0 and RL_3 could attain only 34% and 38% wins respectively against OPT with the Melee configuration after 500 episodes, and 41% and 40% wins respectively for Ranged. Experiments with various other parameter values did not yield better results. For example, RL_5 performed better than RL_0 and RL_3 for Melee, but performed worse for Ranged. It was clear that we had to change the Sarsa(λ) algorithm in a more fundamental manner. Therefore, we developed the action-dependent learning rates in the ALeRT algorithm to overcome this limitation.

The results in Figure 4.19 show that RL defeated NWN and Figure 4.20 shows

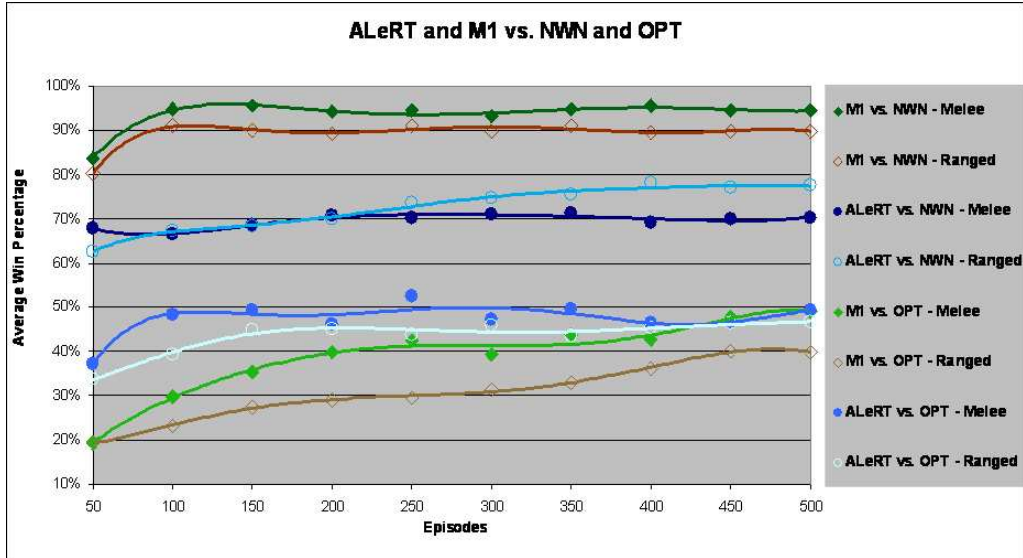


Figure 4.20: ALeRT and M1 vs. NWN and OPT.

that ALeRT defeated NWN. After 500 episodes, ALeRT won 70% for Melee and 78% for Ranged (Figure 4.20), while RL_0 won 92% and 90%, respectively (Figure 4.19). ALeRT could achieve a closer winning rate to RL against the default NWN by tuning the parameters, since ALeRT is a generalization of RL . However, ALeRT converged to the optimal strategies for Melee and Ranged configurations against OPT, while RL did not converge. The action-dependent learning rates in ALeRT are responsible for this convergence. Even though RL performs somewhat better than ALeRT against the default NWN strategy, ALeRT's behaviour is more suitable for a computer game, where it is not necessary (and usually not desirable) for a learning agent to crush either a PC or another NPC agent by a large margin. In fact, ALeRT can be further throttled to reduce its winning percentage in several ways. The simplest solution is to increase ϵ , so that more random actions are selected. The second solution is to change the reward function so that winning is not always rewarded. Essentially, ALeRT should always learn the action with the highest reward, even if it does not always choose to perform that action. Thus, ALeRT will not always defeat its opponent, but if the opponent changes, ALeRT will be able to adapt to the new conditions and still be a worthy adversary. However, it is important for a learning agent to be able to approach the skill level of any agent, even an

optimal one. If the learning agent is the opponent of a PC that has a near optimal strategy, the learning agent should provide a challenge. If the learning agent is a companion of the PC and their opponents are using excellent strategies, the player will be disappointed if the companion agent causes the PC's team to fail.

ALeRT and M1 vs. NWN and OPT

Although *RL* cannot compete with OPT, M1 competes well against OPT, just as ALeRT does. A natural question arises: why use ALeRT instead of M1? The experiments in the next subsections are designed to address this question.

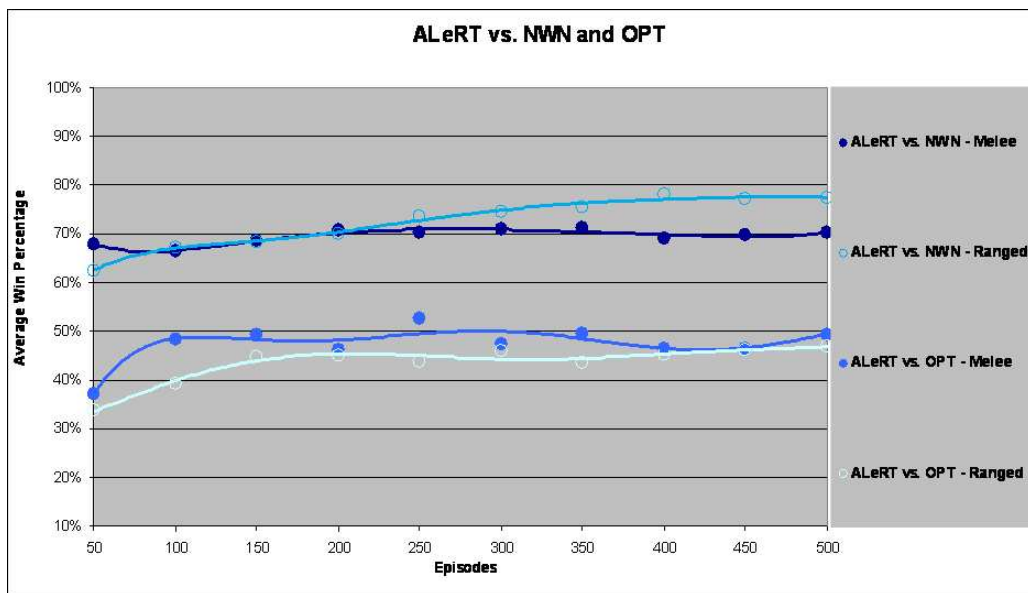


Figure 4.21: ALeRT vs. NWN and OPT.

Our second set of experiments, illustrated in Figure 4.20, compares combat between ALeRT and a static agent to combat between M1 and the same static agent for the Melee and Ranged configurations. For clarity, the two figures, Figure 4.21 and Figure 4.22, each show a subset of the traces illustrated in Figure 4.20 for ALeRT and M1 respectively. In fact, we used two static agents, NWN and OPT. We hand-coded OPT to use the optimal S-M* and S-R* strategies, as shown in Table 4.1.

The upper four traces of Figure 4.20 show the results against NWN. M1 had a higher final winning rate (94% - 47 wins out of 50) than ALeRT (70%) against NWN for the Melee configuration and for the Ranged configuration (90% vs. 78%).

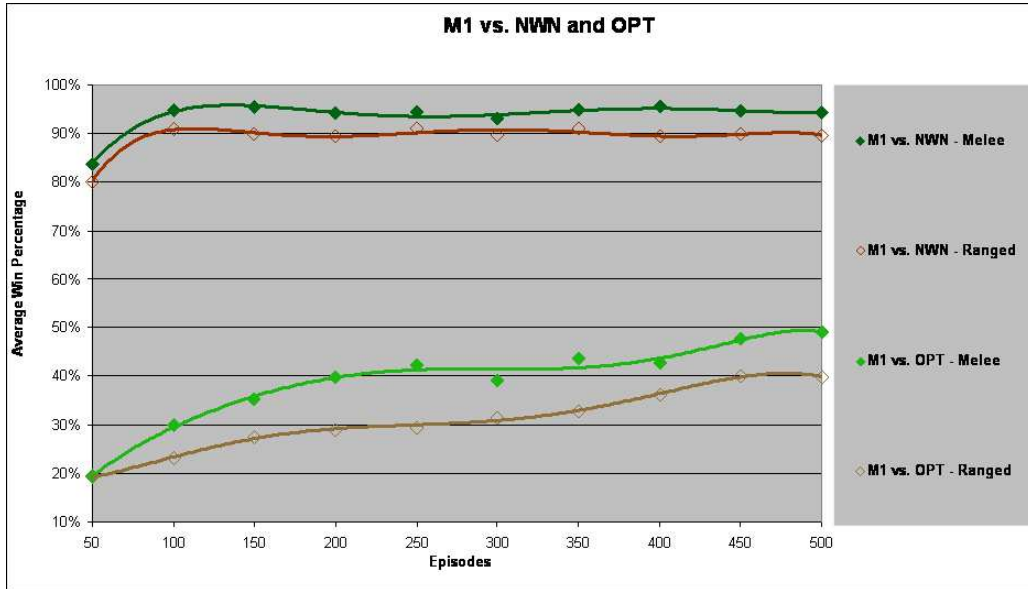


Figure 4.22: M1 vs. NWN and OPT.

These winning rates are more than 20% higher than ALeRT's winning rates. However, as stated before, an NPC should challenge, not defeat the PC, so both ALeRT and M1 could be throttled to have lower winning rates.

The lower four traces of Figure 4.20 show the results against OPT. ALeRT converged to OPT in both configurations, but M1 did not converge to OPT for Ranged by the end of the experiment. M1 converged more slowly than ALeRT for Melee, since the latter won 48% after the first 100 episodes and exceeded 46% after that. M1 won only 30% at episode 100 and did not reach 46% until episode 450 for Melee. For Ranged, ALeRT converged quickly, winning 44% after episode 150 and 46% by episode 450, while M1 achieved its highest win rate (40%) after episode 450. Although a higher winning rate is not necessarily better in computer games, a winning rate (at least 50%) is necessary to ensure a competitive game experience.

The Melee configuration was taken directly from Spronck's module [77], but the Ranged configuration was created for our experiments to test adaptability. It is possible that M1 was tuned for the Melee configuration and re-tuning may allow M1 to converge to the OPT-Ranged strategy. Recall that the *RL* agents outperformed NWN, but they did not converge to OPT (Figure 4.19).

4.7.2 Dynamic Opponents - ALeRT and RL vs. M1

In this set of experiments, we studied RL and ALeRT versus a dynamic opponent, M1. As shown in Figure 4.23, the RL agent cannot defeat the M1 agent, while the ALeRT agent ties the M1 agent.

In Figure 4.23, a score of 50% represents a tie. Each data point represents the average score over 300 trials (100 for each of the three configurations: Melee, Ranged, and Heal). The RL score is consistently less than 50% (42% at episode 500), while the ALeRT score is close to 50%. To be useful, a learning algorithm must be able to at least tie (i.e., a winning rate of 50%) every other strategy. If the opponent's strategy is optimal, then a tie is the best a learning agent can do. Since RL_0 can only attain a sustained winning rate of 42% against M1, it is an inadequate learning algorithm.

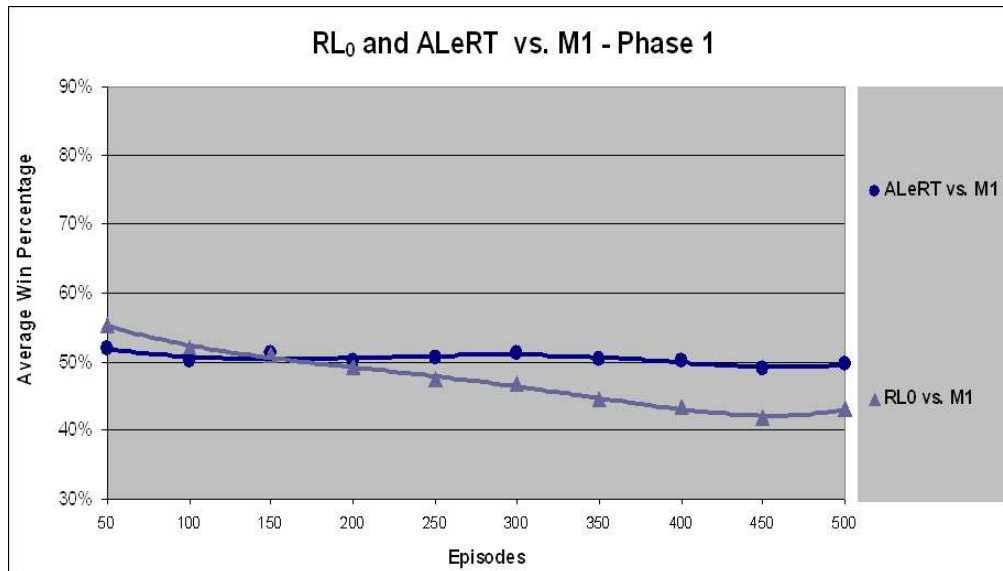


Figure 4.23: RL_0 and ALeRT vs. M1 - Phase 1 (500 episodes).

It was clear that RL (lower trace) was inferior to M1, therefore, at this point in the research project we designed the enhancements to create the ALeRT algorithm (upper trace).

An analysis showed that RL 's poor performance was caused by a long string of losses after the learning rate had decayed to a small value, so rapid adjustment was

impossible. Therefore, we first added a fixed window size to represent consecutive lost episodes. If the number of losses exceeded a threshold (e.g., 15), we restarted our learning algorithm with initial parameter values and Q-values reset to zero. This approach did not prove successful because of the lack of flexibility imposed by the fixed window size. Therefore, we developed the trend approach. As can be seen in Figure 4.23, this approach was successful, since it attained a winning rate of 50% against an optimal M1 strategy.

4.7.3 Adaptability in a Dynamic Environment

In this set of experiments, we measured how fast a learning agent could recover after a change in equipment configuration to test the adaptability of agents in combat.

After 500 consecutive episodes (first phase), we changed the equipment configuration of the two agents and observed 500 more episodes (second phase). Each agent was required to overcome the bias developed over the first phase and learn a different strategy for the second phase. NWN is not adaptive, therefore, we compared ALeRT and RL_0 to M1. We used the following combined configurations: Melee-Heal, Melee-Ranged, Ranged-Melee, Ranged-Heal, Heal-Melee, Heal-Ranged. For example, Melee-Heal represents a two-phase combat (a thousand episodes), in which the agents used the Melee configurations in the first phase (first 500 episodes) and the Heal configuration in the second phase (last 500 episodes). The learning algorithms were not re-initialized between phases so each agent was biased towards the strategy developed in the first phase. In the first phase, we evaluated how quickly an agent was able to learn a winning strategy without prior knowledge. In the second phase, we evaluated how quickly an agent could discover a new winning strategy after an equipment change.

We ran 50 trials for each of the Melee-Ranged, Melee-Heal, Ranged-Melee, Ranged-Heal, Heal-Melee, and Heal-Ranged configurations. The cumulative results over 300 trials are shown in Figure 4.24.

ALeRT adapts faster than M1 to changes in environment (equipment configuration) that affect a strategy's success. ALeRT increased its average win rate to 56%, 50 episodes after the phase change. By episode 1000, ALeRT defeated M1 at an

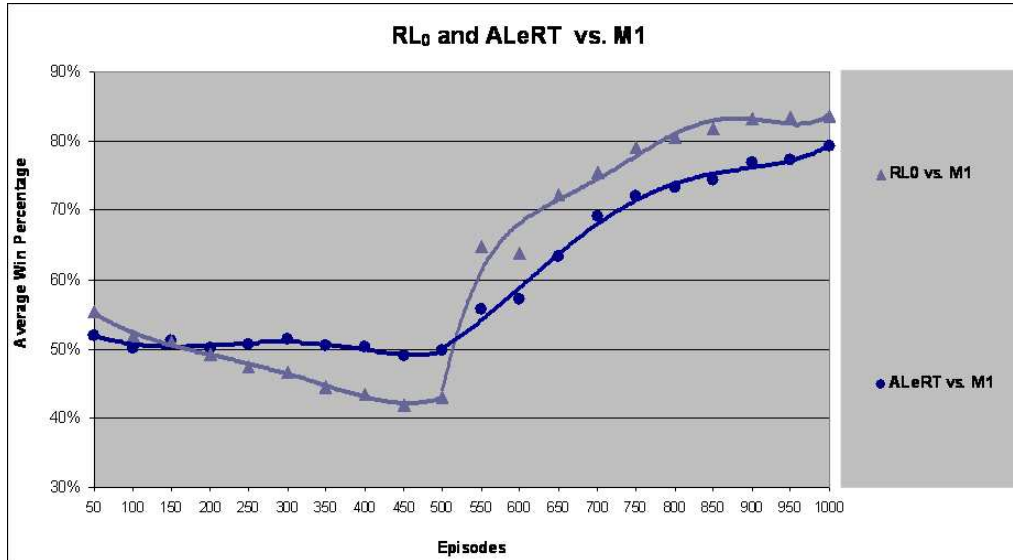


Figure 4.24: RL_0 and ALeRT vs. M1.

average rate of 80%. The features of ALeRT that contribute to this rapid learning are the trend-based step-sizes and the win-based exploration rate modifications to Sarsa(λ). In fact, RL_0 defeated M1 at a slightly higher rate (84%) than ALeRT after the phase change, but RL_0 won only 42% against M1 in the first phase [17]. As stated in the previous section, RL_0 won only 34% and 41% against optimal strategies with Melee and Ranged configurations respectively, which is not an acceptable strategy. Figure 4.24 shows that RL_0 did not find the optimal strategy in the first phase against a dynamic opponent. Therefore, we do not include RL_0 traces for added clarity of the next detailed graphs.

Rather than showing six separate detailed graphs, we combined the common first phase configurations so that the experiments can be shown in three graphs: Melee-Ranged&Heal, Ranged-Melee&Heal, and Heal-Melee&Ranged. The data points from two separate experiments were combined into one trace in the first phase of each graph. Therefore, each data point represents the average winning percentage over 100 trials. Each second phase data point represents the average winning percentage over 50 trials.

Figure 4.25 shows the Melee-Ranged and Melee-Heal results. ALeRT performed almost as well as M1 in the first phase with a 3% deficit at episode 450,

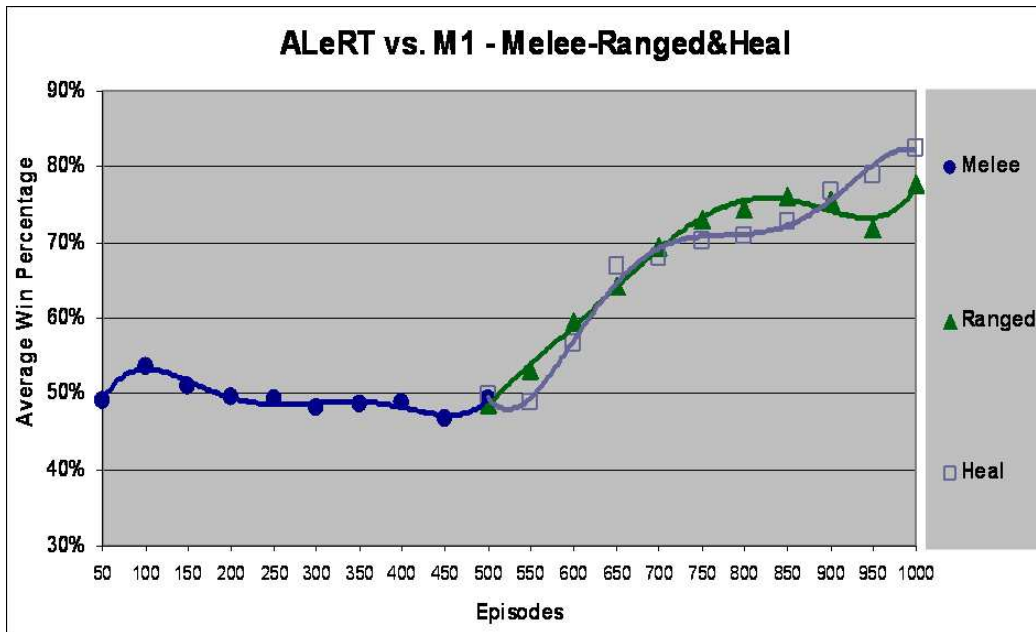


Figure 4.25: ALeRT vs. M1 - Melee-Ranged&Heal.

recovering to a 49.3% win rate at episode 500. M1 performed very well in the initial Melee configuration, perhaps due to manual tuning. Nevertheless, ALeRT's winning rate was very close to 50% throughout the first phase and it dominated M1 during the second phase.

In the Ranged-Melee and Ranged-Heal configurations (Figure 4.26), the agents tied in the first phase, while ALeRT clearly outperformed M1 in the second phase. One of the reasons for the poor performance of M1 is that when it cannot decide what action to choose, it selects an attack with the currently equipped weapon.

In the first phase of the Heal-Melee and Heal-Ranged configurations (Figure 4.27), ALeRT and M1 tied again, but ALeRT outperformed M1 during the second phase. In each configuration, the major advantage of ALeRT over M1 is that ALeRT adapts faster to a change in environment (equipment configuration), even though it does not always find the optimal strategy.

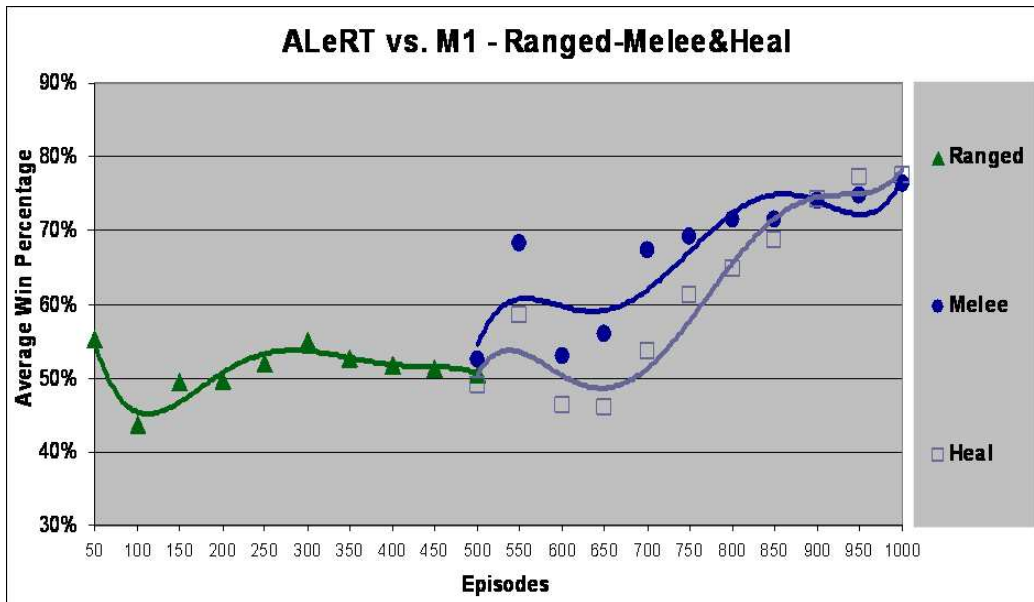


Figure 4.26: ALeRT vs. M1 - Ranged-Melee&Heal.

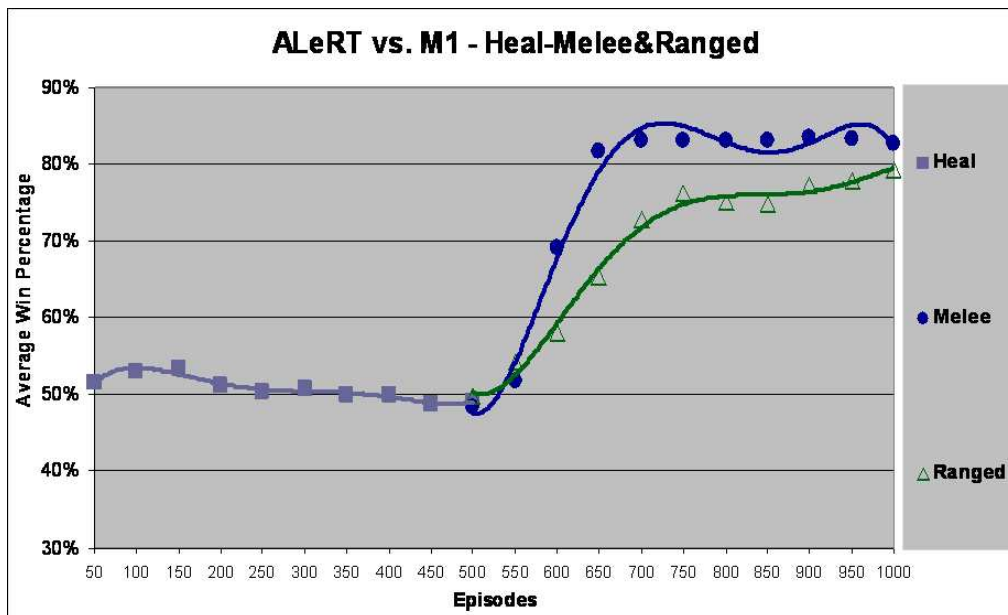


Figure 4.27: ALeRT vs. M1 - Heal-Melee&Ranged.

4.8 Observations

ALeRT is based on Sarsa(λ) and the only domain knowledge it requires is a value function, a set of actions, and a small number of binary states. Unmodified Sarsa(λ) (e.g., RL_0 or RL_1) does not perform well against either an optimal strategy (OPT in Figure 4.19) or against the dynamic reordering rule-based system, M1, in the first phase (Figure 4.23). ALeRT overcomes this limitation, using three fundamental modifications to traditional RL techniques. ALeRT uses (1) action-dependent step-size variation, (2) larger step-size increases during trends, and (3) adjustable exploration rates based on episode outcomes. While conducting our experiments, we made several observations that explain why ALeRT adapts better to change than M1.

ALeRT achieves a good score even when the opponent is not performing optimally and it does not attempt to mimic the opponent. Although ALeRT may not find the optimal solution, it finds a strategy that defeats the opponent at least 50% of the time. In the games domain, this is an advantage, since the agent should not crush the PC.¹

ALeRT works effectively in a variety of situations: short episodes (Melee configuration), long episodes (Ranged configuration), and time-critical action selection situations, such as drinking a speed potion at the start of an episode and a heal potion when the agent's HP are low.

The ALeRT game state is simple (five binary features, including a constant), so each observation is amortized over a small number of relevant states to support fast learning. Although the designer must specify a game state, the "obvious" properties such as health, distance, and potion availability are familiar to designers. Conversely, M1 relies on a set of 21 rules. The discovery and specification of these rules could be challenging.

Although ALeRT may select any valid action during an episode, M1 chooses only one type of attack action (ranged or melee) per episode (it never performs two different attack actions in one episode) in conjunction with the speed or heal

¹In our AIIDE 2008 paper [18], a reviewer of this idea commented: "Both of these are to my view correct (and often under-appreciated when academic AI is applied to games)."

actions. This restriction proved important in the Melee-Heal experiments, where although M1 discovered heal in the second phase, it sometimes selected only ranged attacks and could not switch to melee attacks during the same episode. In some trials, this allowed ALeRT to win even though it did not discover heal in the second phase (i.e., Heal configuration) during that particular trial. Moreover, when M1 cannot decide what action to choose, it selects an attack with the currently equipped weapon (calling the default NWN, if there is no action available). This is a problem if the currently equipped weapon is not the optimal one. Conversely, ALeRT always selects an action based on the value function. If there is a tie, it randomly selects one of the actions with equal value. Therefore, there is no bias to the currently equipped weapon.

ALeRT uses an $\epsilon - greedy$ action selection policy which increases ϵ to generate more exploration when the agent is losing and decreases ϵ when the agent is winning. We experimented with several other $\epsilon - greedy$ strategies, including fixed and decaying ϵ strategies, but they did not adapt as quickly when the configuration changed. We also tried *softmax*, but it generated estimated values of Q with large variations. The result was that the agent could not recover as fast once it selected a detrimental action. M1 uses *softmax* from the Boltzmann (Gibbs) distribution. Most importantly, ALeRT's action-dependent step-sizes provide a mechanism to recover from contiguous blocks of losses. ALeRT's trend-based step-size modification is natural, flexible and robust. In addition to identifying winning trends and converging fast on a new strategy, ALeRT smoothly changes strategies during a losing trend. M1 appears to use a window of ten losses to force a radical change in strategy. This approach is rigid, especially when the problem domain changes and the agent should alter its strategy rapidly.

4.9 Chapter Summary

This chapter presents an enhancement to our behaviour system. We showed how our behaviour model presented in Chapter 3 can be extended to support an additional layer that allows an NPC to select behaviours and to adapt to new situations while

interacting with the environment, based on learning instead of motivations, and without the need for manual scripting on the behalf of the story author. To assess our approach, we devised experiments to evaluate learning rates and adaptability to new situations in a changing game world. Our experimental results show that we outperform both static and dynamic opponents and we tie an optimal opponent.

We introduced a new algorithm, ALeRT, which makes three fundamental modifications to traditional RL techniques. Our algorithm uses action-dependent step-sizes based on the idea that if an agent has not had ample opportunities to try an action, the agent should use a step-size for that action that is different than the step-sizes for the actions that have been used frequently. Each action-dependent step-size should vary throughout the game (following trends), since the agent may encounter situations in which it has to learn a new strategy. Moreover, at the end of an episode, the exploration rate is increased or decreased according to a loss or a win. We implemented our changes that resulted in the ALeRT algorithm using the Sarsa(λ) algorithm. However, our technique is generally applicable to RL algorithms, since it does not rely on any particular assumption. The ALeRT algorithm automatically adjusts the step-size parameters for the actions selected in response to detecting changes in the estimator values. As a result, estimator variance and adaptation times are reduced. This approach is an alternative to traditional techniques of minimizing estimator variability, such as manually tuning the values of individual RL parameters.

We showed that variable action-dependent step-sizes are successful in learning combat actions in a commercial computer game, *NWN*. ALeRT achieved the same performance as M1's dynamic ordering rule-based algorithm when learning started from an initial unlearned strategy. Our empirical evaluation also showed that ALeRT adapts better than M1 when the environment suddenly changes. ALeRT substantially outperformed M1 when learning started from a learned strategy that did not match the current equipment configuration. The ALeRT agent adjusts its behaviour dynamically during the game. Our decision to use combat to evaluate ALeRT was informed by the fact that combat scores provide an objective evaluation criterion. However, RL can be applied to learning any action set, based on a

feature vector and a value function, so we intend to deploy ALeRT for a variety of *NWN* behaviours. ALeRT will be used to improve the quality of individual episodic NPCs and of NPCs that are continuously present in the story.

One of the advantages of using RL in our behaviour patterns is that it enables the author to create NPCs with different behaviours using the same learning algorithm simply by changing parameters. Before a story is released, the author may pre-train NPCs using the general environment for that story. For example, if the PC is intended to start the story at a particular power level, the author uses this power level to train the NPCs. During the game, when an NPC learns a strategy or adapts a strategy, all other NPCs of the same type (e.g., game class, game faction) inherit this strategy and can continue to learn. Each of these vicarious learners jump-starts its learning process using the $\vec{\theta}$ vector generated by the experienced NPC. Ultimately, these improved adaptive behaviours can enhance the appeal of interactive stories, maintaining an elevated player interest.

Chapter 5

Evaluation of the Behaviour Pattern Model

We conducted several experiments to validate our behaviour model. Our goals are to evaluate the usability (Section 5.1) of behaviour patterns by non-programmers through a set of user studies and to assess the efficacy (Section 5.2) of ScriptEase behaviour patterns through a set of case studies using our basic set of behaviours.

5.1 Evaluation from a Usability Perspective

The usability studies we conducted were aimed at investigating whether the *behaviour patterns* we created were simple enough to be used by non-programmers. We accomplished this by studying high-school students' use of behaviour patterns. This user study mirrored previous user studies that determined whether high-school students could use ScriptEase *encounter patterns* to write interactive short stories [11][12][80].

We constructed a pattern catalogue for NPC behaviours containing 27 behaviour patterns commonly used in CRPGs, as described in Appendix B. These behaviour patterns benefit from our previous experience with identifying new encounter patterns to extend the ScriptEase catalogue of encounter patterns [49]. Part of this dissertation research was to develop a way to evaluate encounter patterns. Researchers have developed metrics for the effectiveness of pattern catalogues [16]. We could apply these metrics to assess the usage, coverage, utility, and precision of our behaviour pattern catalogue, but we have not done so yet.

In Chapter 2 (Section 2.2), we defined two types of scalability measures for NPC instances: scalability with respect to the module (across all module areas) and scalability with respect to a single area. The first user study, involving a class of high-school students, tested behaviours in a single area (a medieval castle setting), whereas the second user study, involving one high-school student, tested behaviours across multiple areas. We also addressed the *scalability of behaviours* and the *scalability of use* in these studies.

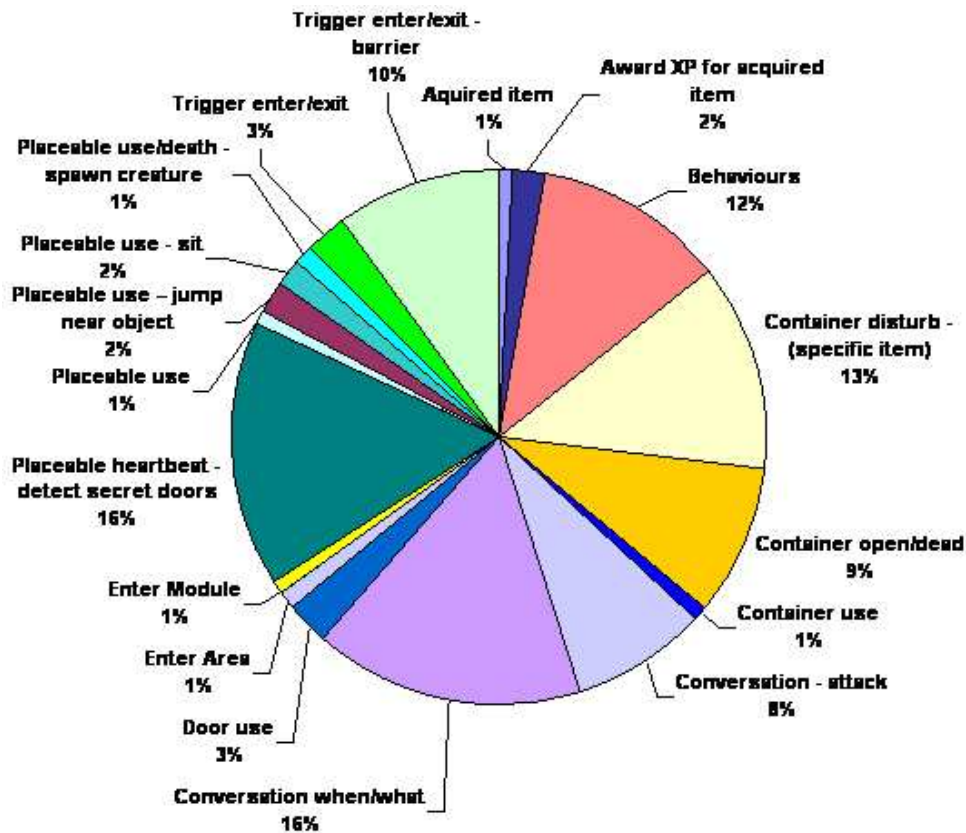


Figure 5.1: Behaviour and encounter patterns used by all 25 high-school students participating in the study. Individual behaviour patterns are grouped into one category.

In the first study, a class of 25 high-school students with no programming experience used ScriptEase behaviour patterns to write an interactive story as part of their Grade 10 English curriculum. This version of ScriptEase contained our library of 27 behaviour patterns, as well as 60 encounter patterns that model interactions between the PC and the objects in the world. We analyzed the behaviour instances

used by the students in their game modules. In the 4.5 hours spent authoring their interactive stories, 9 out of 25 students used behaviour patterns. We did not restrict the type of patterns the students could use and the rest of the students only used encounter patterns. In total, the students used 111 pattern instances, of which 98 were encounter pattern instances and 13 were behaviour pattern instances. The total number of pattern templates was 23, of which 17 were encounter pattern templates and 6 were behaviour pattern templates, as illustrated in Figure 5.1.

Behaviour pattern instances constituted a third (31.6%) of the total number of pattern instances used by the students who included behaviours in their interactive stories, as illustrated in Figure 5.2.

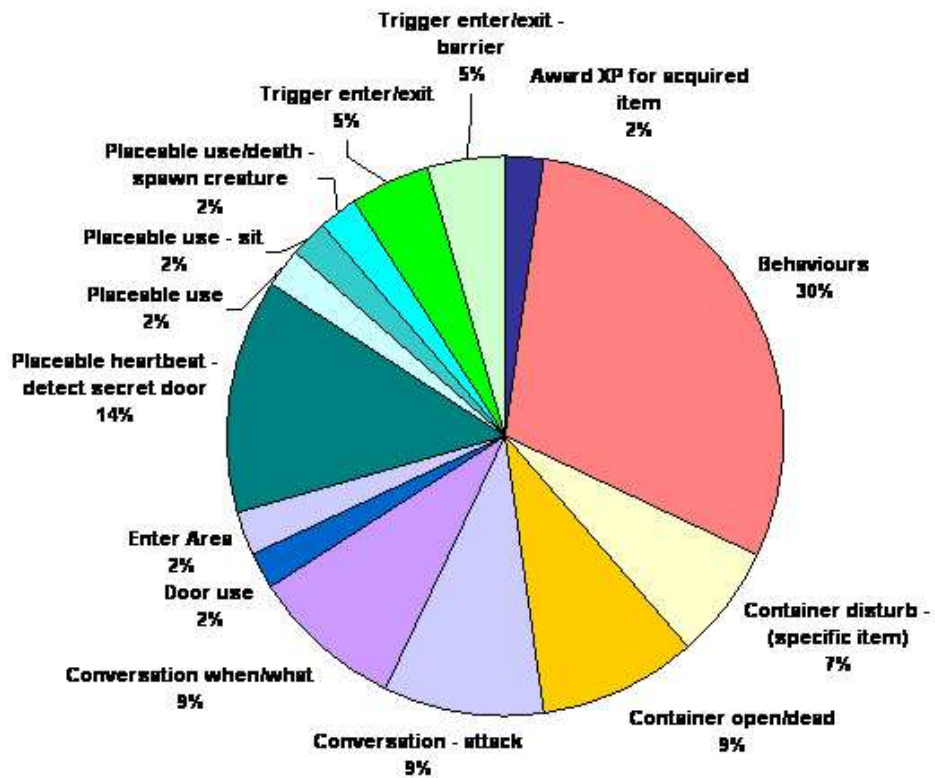


Figure 5.2: Behaviour and encounter patterns used by nine high-school students in their interactive stories.

The most popular encounter patterns (instantiated by all 25 students) were **Placeable heartbeat - detect secret door** and **Conversation when/what**, each instantiated eighteen times, as shown in Figure 5.3.

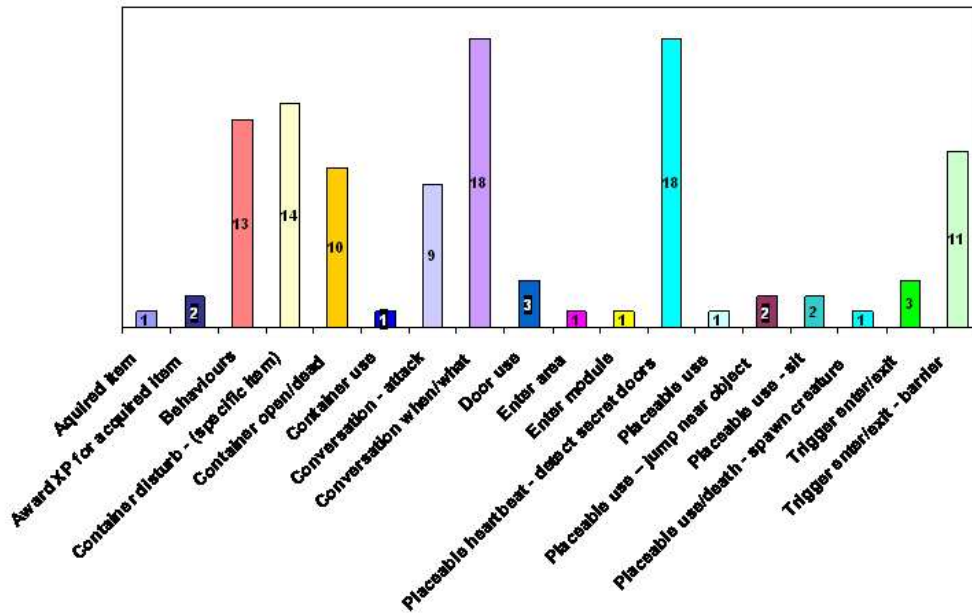


Figure 5.3: Pattern instances (encounter and behaviour) used by all students. Behaviour patterns are grouped into one category.

The most popular behaviour pattern was the **Loiterer** behaviour (the NPC is walking randomly around a fixed point) instantiated four times. Although only nine students used behaviour patterns, these students also selected 13 (using 31 instances of these pattern templates) out of the total 17 encounter pattern templates. Therefore, the students took advantage of pattern reuse and adapted the patterns in different contexts, as shown in Figure 5.3. Moreover, the remaining four encounter patterns that the students who used behaviours did not instantiate were the following: **Acquire item**, **Container use**, **Enter module** (generic patterns), and **Placeable use - jump near object** (specialization of the **Placeable use** pattern, which was already used by students who instantiated behaviours). This indicates that the students understood the difference between the types of patterns in the encounter catalogue and, when they had a choice of similar patterns, they selected the most appropriate (specialized) patterns available to suit their stories.

The students who used behaviours employed 13 instances of 6 different behaviour patterns: **Bystander** (a **Poser** with a **Return** proactive behaviour), **Challenger**, **Follower**, **Loiterer**, **Manipulator**, and **Wanderer**. Besides setting the 13

NPC required options for these instances, 67 other options were set and 15 options were left at default values set by the pattern designer. Students found behaviour patterns easy to use, but were restricted by the time they were allotted to complete the many elements of the story, such as placing objects in the module using the Aurora Toolset or scripting the interactions with game objects using encounter patterns. This user study indicates that behaviour patterns are no more difficult to use than encounter patterns. In addition, the modules that included NPC behaviours ran without any performance degradation. However, to assess all the scalability measures defined in Section 2.2, it is necessary to increase the number of NPCs with behaviours beyond the number found in the student modules.

In the second study, to overcome the time restriction inherent in a classroom, we conducted a more extensive study involving one high-school summer student who did not know how to program. She created an interactive story using 40 pattern instances (*scalability of use*) of 21 behaviour patterns (*scalability of behaviours*) from the same catalogue of 27 behaviour patterns. She also used the ScriptEase *Pattern Designer* tool to create four new behaviour patterns and she adapted the existing behaviours by changing selection values, adding tasks to basic behaviours, and even adding actions to tasks. In total, she created 202 NPCs with ScriptEase generated behaviours in nine areas of her custom *Neverwinter Nights (NWN)* game module. She applied many single pattern instances to multiple NPCs who had the same tag and she organized the behaviour pattern instances in ScriptEase by creating folders for each area. The scene ran flawlessly with no perceptible delays. This exercise shows that our behaviour system is easy to use. Since the behaviours of many NPCs with the same tag can be instantiated at once, the behaviours can be tested quickly. The total number of actors in the entire module (202 NPCs) indicates that our behaviour system can be used without degradation for over 200 NPCs (*scalability of NPC instances*).

5.2 Behaviour Pattern Efficacy

We developed a set of case studies to assess the efficacy of our ScriptEase behaviour patterns. First, we focused on the correctness of the generated behaviours. Second, we assessed the expressiveness of the behaviour patterns and the capability of our behaviour system to support new behaviour patterns. Third, we analyzed the potential for reusability of our existing behaviour patterns.

5.2.1 Correctness - Case Study

We verified that the generated behaviours for three types of NPCs in a tavern area, a common scene in CRPGs, were correctly selected by the NPCs. We used three behaviour patterns that generated all of the scripting code for the tavern scene to illustrate how easy it is to create complex and accurate NPC interactions using behaviour patterns. The background includes patrons, servers and a tavern owner performing their behaviours but, most importantly, interacting with each other. Figure 5.4 shows some of the NPCs in our custom tavern scene module: a tavern owner, two tavern servers, and a patron. The owner asks one of the tavern servers to fetch supplies, the other tavern server approaches the bar to fetch drinks for patrons, and the patron returns from the bar looking for another patron available for a conversation. A set of movies illustrates the proactive, reactive, and latent behaviours that we generated for the tavern NPCs using three behaviour patterns: **Patron**, **Server**, and **Owner** [81].

Although the **Patron**, **Server**, and **Owner** patterns were designed for a tavern scene, they are general enough to be reused for other types of NPC interactions and to generate scripts for other scenes. In addition, the patrons constitute an example of a crowd - a group of NPCs with the same behaviour, but each selecting different behaviours based on local context.

Table 5.1 lists all the behaviours used in the tavern. The first column of the table shows the type of NPC behaviour. This case study was based on ambient (proactive and reactive, not latent) behaviours. Some behaviours are used independently by a single NPC. For example, **Poser** is an independent behaviours. Behaviours that



Figure 5.4: An owner, two tavern servers, and a patron exhibiting ScriptEase-generated behaviours in a tavern scene.

involve more than one NPC are collaborative (joint) behaviours. For example, an **Offer-fetch** behaviour involves two NPCs, one who makes the offer and one who accepts or rejects it. The second column of the table shows the proactive behaviours initiated by the tavern NPCs. The reactive behaviours were included in the sequence of tasks for a proactive behaviour. The letters in parentheses indicate which kind of NPC can initiate the proactive behaviour. For a collaborative behaviour, the kind of collaborator is given as part of the behaviour name, e.g., the **Approach random P** behaviour can be initiated by a server or patron (S, P) and the collaborator is a random patron (P). The third column of Table 5.1 shows the tasks for each proactive basic behaviour. For example, the **Ask-fetch** proactive behaviour generates a set of tasks where the initiator **Speaks** (choosing an appropriate one-liner randomly from a conversation file), the collaborator **Fetches** (goes to the supply room while speaking), the initiator **Receives** the fetched item, and the collaborator **Speaks**. Each task consists of several actions. For example, a **Speak** task consists of facing a partner, pausing, performing a speech animation and uttering the text. The **Converse** proac-

Tavern Behaviours		
Behaviour Type	Proactive Behaviour	Tasks
Independent	Pose (S, P, O)	Pose
	Return (P, O)	Return
	Approach-bar (S, P)	Approach
	Greet nearest P (P)	Speak
	Fetch (O)	Fetch
Collaborative	Approach random P (S, P)	Approach
	Converse with nearest P (P)	Speak, Listen, Listen, Speak
	Ask-fetch nearest S (P, O)	Speak, Fetch, Receive, Speak
	Ask-give O (P)	Speak, Give, Receive, Speak
	Offer-give to nearest P (O)	Speak, Decide, Ask-give (<i>accept</i>)
		Speak, Decide, Speak (<i>reject</i>)
	Offer-fetch to nearest P (S)	Speak, Decide, Ask-fetch (<i>accept</i>)
Speak, Decide, Speak (<i>reject</i>)		

Table 5.1: Tavern behaviours for three types of NPCs: a tavern server (S), a tavern patron (P), and a tavern owner (O).

tive behaviour starts a set of tasks with one or more **Speak** tasks alternating between two NPCs. The **Offer-give** (owner offers a drink) and **Offer-fetch** (server offers to fetch a drink) proactive behaviours each have two different sets of tasks (shown in Table 5.1) depending on whether the collaborator decides to accept or reject the offer.

The behaviour patterns are easy to use. For example, a high-school student created and tested an *NWN* tavern scene in less than half an hour, whereas it may take several days to write the code manually. The simplicity of the process hides the fact that a large amount of scripting code is generated to model complex collaborative interactive behaviours. In fact, 889 lines of NWScript code were generated for the server, while 1087 and 886 lines were generated for the patron and owner respectively. The process of using or adapting a behaviour pattern is simple, since the author only selects options and navigates through menus. ScriptEase automatically generates a large amount of scripting code to implement complex interactive behaviours. For a tavern scene, the story author opens the tavern module in ScriptEase and instantiates each of the three behaviours: **Patron**, **Server**, and **Owner** by selecting the patterns from a menu and then binding each instance to an appropriate NPC and the options of each pattern instance to game objects and/or values. When the *Save and Compile* menu command is selected, ScriptEase generates 3,107 lines

of NWScript code (for the entire tavern scene) that could be edited in the Aurora Toolset, if desired.

The generated code is efficient, producing behaviours that are crisp and responsive, with no perceptible effect on response time for PC movement and actions. The NPCs interact with each other flawlessly with natural movements at more than 30 frames per second. A scene with eighteen patrons, two servers, and an owner was left to play for hours without any deadlock, degradation in performance, repetition or indefinite postponement of behaviours for any actor. The NPCs displayed all their behaviours and selected one of their appropriate behaviours in every situation.

5.2.2 Expressiveness - Case Study

At the beginning of this research, we evaluated the expressiveness of the ScriptEase *encounter patterns*. We identified the implicit patterns that exist in the different modules of the original BioWare Corp. produced *NWN* story for placeable objects [49], with the goal of using the patterns to generate replacement code for the hand-written scripts. During this process, we determined which of the CRPG patterns that we constructed before the case study could be reused and which new patterns were necessary. From this experiment, we concluded that there is significant pattern reuse among successive modules and, consequently, a reduced number of new patterns are required as the pattern base grows. *NWN* was released with a single adventure that consisted of seven modules: *Prelude*, *Chapter One*, *Chapter One Finale*, *Chapter Two*, *Luskan and Host Tower*, *Chapter Three*, and *Chapter Four*. Table 5.2 summarizes the information extracted from the analysis of these modules with regards to placeable objects: the number of calls made to scripts by placeable objects, the number of unique scripts that were referred to by placeable script calls, and the total number of non-comment lines of hand-written code that were contained in the placeable scripts for each module. Finally, the number of pattern instances that were used to replace all of these script calls and the number of patterns that were used in each module are included in the table. However, the total for this column is not the sum of the column entries since many of the patterns were reused across modules. The total (24) is the number of unique patterns used across

all modules. The total for the script templates column is the sum of the column entries, since none of the hand-written scripts was reused across modules. Note that multiple script templates were often replaced by a single pattern with multiple situations. As the number of ScriptEase patterns grows, fewer new ones need

Chapter	Script Calls	Script Templates	Lines of Code	Pattern Instances	Pattern Templates
One*	153	47	391	108	15
Two	112	40	279	104	14
Luskan	54	28	454	51	10
Three	127	50	669	118	15
Four	51	17	132	50	7
Total	497	182	1925	431	24

Table 5.2: ScriptEase encounter pattern statistics. *Chapter One** consists of the *Prelude*, *Chapter One Finale*, and *Chapter One* modules.

to be created to construct new modules. Figure 5.5 shows the number of patterns required to generate the scripts for placeables in successive modules. The first bar shows that *Chapter One** reused 6 existing patterns and required 9 new ones. The second bar shows that *Chapter Two* reused 9 existing patterns, reused 4 patterns created for *Chapter One** and required one new pattern. In general, the percentage of new patterns required for each successive chapter tended to decrease as the case study progressed (60%, 7%, 10%, 20%, 0%) with the exception of *Chapter Three*, which was the largest module. Of the three new patterns created for *Chapter Three*, only one was reused in *Chapter Four*.

To determine the range of CRPG behaviours that can be accommodated by patterns, we conducted a similar case study for the *Prelude* of the *NWN* official campaign, directed at both independent and collaborative behaviours [19]. This study did not include latent or motivational behaviours. The original code used ad-hoc scripts to simulate collaborative behaviours. We removed all of the manually scripted NPC behaviours and replaced them with behaviours generated from encounter patterns. During this process, we identified six new behaviour patterns: **Poser**, **Bystander**, **Exclaimer**, **Duet**, **Striker**, and **Expert**. These patterns were sufficient to generate all of the NPC behaviour scripts.

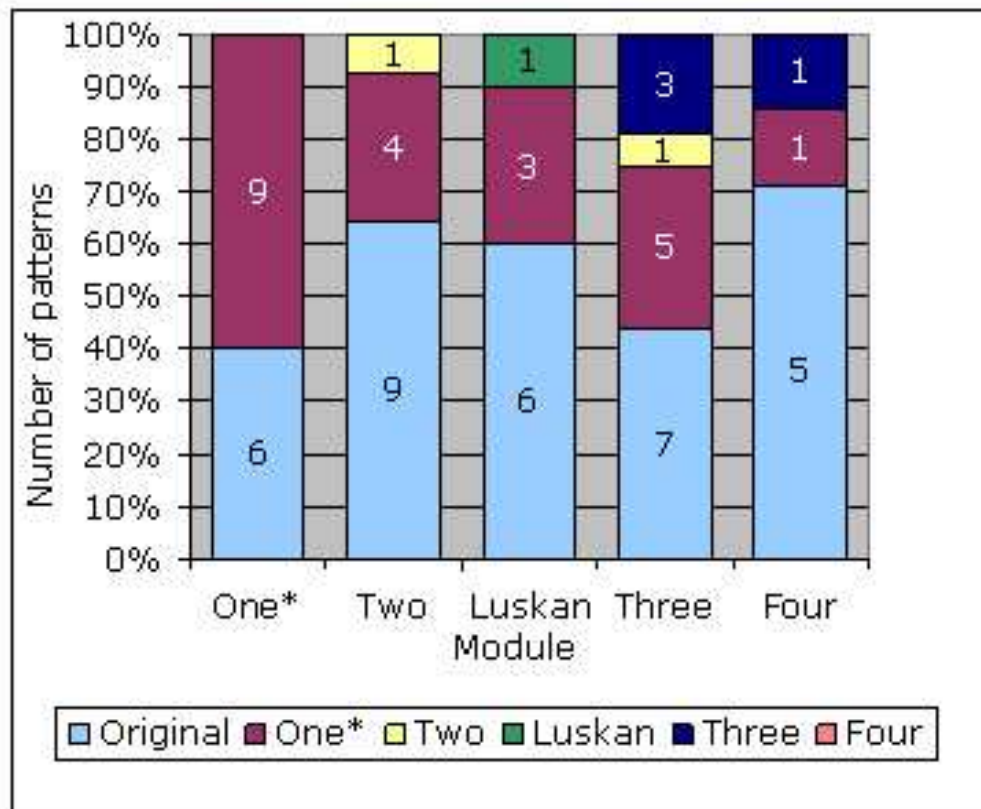


Figure 5.5: Encounter pattern reuse by module in the *NWN* campaign.

The behaviour patterns in this study were implemented [20] on top of encounter patterns to provide preliminary results without having to implement behaviour patterns directly. The five simple patterns and one meta-pattern we identified were sufficient to generate all of the NPC behaviour scripts in the *Prelude* for 48 NPCs. In the original *Prelude*, 39 of these 48 NPCs had scripts attached to them. We replaced 265 lines of manually-written scripting code in 25 files called 73 times for all the 39 NPCs of the original scripted *Prelude*. For the other unscripted 6 NPCs in the original *Prelude*, we attached a **Poser** pattern (the NPC performs a simple animation and optionally utters a random text) to each of them. This pattern provides default initial values for the specific animation and the duration of the animation. For example, we attributed a **Poser** pattern to a silent *injured man* NPC whose behaviour was an animation to beg for help and to a *door guard* who was not scripted in the original *Prelude*. We improved both their behaviours by allowing them to also randomly utter a sentence from a conversation file. In addition, we discovered that the proactive independent behaviours in the original *Prelude* are not always able to recover if the NPC is jostled during the game. For example, a trainee thief who faces a target (combat dummy) and performs a “pickpocket” skill will not be able to return to the combat dummy if the PC clicks on the NPC to initiate a conversation. Instead, the thief picks the pocket of an empty space instead of the combat dummy after the conversation with the PC ends. To solve this problem, we re-generated the behaviour of this NPC from our proactive independent **Expert** behaviour. Our behaviour includes an **Expert** proactive behaviour that consists of a single **Expert** task. The first two actions in this task are the following:

1. walk to the combat dummy, and
2. face the combat dummy, so that the NPC “pickpockets” the appropriate target (the combat dummy).

The **Expert** pattern generates a robust behaviour, since the **Expert** task provides a self-sufficient rational behaviour block, as discussed in Section 3.4.8. The **Expert** behaviour has an additional **Return** proactive behaviour which causes the NPC to return to the original location from time to time.

A **Duet** is a meta-pattern, a more complex behaviour pattern that expresses collaboration between two NPCs. This pattern allows story authors and pattern designers to create a series of collaborative patterns by combining independent patterns. We identified three types of potentially collaborative behaviours in the original *Prelude* and we abstracted them using the **Duet** pattern. First, six NPCs grouped in pairs mimic a conversation by facing each other and performing independent speaking gestures. We replaced the manual scripting code that controls these six NPCs by code automatically generated from instances of the **Duet-Converser-Converser** behaviour pattern. This pattern constitutes a true collaborative behaviour involving two converse behaviours that alternate for the two NPCs so that the conversation between them seems natural. An NPC waits for the collaborator's reply before it provides a response or initiates a new collaboration. Figure 5.6 shows the generated **Duet-Converser-Converser** NPCs.



Figure 5.6: Generated behaviours in the *Prelude*: **Duet-Converser-Converser**.

Second, a pair of spellcaster NPCs constitutes another example of simulated collaboration in the original *Prelude*. Two NPCs successively cast spells on a com-

bat dummy. This is accomplished by applying a delay to one of the NPCs, so that their actions appear to alternate. We also replaced the manual scripting code that controls these NPCs by code automatically generated from an instance of the **Duet-Spellcaster-Spellcaster** collaborative behaviour pattern. The behaviours of the two spellcasters alternate so that the collaboration seems natural: the first NPC casts a spell while the second NPC waits and, only when the first NPC finishes the behaviour, they reverse their roles. In addition to casting spells successively, our NPCs are also engaging in a conversation.

Third, a pair of NPCs is performing another type of training: one NPC spawns a skeleton and the other destroys the spawned skeleton. Figure 5.7 shows the replacement scene as generated from a **Duet-Spawner-Destroyer** behaviour pattern. In the original code, a different ad-hoc technique was used to compensate for not having true collaboration support: one NPC spawns skeletons and the other destroys any perceived skeletons without employing any delay mechanism. This does not constitute true collaboration and the various techniques used to compensate for a lack of support for true collaboration make the scripts hard to understand and maintain. Figure 5.8 shows the manually written NWScript code for the spawner and destroyer NPCs.

When the spawner NPC (*Ansel*) perceives the PC, it executes the script code attached to the *OnPerception* event, firing a user-defined event on itself. This causes the code attached to the *OnUserDefined* event to be executed. As a result, *Ansel* spawns a skeleton with tag “M1Q0BSUM_SK” at a waypoint “WP_Skeleton” and casts a fake spell at this waypoint. Then, *Ansel* fires the same user-defined event with a 30 seconds delay so that a new skeleton is spawned. The destroyer NPC (*Tabitha*) casts a spell that destroys any creature perceived with the racial type undead that is different from a skeleton with tag “M0Q0.SKELETON” already located in the room. The intent of the story author is to simulate a collaborative spellcaster training of these two NPCs. However, if the destroyer NPC takes more time to destroy the spawned skeleton, then the spawner can create another skeleton, since the spawner generates skeletons every 30 seconds, regardless of the destroyer’s actions. Their collaboration is achieved not by communication between the NPCs,



Figure 5.7: Generated behaviours in the *Prelude*: **Duet-Spawner-Destroyer**.

NWScript code for *Ansel* (the spawner):*OnPerception:*

```
void main()
{
    if (GetIsPC(GetLastPerceived()) &&
        GetLastPerceptionSeen())
    {
        SignalEvent(OBJECT_SELF,
                    EventUserDefined(0));
    }
}
```

OnUserDefined:

```
void main()
{
    if (IsInConversation(OBJECT_SELF) == FALSE &&
        GetIsDead(OBJECT_SELF) == FALSE)
    {
        location lLoc = GetLocation(
            GetNearestObjectByTag("WP_Skeleton"));
        ActionCastFakeSpellAtLocation(
            SPELL_ANIMATE_DEAD, lLoc);
        CreateObject(OBJECT_TYPE_CREATURE,
                    "M1Q0BSUM_SK", lLoc);
    }
    DelayCommand(30.0, SignalEvent(OBJECT_SELF,
                                    EventUserDefined(0)));
}
```

NWScript code for *Tabitha* (the destroyer):*OnPerception:*

```
void main()
{
    object oPerceived = GetLastPerceived();
    if (GetRacialType(oPerceived) ==
        RACIAL_TYPE_UNDEAD &&
        GetLastPerceptionSeen() &&
        GetTag(oPerceived) != "M0Q0_SKELETON" &&
        IsInConversation(OBJECT_SELF) == FALSE)
    {
        ActionCastSpellAtObject(
            SPELLABILITY_TURN_UNDEAD,
            oPerceived, METAMAGIC_ANY, TRUE);
    }
}
```

Figure 5.8: Manually written NWScript code for the spawner and destroyer NPCs.

but through the common object of their training: the skeleton. This does not reflect a true NPC collaboration.

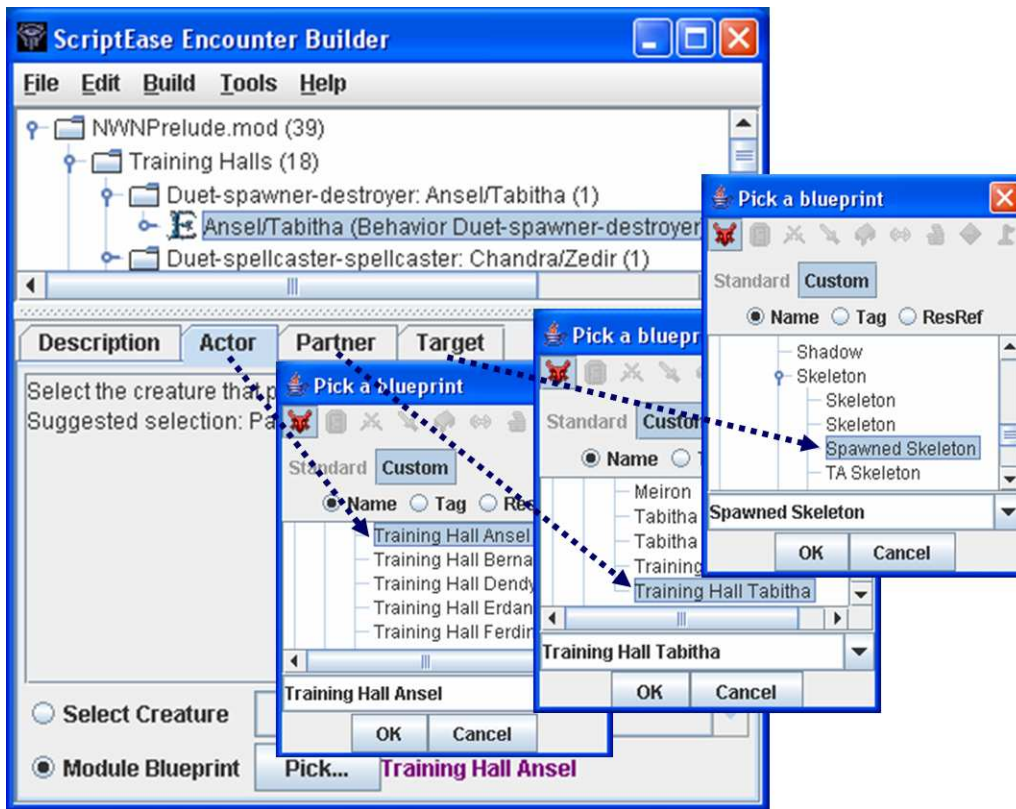


Figure 5.9: An instance of the **Duet-Spawner-Destroyer** pattern in the *NWN Prelude*.

The **Duet-Spawner-Destroyer** pattern generates true collaborative scripts that ensure synchronization between the NPCs. The second NPC’s destroy behaviour does not start until the completion of the first NPC’s spawn behaviour. The complex hand-written code shown in Figure 5.8 can be contrasted with Figure 5.9 that shows how a **Duet-Spawner-Destroyer** pattern can be used to generate code by simply instantiating the pattern and selecting three options: the actor (*Ansel*), the partner (*Tabitha*) and the target creature to be spawned-destroyed (*Spawned Skeleton*). Figure 5.10 shows the **Spawner** and **Destroyer** performances together with their corresponding roles using ScriptEase behaviour patterns. The **Spawner** role contains a proactive collaborative **Spawn** behaviour on the topic “spawner-destroyer” and the **Destroyer** role contains a reactive **Destroy** behaviour on the same topic.

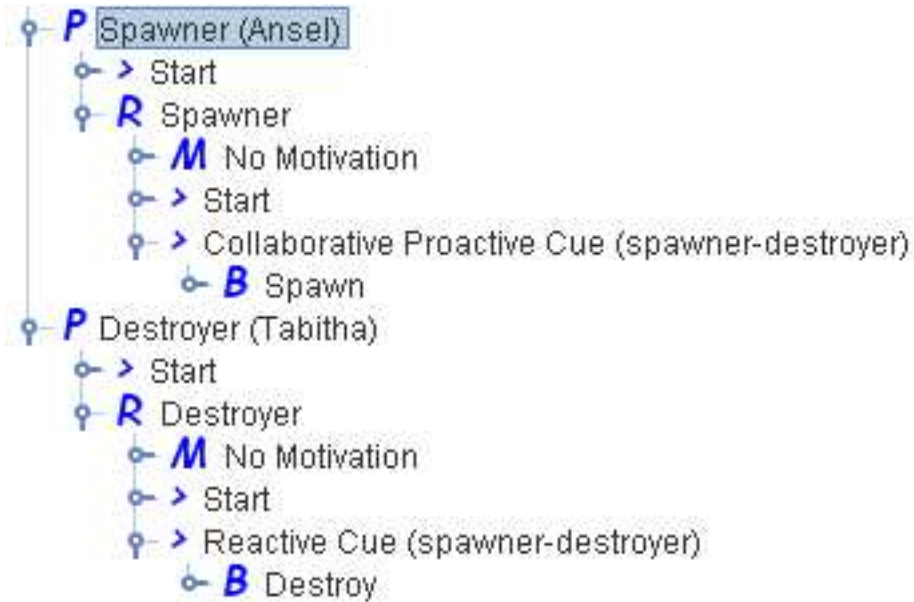


Figure 5.10: The **Spawner** and **Destroyer** pattern instances in the *NWN Prelude*.

Figure 5.11 shows the number of instances of each kind of behaviour pattern that were used in each of the five areas in the *Prelude* chapter. Note that the 32 behaviour pattern instances in Figure 5.11 are applied to 48 NPCs. For example, each instance of the **Duet** meta-pattern involves two NPCs. Moreover, only one **Poser** pattern instance generates the proactive behaviours of nine goblin NPCs that share a common tag. This study shows how behaviours can be inserted into BioWare Corp.'s *NWN* game, improving the overall game experience. Behaviour patterns are used with ScriptEase to easily and quickly re-generate all of the background behaviours of NPCs in the *NWN Prelude* and improve them. This shows how patterns hide the level of complexity necessary to create a realistic interactive story and how patterns can be reused for several NPCs.

To validate the expressiveness of our *behaviour patterns* as first-class objects in ScriptEase, we replaced all the manual scripts for the NPCs of the *Prelude* module with the code generated automatically from our *behaviour patterns* in less than an hour. To increase expressiveness, we have also added simple default roles, such as **Poser** (the NPC performs an animation and speaks text from a conversation file) and **Bystander** (the NPC poses and returns to the NPC's original location), to six

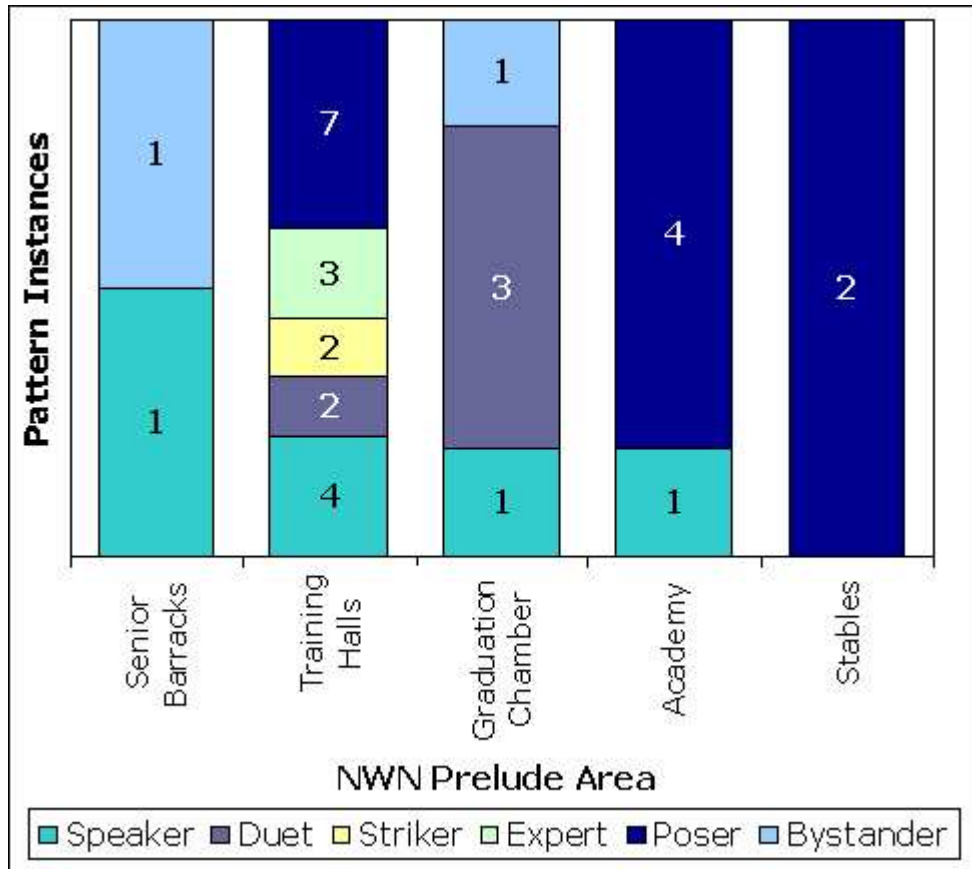


Figure 5.11: Using ScriptEase encounter patterns to generate behaviour scripts in the *NWN Prelude*.

previously unscripted NPCs in the *Prelude*. We built this case study using our prior experience with re-generating the original NPC behaviours in the *Prelude* using encounter patterns that simulated behaviours. Since then, we created new motivational behaviours using behaviour patterns. For example, we created a **Challenger** behaviour pattern, in which the NPC walks to the PC (or another NPC) and starts a conversation, to solve a problem that affected an NPC's behaviour discussed in Chapter 2. Recall that, in the original *Prelude*, an NPC located in the centre of a trigger approached the PC and started a conversation when the PC entered that trigger. The **Challenger** behaviour does not use any other game objects except for the NPC and the conversation file. Since the **Challenger** includes a **Return** proactive behaviour, the NPC returns to the original location without using a trigger or a waypoint as did the original NPC behaviour. Our system records the original location of any NPC with behaviours and the **Return** proactive behaviour uses this information to restore an NPC's facing and location. In addition, our system does not use an *OnPerception* event to trigger behaviours, avoiding the drawbacks associated with this event discussed in Chapter 2. We re-generated behaviour scripts from *behaviour patterns* for all the NPCs in the *NWN Prelude*. This study shows that the new behaviour pattern approach is efficient, robust, and easy to use. The study also validates our ScriptEase behaviour patterns for a real commercial computer role-playing game using real adventure modules.

The **Guard** behaviour pattern described in Chapter 3 generates proactive, reactive, and latent behaviour scripts, resulting in a more engaging guard NPC that is perceptive to the actions of both the PC and other NPCs. A set of movies illustrates this guard's proactive, reactive, and latent behaviours generated for the guard's different roles [35]. The **Guard** pattern generates 1065 non-comment lines of NWScript code, which is a significant amount of scripting code for the amount of work required by an author to use this pattern. The additional work performed in creating this pattern is amortized over the many instances of the **Guard** pattern that occur in this and other game stories.

Further evidence of the generality of behaviour patterns will require a case study that replaces behaviours in other game genres as well. For example, a soccer or

hockey goaltender could be provided with entertaining behaviours to exhibit when the ball (puck) is in the other end of play, such as standing on one leg, stretching, leaning against a goal post, or trying to quiet the crowd with a gesture. One of the criticisms for *EA FIFA 04* was directed to the goalie's behaviour [25] and was addressed in *EA FIFA 06* [26]. However, a study that involves other game genres is beyond the scope of this research.

5.2.3 Inheritance of Behaviour Patterns - Case Study

We have designed our behaviours with a few considerations in mind. Every role that we have in our library of behaviours descends from a generic behaviour. For example, a **Guard** is a **Poser**.

We conducted a preliminary case study that included the introduction of the concept of a meta-pattern, an analysis and graphical representation of pattern usage, an analysis of the number of lines of code, and a description of the inheritance model employed to foster code reuse. We based this study on the set of behaviours implemented using *encounter patterns* rather than the latest set of behaviour patterns that are implemented natively in ScriptEase.

The behaviour patterns form an inheritance hierarchy as shown in Figure 5.12. This hierarchy makes it easier for an author to understand the intents of the behaviour patterns and to utilize them in stories. For example, a **Poser** is an NPC who plays an animation and occasionally utters a random text. The **Poser** pattern is the simplest behaviour, consisting of one proactive behaviour, **Pose**. A **Bystander** is a **Poser** who can also return to the original scene location. This may be necessary if the **Bystander** collides with a game object, the PC, or another NPC, and it moves too far from the intended location. The **Bystander** pattern inherits the proactive **Pose** behaviour and adds one other proactive behaviour, **Return**. An **Exclaimer** is a **Bystander** who can also face a nearby creature and speak. The **Exclaimer** pattern adds a third proactive behaviour, **Exclaim**. Each of these proactive behaviours consists of a single task (**Pose**, **Return**, and **Exclaim** respectively). An **Expert** is a **Bystander** who can perform a skill on a nearby object. A **Striker** is a **Bystander** who can strike a nearby object. In the *NWN Prelude*, an **Expert** thief tries

to pick the pocket of a nearby practice dummy and a **Striker** attacks a nearby combat dummy. The **Expert** and **Striker** patterns inherit the two proactive behaviours from **Bystander**, but add alternate third simple proactive behaviours, **Perform** and **Strike** respectively.

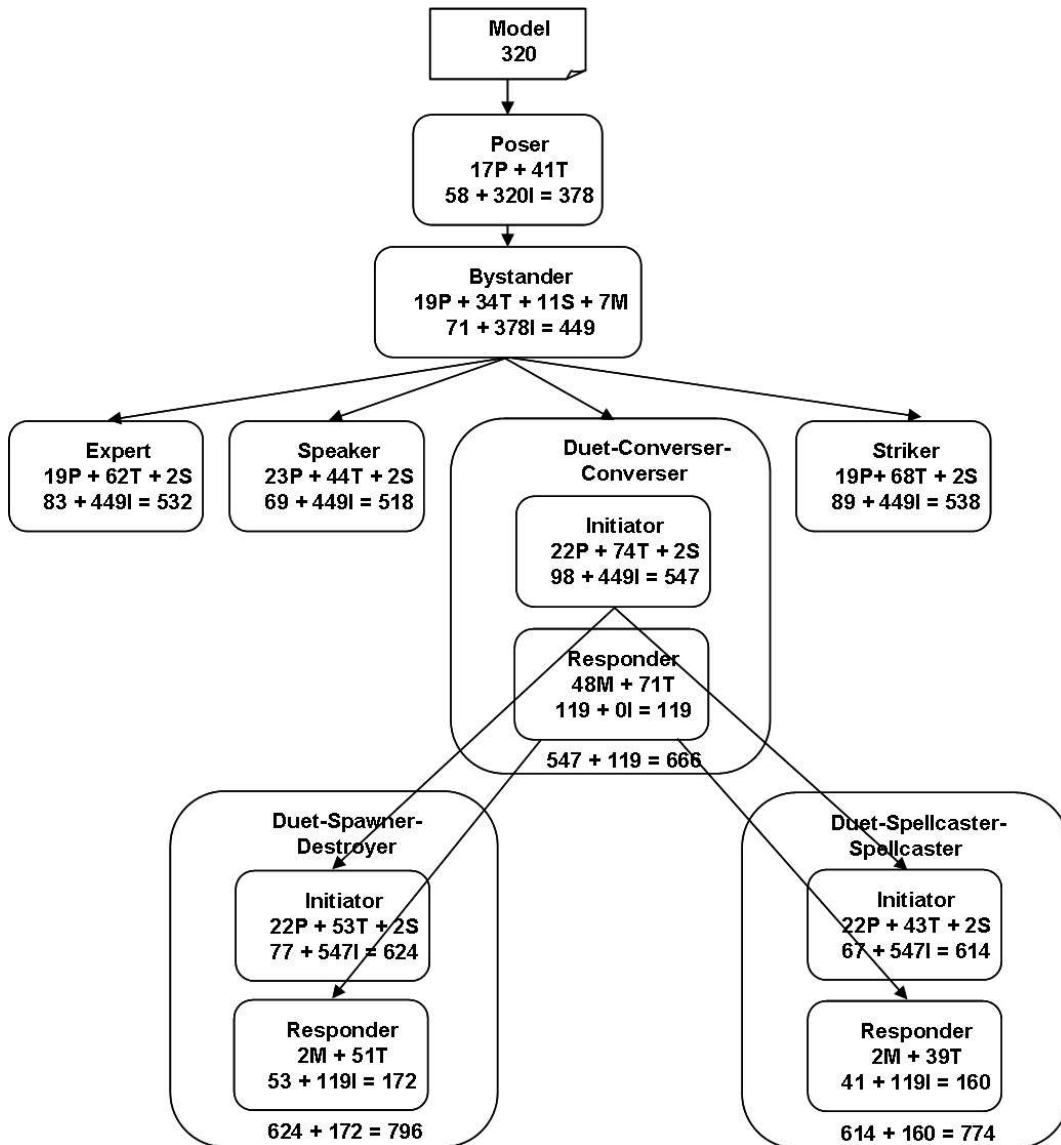


Figure 5.12: The inheritance hierarchy of ScriptEase behaviours using encounter patterns.

The **Duet meta-patterns**, the precursors of collaborative behaviours in the current ScriptEase behaviour pattern catalogue, contain behaviours for two NPCs. The initiator NPC of the **Duet-Convener-Convener** pattern inherits two proac-

tive behaviours from the **Bystander** pattern and adds a third proactive behaviour, **Converse-Converse**, in which the initiator NPC engages a nearby responder NPC in a conversation. The spawner of the **Duet-Spawner-Destroyer** pattern and the initiator of the **Duet-Spellcaster-Spellcaster** pattern each inherits the three proactive behaviours from the initiator of the **Duet-Converser-Converser** pattern and adds a fourth proactive behaviour, **Spawn-Destroy** or **Spellcast-Spellcast**, respectively. This means that the spawner in a spawn-destroy duet sometimes engages in a conversation with the destroyer and sometimes alternates creating and destroying an object. Similarly, the destroyer in the **Duet-Spawner-Destroyer** pattern and the responder in the **Duet-Spellcaster-Spellcaster** pattern inherit the responder tasks from the responder of the **Duet-Converser-Converser** patterns, so that in addition to the respective destroy/spellcast responses they can also engage in a conversation.

In addition to simplifying the understanding and utility of behaviour patterns, inheritance also supports code reuse. Figure 5.12 also shows the amount of automatically generated NWScript code for each behaviour pattern where the code from the parent behaviour pattern is reused. The code is divided into five components: inherited code (I), code used for the model (M), code used for the selector (S - the more proactive choices, the greater amount of selector code needed), code used for the proactive behaviour (P) that calls the appropriate task chain and code used for each task chain (T). In the case of the **Duet** patterns, the task chain code is divided into the code used by the initiator and the code used by the responder.

Inheritance is a mechanism for reusing behaviours. An alternative mechanism is a hierarchy of behaviours where new behaviours can be constructed by combining a sequence of other behaviours. There is no technical challenge in implementing hierarchical behaviours in our model, since a behaviour could be redefined to contain a sequence of tasks and behaviours, achieving a hierarchy of behaviours. However, we think this property of the model might have a negative impact on usability, since the model would become much more complex.

5.3 Evaluation Measures

We designed our behaviour system to satisfy the computational and functional requirements of commercial computer games [74]. Our behaviour model implementation is *robust, flexible, extendable, and scalable* to hundreds of NPCs active at a time.

- **Adaptability** Since ScriptEase generates scripts from behaviour pattern instances, an NPC cannot react to the game environment (e.g., the PC's actions, time of day, other NPCs) if a role or behaviour cue is not provided by the story author or by the pattern designer and it cannot acquire new behaviours dynamically. However, we provide a learning mechanism as another layer to our behaviour patterns. We described this mechanism in Chapter 4. This layer enables NPCs to select behaviours based on their previous experiences rather than on static probabilities or motivations. When their environment changes, the NPCs can adapt their choices to reflect the new conditions.
- **Clarity/Consistency/Intentionality** In our system, behaviours are easy to understand by observing the NPCs during game-play. For example, a guard NPC should be able to walk to the nearest creature (NPC/PC) in the room and start a conversation. This behaviour is easy to observe and to understand. We often encounter situations where the player may have to predict NPC behaviours based on some reasonable criteria. For example, if the PC intends to steal the guarded item, then the PC should wait until the guard rests to minimize the chances of being noticed. The design of our patterns, including the motivational and the learning components, supports the generation of unpredictable, but rational and *consistent* behaviours. If the general behaviour of the NPC is specified at design time, then the NPC is able to behave accordingly. At any time, the NPC chooses appropriate behaviours for the current role. For example, an NPC who has just finished the **Guard** role and performs the **Patron** role will not check the chest or attack an intruder. In general, the patron will not perform any of the behaviours of the **Guard** role when it is no longer in that role. Since an NPC displays a role activated

by a specific role cue (e.g., time of day) and because in each role the NPC acts according to that role, the NPC selects an appropriate behaviour at any time, but not always what the player expects. For example, even if it patrols for a while, the guard will not sit if the PC is close. ScriptEase provides a GUI in which behaviours are specified at a high level of abstraction. This facilitates pattern adaptation, experimentation, testing, and understanding. In addition, ScriptEase patterns can be easily understood by both story authors and pattern designers. Moreover, the generated scripts can be easily understood by programmers. The fully-commented code generated by ScriptEase from behaviour patterns can be found in a behaviour script that corresponds to the behaviour event. Therefore, a programmer can find the code from all the behaviours attached to this NPC together in one script. The script can be consulted and modified by a programmer if needed.

- **Effectiveness** In our system, since the behaviour pattern catalogue includes all the logic behind the behaviour, we generate correct and effective behaviours. During behaviour adaptation, the underlying synchronization mechanism is preserved, therefore the model cannot be altered. When an NPC uses learning to select behaviours, we include pre-training knowledge into the module for the specific NPC class to reduce the effects of learning the wrong behaviour during game-play.
- **Robustness** Our system ensures that an NPC quickly recovers if it is blocked due to game randomness. The reasons for blocking are discussed in Chapter 3. Our timeout mechanisms ensure that if the NPC does not respond in a specific number of seconds, then the behaviour dispatcher clears all outstanding actions and resends a behaviour event on the NPC. Our system also ensures no starvation: actors always execute a task, whether it originates from a latent (independent or collaborative), a proactive collaborative, a reactive (collaborative), or a proactive (independent) queue. Moreover, the learning system allows the NPC to avoid exploits that may occur in the NPC's behaviour. For example, in a combat situation, the opponents (including the PC) have to

always adjust their action selection to compete with our NPC.

- **Variety** Our system provides *variety*, first in selecting roles and second in choosing motivational or learning-based proactive behaviours. Each NPC varies its behaviour through the selection of roles. Furthermore, for each role, the NPC selects appropriate behaviours. We add unpredictability into our system by selecting the next behaviour out of all the possible behaviours randomly, based on static probabilities, motivations, or learning, to mimic the NPC decision-making. Events that happen in the game may alter the NPC's predilection to certain behaviours. Moreover, roles have a very important purpose, as they allow NPCs to adjust their behaviours and appear as though they are taking the appropriate course of action in every situation. For example, if a guard NPC patrols more before it rests, we understand that the guard becomes increasingly tired. However, the NPC may choose to check the guarded items a few times before it rests. An NPC's behaviours are also versatile, as the actor selects from a set of latent behaviours given a certain event in the game that interrupts the current behaviour. The "Bob-Sally" example in Chapter 3 illustrates the variety that can be achieved with our model. Our implementation uses the *NWN* game engine, which provides low-level parallelism, such as an NPC performing a walk and a speak action simultaneously. An author can insert the walk and the speak actions in a task of a behaviour and the game engine ensures basic time-sharing between the actions. The NPC starts walking and, since this action takes a long time to complete, the speak action overlaps its execution. Our AI architecture supports this directly, since the speak behaviour could be added on a different queue and could be performed simultaneously with behaviours stored on existing queues.
- **Autonomy*** The behaviour dispatcher for each NPC always selects an appropriate proactive (independent or collaborative) or reactive (collaborative) behaviour for that NPC autonomously. For example, a guard NPC patrols all day, checks the guarded item, and rests when it is tired, without instructions from an external source. Even in the case of a latent behaviour, the author de-

termines the appropriate response for each different NPC, so that they appear autonomous. The NPC adjusts its motivations to better adapt to the environment or it uses our learning algorithm to decide what action to select.

- **Alertness*** We support interruptible and resumable behaviours that create a more responsive NPC to the PC or to other NPCs, as illustrated in the “Bob-Sally” example of Chapter 3 shown in Figure 3.28, where a series of interruptions occurs. Our behaviour system generates behaviour events continuously: the behaviour script uses a delay command to generate a new behaviour event on the same NPC. The system selects latent behaviours first (according to the priority of the queues), as soon as their behaviour cues are triggered. This is possible because, after each task is complete, the control is given to the behaviour dispatcher that selects tasks to perform from the four available queues in descending priority order. Since only one latent queue is implemented, cascading latent behaviours are avoided. Consequently, latent behaviours are either cancelled (if they have lower priority) or executed immediately. The tasks contain the minimum number of actions necessary to ensure a coherent mini-behaviour and, at the same time, rapid execution. The behaviour dispatcher updates the consequences (motivations or learning data structures) of behaviours in a timely fashion, once per basic behaviour.
- **Interactivity*** In our system, an NPC can initiate a collaborative behaviour with another NPC and it can interact with the PC. An NPC is also able to respond to the actions of another NPC or the PC, as long as necessary role and behaviour cues are provided. For example, another NPC can start a dialogue with the guard. We provide proactive, reactive, and latent behaviours that can be independent or collaborative. In our system, latent behaviours always interrupt proactive or reactive behaviours and they cancel latent behaviours with lower priority. Therefore, the NPCs are not *self-absorbed* [48]. For example, a different NPC or a PC can interrupt the guard’s routine by stealing the guarded item. The guard has to be able to respond promptly to this situation by attacking the intruder or by executing a different appropriate

behaviour designed by the story author. Once again, the high level of potential interactivity is reflected by the “Bob-Sally” example of Chapter 3, where multiple individuals are involved in multiple interactions.

- **Reusability*** We developed a library of behaviour patterns that can be generalized to cover a variety of NPC behaviours. Reusability is a crucial part of behaviour design. The behaviour components included in a role are reusable from one NPC to another and even within the behaviours of the same NPC. For example, in a house scene, the **Patron** pattern designed for a tavern scene is sufficiently general to be used for the inhabitants, the **Server** pattern for a butler, and the **Owner** pattern for a cook. The butler interacts with the inhabitants, fetching for them by going to the kitchen. The inhabitants talk amongst themselves and the cook occasionally fetches supplies. Our approach handles group (crowd) behaviours in a natural way. The patrons constitute a crowd example - a group of NPCs with the same behaviour, but each selecting different behaviours based on local context. The same behaviour instance can be attached to a group of NPCs displaying the same general behaviour (e.g., all the patrons in a tavern scene) if the NPCs have identical tags. Each NPC in the group will select individual behaviours at different times based on local context. The ScriptEase behaviour pattern library provides sufficient reusable components to create other proactive, reactive, or latent behaviour patterns. For a single NPC, the potential **Approach a random patron** and **Approach-bar** tasks could be replaced by a single **Approach** task with a *target* option. The same basic behaviour can be instantiated in several ways. For example, the author may instantiate a **Converse-Listen** reactive behaviour twice: once with the “weather” topic and once with the “drink” topic. In addition, the same **Converse-Listen** reactive behaviour could be triggered by a proactive or latent behaviour on a collaborator. With ScriptEase, it is easy to create new reusable behaviours as well.
- **Scalability*** The total number of NPCs across *a module* used by a high-school student indicates that our behaviour system can be used without degra-

dation for over 200 NPCs. This study also shows that our behaviour system can be quickly and easily used with over 40 instances of 21 behaviour patterns. Therefore, our system scales well with an increased number of behaviour pattern templates and behaviour pattern instances from both the author's usability and the performance efficiency perspective. Note that it is possible for one instance to generate behaviour scripts for multiple NPCs. In *a single area*, a scene with three behaviour instances for eighteen patrons, two servers, and an owner ran for hours without degradation. Our system also scales well when the NPCs display *different roles*: a module with three guard NPCs, each with four roles instantiated at a time and with each role displaying seven basic behaviours, was played and it ran without performance degradation.

5.4 Chapter Summary

In this chapter, we presented the experiments we conducted to validate our behaviour model. The evaluation of our learning mechanism was presented in Chapter 4. From a usability perspective, we wanted to determine whether behaviour patterns were simple enough to be used by non-programmers. We conducted two experiments in which high-school students created interactive stories using encounter and behaviour patterns. From an efficacy perspective, we developed a set of three case studies focused on the correctness of the generated behaviours, the expressiveness of the behaviour patterns, including the capability of our behaviour system to support new behaviour patterns, and the potential for reusability of our existing behaviour patterns. In addition, we discussed how our model meets the computational and functional requirements of commercial computer games that we introduced in Chapter 2.

Chapter 6

Conclusions and Future Work

In the past, ScriptEase used *encounter patterns* to represent interactions between the PC and inanimate game objects [49]. In this dissertation, we described a model for representing NPC behaviours using generative patterns. This model provides a solution to the difficult problem of interacting NPCs. We designed and implemented *behaviour patterns* in ScriptEase to realize our behaviour model. We validated our approach using a real commercial application, BioWare Corp.'s *Neverwinter Nights* game. The first implementation of behaviour patterns used encounter patterns as primitives. A native ScriptEase implementation followed. We are building a common library of rich behaviour patterns for use and reuse across CRPGs. We showed how patterns hide the level of complexity necessary to create a realistic interactive story and how the same patterns can be reused for several NPCs.

Our behaviour patterns provide the following important innovations:

1. They support proactive and reactive, as well as independent behaviours for individual NPCs and collaborative behaviours for interacting NPCs. These behaviours can be interrupted and resumed depending on their *behaviour cue* priorities.
2. In addition to proactive behaviours, they support reactive behaviours triggered as a response to collaborative behaviours, as well as latent behaviours triggered by NPCs or by the PC. The behaviour patterns include the ability to return to partially-completed interrupted behaviours after the interrupting behaviour is completed.

3. They support *performances*, a powerful mechanism that allows the NPC to change behaviour models (*roles*) activated by *role cues* (such as game events or timers) during the story.
4. They include a novel collaboration protocol based on topics that simplifies the way in which NPCs can collaborate and that allows them to more easily interact with a broader range of NPC types, without having to know the specific collaborator until the game is played. This protocol is easy to use by story authors and it can be employed to reuse basic behaviours for both sides (initiator and responder) of a collaboration.
5. They facilitate a system for generating complex NPC behaviours that is accessible to non-programmers and that requires no manual coding.
6. They contain a concurrency mechanism that solves the inherent synchronization problems of collaborative and interruptible behaviours.
7. They provide a motivation model for selecting proactive behaviours.
8. They support a novel online learning algorithm for behaviour selection.

In the future, the model can be extended to support more than two NPC collaborations at the same time. In this case, an additional queue is needed for each new collaboration. We added learning capabilities to the NPCs for combat support that generate even more believable behaviours and interactions. To achieve this goal, we built a learning model on top of our motivational model to take full advantage of the behaviour patterns we already constructed.

Our collaborative protocol mechanism can be extended to include a family of topics that would result in broader NPC collaborations and more behaviour variety. If the NPC behaviours in *The Elder Scrolls 4: Oblivion* [82] were analyzed, in addition to new behaviour patterns emerging, opportunities for model expansion may be discovered. There are approximately 1,000 NPCs in *Oblivion*, excluding the NPCs that are spawned randomly during the game. A case study directed at *Oblivion* NPC behaviours could test our behaviour pattern library in a different CRPG. One

of the cities in *Oblivion* can be chosen and all the interesting behaviours that occur in selected areas of the city can be reproduced using our behaviour patterns.

As described in Chapter 5, it would be easy to add hierarchical behaviours to our model. However, it is an open question as to whether this would complicate the user interface. A study could be performed to determine the utility of hierarchical behaviours.

Each of these goals involves escalating challenges, but we have constructed our behaviour model with these challenges in mind. We initially designed our model to support proactive independent behaviours selected with a static probability function to eliminate repetitive behaviours that may decrease the interest of the player. Then, we extended our model to include proactive collaborative and reactive behaviour patterns. In the next step, we added motivations to proactive behaviour selection by designing a motivation construct that, when used, replaced the static probability selection model (the default behaviour selector). We designed a collaborative protocol by exploiting the existing basic behaviours and connecting them at run-time based on topics to maximize the NPCs' interaction potential. At the same time, we preserved the *eye-contact* protocol, together with the framework of the previous collaborative behaviour model. Then, we allowed behaviours to be interrupted and we enhanced our model to accommodate latent behaviours. In addition to triggering latent behaviours based on behaviour cues, we support the selection of latent behaviours either probabilistically (when two or more latent behaviours could be triggered by the same behaviour cue) or based on motivation for more dynamic and challenging opponents and allies. We constructed a synchronization model that is scalable to the more complex interactions that can take place among NPCs and between NPCs and the PC. For example, the tavern scene example can be generalized to cover the behaviours of NPCs in other settings. The same patterns, but with different parameters, can be used to generate behaviours for NPCs in a mansion, where tavern patrons are replaced by inhabitants, the tavern server is replaced by a butler, and the tavern owner is replaced by a cook. Finally, the progression from probabilistic to motivation-based behaviour selection was extended to learning-based behaviours. We incorporated ALeRT, a modification of

the Sarsa(λ) online reinforcement learning algorithm, to ensure that behaviours are selected more naturally, depending on the NPCs' previous experiences and interactions with their environment.

This research could be extended to consider a team of cooperating agents, instead of individual agents. For example, a team composed of a fighter and a sorcerer could challenge another team composed of a fighter and a sorcerer. Spronck's pre-built arena module will support experiments on such cooperating agents and ScriptEase supports cooperative behaviours, so generating the scripts is possible. In this case, the state space would be enlarged, since the attributes of the player's team members and multiple opponents would have to be represented. In the team case, it may also be necessary to construct models of the other agents, since one NPC's best action may depend on the actions of its team members and opponents.

It would also be helpful to dynamically discover the game state necessary for efficient learning, instead of requiring the pattern designer to specify it. After several scenarios are executed, game state options could be suggested to the pattern designer, based on game state changes during the actions being considered. The pattern designer should be able to easily modify RL parameter values. Moreover, the pattern designer should be able to adjust a game's difficulty level using a threshold that represents the maximum amount that an agent is allowed to exploit another agent (usually the PC). ALeRT could be modified so that the agent's value function does not exceed this threshold. This could be accomplished by increasing exploration or by not picking the best action during exploitation. We could also experiment with different policies throughout the game, depending on the changes in the environment [5].

We have experimented with a *variable lambda* parameter that automatically adjusts its value to the number of steps in an episode. The value of lambda changes dynamically whenever an NPC undergoes an equipment configuration change. More experiments are necessary to assess the utility of this approach. We expect the agent to learn faster in some situation if lambda is a function of the number of steps in an episode (e.g., directly proportional). For example, on average, in the Melee phase of a Melee-Ranged experiment there are 3.5 steps per episode and in the Ranged

phase there are 5.5 steps per episode. Lambda is responsible for propagating future rewards quickly to earlier actions. Therefore, lambda needs a higher value to propagate knowledge back to the start action in a longer episode.

Finally, evaluation mechanisms must be developed for measuring success in non-combat behaviours. This is a hard problem, but without metrics, it is difficult to assess whether learning is sufficiently effective. In general, metrics can be inferred from the requirements presented in Chapter 2 to evaluate NPC behaviours. A simple reward function can be computed from an NPC's motivations. For example, for a guard NPC, we could introduce a reward function that is the sum of the motivational attributes, **Duty**, **Tiredness**, and **Threat**. In fact, in this case the reward should be the negative of the sum, since the guard wants to minimize all these attributes.

Although we demonstrated our approach using a CRPG, our model based on the generality of the design pattern abstraction could have a broader application domain that includes other kinds of computer games, synthetic performance, autonomous agents in virtual worlds, and animation of interactive objects.

Bibliography

- [1] Aurora Toolset, BioWare Corp. 2009. <http://nwn.bioware.com/builders>.
- [2] N. Badler, B. Webber, W. Becket, C. Geib, M. Moore, C. Pelachaud, B. Reich, and M. Stone. Planning and parallel transition networks: Animation's new frontiers. In *Proceedings of the Computer Graphics and Applications: Pacific Graphics '95*, pages 101–117, 1995.
- [3] S. Björk and J. Holopainen. *Patterns in Game Design*. Charles River Media Hingham, Mass., 2004.
- [4] D. M. Bourg and G. Seemann. *AI for Game Developers*. O'Reilly Media, Inc., 2004.
- [5] M. Bowling and M. Veloso. Rational and convergent learning in stochastic games. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 1021–1026, 2001.
- [6] M. Bowling and M. Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2):215–250, 2002.
- [7] J. Bradley and G. Hayes. Group utility functions: Learning equilibria between groups of agents in computer games by modifying the reinforcement signal. In *Congress on Evolutionary Computation*, 2005.
- [8] Bungie Studios. 2009. <http://www.bungie.net>.
- [9] A. Caicedo and D. Thalmann. Virtual humanoids: Let them be autonomous without losing control. In *Proceedings of the 4th Conference on Computer Graphics and Artificial Intelligence*, pages 101–117, 2000.
- [10] T. K. Capin, I. S. Pandzic, H. Noser, N. M. Thalmann, and D. Thalmann. Virtual human representation and communication in VLNET. *IEEE Computer Graphics and Applications*, 17(2):42–53, 1997.
- [11] M. Carbonaro, M. Cutumisu, H. Duff, S. Gillis, C. Onuczko, J. Siegel, J. Schaeffer, A. Schumacher, D. Szafron, and K. Waugh. Interactive Story Authoring: A Viable Form of Creative Expression for the Classroom. *Computers and Education*, 51(2):687–707, 2008.
- [12] M. Carbonaro, M. Cutumisu, M. McNaughton, C. Onuczko, T. Roy, J. Schaeffer, D. Szafron, S. Gillis, and S. Kratchmer. Interactive story writing in the classroom: Using computer games. In *Proceedings of the International Digital Games Research Conference (DiGRA 2005)*, pages 323–338, Vancouver, Canada, June 2005.

- [13] M. Cavazza, F. Charles, and S. J. Mead. Interacting with virtual characters in interactive storytelling. In *Proceedings of the ACM Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002)*, pages 318–325, Bologna, Italy, 2002.
- [14] E. G. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, June 1971.
- [15] M. Cutumisu, C. Onuczko, M. McNaughton, T. Roy, J. Schaeffer, A. Schumacher, J. Siegel, D. Szafron, K. Waugh, M. Carbonaro, H. Duff, and S. Gillis. ScriptEase: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 67(1):32–55, June 2007.
- [16] M. Cutumisu, C. Onuczko, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, J. Siegel, and M. Carbonaro. Evaluating pattern catalogs - the computer games experience. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 132–141, Shanghai, China, May 2006.
- [17] M. Cutumisu and D. Szafron. A demonstration of agent learning with action-dependent learning rates in computer role-playing games. In *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2008)*, pages 218–219, Stanford, USA, October 22-24 2008.
- [18] M. Cutumisu, D. Szafron, M. Bowling, and R. S. Sutton. Agent learning using action-dependent learning rates in computer role-playing games. In *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2008)*, pages 22–29, Stanford, USA, October 22-24 2008.
- [19] M. Cutumisu, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, C. Onuczko, and M. Carbonaro. Generating ambient behaviors in computer role-playing games. In *Proceedings of the 1st Intelligent Technologies for Interactive Entertainment (INTETAIN 2005)*, LNAI 3814, Springer-Verlag, pages 34–43, Madonna di Campiglio, Italy, November 30 - December 2 2005.
- [20] M. Cutumisu, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, C. Onuczko, and M. Carbonaro. Generating ambient behaviors in computer role-playing games. *IEEE Journal of Intelligent Systems (IEEE IS)*, 21(5):19–27, 2006.
- [21] K. Czarnecki and U. Eisenecker. *Generative Programming Methods, Tools, and Applications. Chapter 11: Intentional Programming*. Addison-Wesley, Reading, MA, 2000.
- [22] Destroy All Humans! 2. Pandemic Studios. 2009. <http://www.destroyallhumansgame.com>.
- [23] M. Dyckhoff. Decision Making and Knowledge Representation in Halo 3. In *Machine Learning and Games (MALAGA) Workshop, NIPS '07*, Whistler BC, 2007. <http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/nips07.pdf>.
- [24] M. Dyckhoff. Evolving Halo’s Behavior Tree AI. Invited talk. In *Game Developers Conference, 2007*. <http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/gdc07.pdf>.
- [25] EA FIFA Soccer 2004. 2009. <http://www.fifa04.com>.

- [26] EA FIFA Soccer 2006 GameSpot Review. 2009. <http://www.gamespot.com/xbox360>.
- [27] Electronic Arts (EA). 2009. <http://www.ea.com>.
- [28] Epic Games. 2009. <http://www.epicgames.com>.
- [29] Fable, Lionhead Studios. 2009. <http://www.fablegame.com>.
- [30] T. Fullerton, C. Swain, and S. Hoffman. *Game Design Workshop: Designing, Prototyping, and Playtesting Games*. CMPBooks, San Francisco, 2004.
- [31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [32] T. Graepel, R. Herbrich, and J. Gold. Learning to fight. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004.
- [33] B. Grosz and S. Kraus. Collaborative plans for complex group actions. *Artificial Intelligence*, 86:269–358, 1996.
- [34] E. Grundstrom, Electronic Arts/Maxis. The AI of Spore. Invited talk. In *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, Stanford, USA, October 22-24 2008.
- [35] Guard Behaviour Movies. 2009. <http://www.cs.ualberta.ca/~script/movies/ai-ide07>.
- [36] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
- [37] S. W. Hasinoff. Reinforcement learning for problems with hidden state. Technical report, University of Toronto, Department of Computer Science, 2002.
- [38] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation (2nd Edition)*. Addison-Wesley, Reading, MA, 2000.
- [39] R. Houlette and D. Fu. The Ultimate Guide to FSMs in Games. *AI Game Programming Wisdom 2*, Charles River Media, 2003.
- [40] D. Isla. Handling complexity in the Halo 2 AI. In *Proceedings of the Game Developers Conference (GDC 2005)*, 2003.
- [41] J. P. Kelly, A. Botea, and S. Koenig. Offline planning with hierarchical task networks in video games. In *4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2008)*, pages 60–65, Stanford, USA, October 22-24 2008.
- [42] B. King and J. Borland. *Dungeons and Dreamers: The Rise of Computer Game Culture from Geek to Chic*. McGraw-Hill/Osborne, 2003.
- [43] B. Kreimeier. The case for game design patterns. 2002. http://www.gamasutra.com/features/20020313/kreimeier_03.htm.
- [44] H. Kushner and D. Clark. *Stochastic Approximation Methods for Constrained and Unconstrained Systems*. New York: Springer-Verlag, 1978.

- [45] Lilac Soul's NWN Script Generator. 2009. <http://nwvault.ign.com/view.php?view=other.detail&id=625>.
- [46] M. Mateas and A. Stern. Façade: An experiment in building a fully-realized interactive drama. In *Proceedings of the Game Developers Conference (GDC 2003), Game Design Track*, March 2003.
- [47] M. Mateas and A. Stern. Beyond state machines: Managing complex, intermixing behavior hierarchies. In *Proceedings of the Game Developers Conference (GDC 2004), Programming Track*, March 2004.
- [48] M. Mateas and A. Stern. Procedural authorship: A case-study of the interactive drama Façade. In *Digital Arts and Culture (DAC)*, Copenhagen, November 2005.
- [49] M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. ScriptEase: Generative design patterns for computer role-playing games. In *Proceedings of the 19th IEEE Conference on Automated Software Engineering (ASE 2004)*, pages 88–99, Linz, Austria, September 2004.
- [50] M. McNaughton, J. Redford, J. Schaeffer, and D. Szafron. Pattern-based AI scripting using ScriptEase. In *Proceedings of the 16th Canadian Conference on Artificial Intelligence (AI 2003)*, pages 35–49, Halifax, Canada, June 2003.
- [51] M. McNaughton, J. Schaeffer, D. Szafron, D. Parker, and J. Redford. Code generation for AI scripting in computer role-playing games. In *Challenges in Game AI Workshop at AAAI-04*, pages 129–133, San Jose, USA, July 2004.
- [52] M. McPartland and M. Gallagher. Learning to be a bot: Reinforcement learning in shooter games. In *4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2008)*, pages 78–83, Stanford, USA, October 22-24 2008.
- [53] Memetic Artificial Intelligence Toolkit. 2009. <http://www.memeticai.org>.
- [54] K. Merrick and M-L. Maher. Motivated reinforcement learning for non-player characters in persistent computer game worlds. In *ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, Los Angeles, USA, 2006.
- [55] Morrowind, Bethesda Softworks. 2009. <http://www.morrowind.com>.
- [56] S. R. Musse, C. Babski, T. K. Capin, and D. Thalmann. Crowd modelling in collaborative virtual environments. In *Proceedings of the ACM Symposium on VRST*, pages 115–123, 1998.
- [57] S. Nason and J. E. Laird. Soar-RL: Integrating reinforcement learning with Soar. *Cognitive Systems Research*, 6(1):51–59, 2005.
- [58] Neverwinter Nights. 2009. <http://nwn.bioware.com>.
- [59] Neverwinter Nights Vault. 2009. <http://nwvault.ign.com>.
- [60] Neverwinter Nights Wiki. 2009. http://nwn.wikia.com/wiki/main_page.
- [61] J. Orkin. Three states and a plan: The A.I. of F.E.A.R. In *Proceedings of GDC-06*, 2006.

- [62] M. D. Pendrith and M. R. K. Ryan. Estimator variance in reinforcement learning: Theoretical problems and practical solutions. In *On-line Search: Collected Papers from the 1997 Workshop. AAAI Technical Report WS- 97-10. AAAI Press.*, pages 81–88, 1997.
- [63] K. Perlin and A. Goldberg. Improv: A system for scripting interactive actors in virtual worlds. *SIGGRAPH*, 29(3):205–216, 1996.
- [64] F. Poiker. Creating scripting languages for non-programmers. *AI Game Programming Wisdom, Charles River Media*, pages 520–529, 2002.
- [65] Quake, id Software. 2009. <http://www.idsoftware.com/games/quake>.
- [66] S. Rabin. Promising game AI techniques. *AI Game Programming Wisdom 2, Charles River Media*, 2003.
- [67] E. Sacerdoti. The Nonlinear Nature of Plans. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 206–214, 1975.
- [68] D. Sanchez-Crespo Dalmau. *Core Techniques and Algorithms in Game Programming*. Indianapolis, Indiana: New Riders, 2003.
- [69] B. Schwab. Implementation walkthrough of a homegrown “abstract state machine” style system in a commercial sports game. In *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference (AI-IDE)*, pages 145–148, Stanford, USA, October 22-24 2008.
- [70] ScriptEase. 2009. <http://www.cs.ualberta.ca/~script/scriptease.html>.
- [71] Spore. 2009. <http://www.spore.com>.
- [72] Spore. The Sporum - The Official Spore Forum. 2009. <http://forum.spore.com/jforum/posts/list/1928.page>.
- [73] P. Spronck. Adaptive entertainment. Invited address. In *Proceedings of the 2nd Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, Marina del Rey, USA, June 20-23 2006. <http://www.aiide.org/aiide2006/speakers/spronck.ppt>.
- [74] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma. Adaptive game AI with dynamic scripting. *Machine Learning*, 63(3):217–248, 2006.
- [75] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma. Online adaptation of computer game opponent AI. In *Proceedings of the 15th Belgium-Netherlands Conference on Artificial Intelligence*, pages 291–298, 2003.
- [76] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma. Online adaptation of game opponent AI with dynamic scripting. *International Journal of Intelligent Games and Simulation*, 3(1):45–53, 2004.
- [77] Spronck’s Arena Module. 2009. <http://www.cs.unimaas.nl/p.spronck/GameAI/OnlineAdaptation3.zip>.
- [78] R. S. Sutton. Adapting bias by gradient descent: An incremental version of Delta-Bar-Delta. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 171–176, 1992.

- [79] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, Mass.: MIT Press, 1998.
- [80] D. Szafron, M. Carbonaro, M. Cutumisu, S. Gillis, M. McNaughton, C. Onuczko, T. Roy, and J. Schaeffer. Writing interactive stories in the classroom. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning (IMEJ)*, 7(1), 2005.
- [81] Tavern Behaviour Movies. 2009. <http://www.cs.ualberta.ca/~script/movies/aaai07>.
- [82] The Elder Scrolls IV: Oblivion, Bethesda Softworks. 2009. <http://www.elderscrolls.com>.
- [83] T. Timuri, P. Spronck, and J. van den Herik. Automatic rule ordering for dynamic scripting. In *Proceedings of the 3rd Artificial Intelligence and Interactive Digital Entertainment (AIIDE) Conference*, pages 49–54, Stanford, USA, June 6-8 2007.
- [84] UnrealKismet, Unreal Technology. 2009. <http://www.unrealtechnology.com>.
- [85] R. Valdes. In the mind of the enemy: The artificial intelligence of Halo 2. 2004. <http://stuffo.howstuffworks.com/halo2-ai.htm>.
- [86] M. Vasta, S. Lee-Urban, and H. Muñoz-Avila. RETALIATE: Learning winning policies in first-person shooter games. In *Proceedings of the 17th Innovative Applications of Artificial Intelligence Conference (IAAI-07)*, Vancouver, Canada, July 2007.
- [87] R. M. Young. An overview of the Mimesis architecture: Integrating intelligent narrative control into an existing game environment. In *Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, pages 78–81, USA, 2001.

Appendix A

An Introduction to ScriptEase

This appendix is an excerpt from one of our previous articles [15] placed here for the convenience of the reader.

“ScriptEase patterns are applied to NPCs and other game objects. BioWare Corp.’s Aurora Toolset is used to define and populate scenes. The Aurora Toolset is a drag-and-drop CAD tool for creating game worlds. It provides a rich palette of interior and exterior map tiles, objects, creatures, etc. for creating the environments in which the game story will unfold. The tool is intuitive and easy to use. ScriptEase makes the scripting of game objects as easy as the Aurora Toolset makes the creation and placement of game objects. Fig. A.1 shows an author placing a container (named *Dresser*) into a room; it will be used in the example of this section and it comes from one of the stories written by a student author in the case study of Section 5.

A story is written by instantiating patterns and adapting the generated descriptions. In the example story, the author wants the PC to open the *Dresser* shown in Fig. A.1. When the PC opens it and removes an item from it, the author wants a creature named *Avadel* to be spawned. The author created *Avadel* using the Aurora Toolset (not shown). A similar scenario occurs frequently in role-playing fantasy games (and stories). However, it involves some other container rather than the *Dresser* and some other creature rather than *Avadel*, since these objects are specific to this story. Because this scenario occurs so often, ScriptEase has a pattern for it: *Container disturb - spawn creature*. To use the pattern in a particular game story, the author must adapt the pattern by specifying three options (parameters): the con-

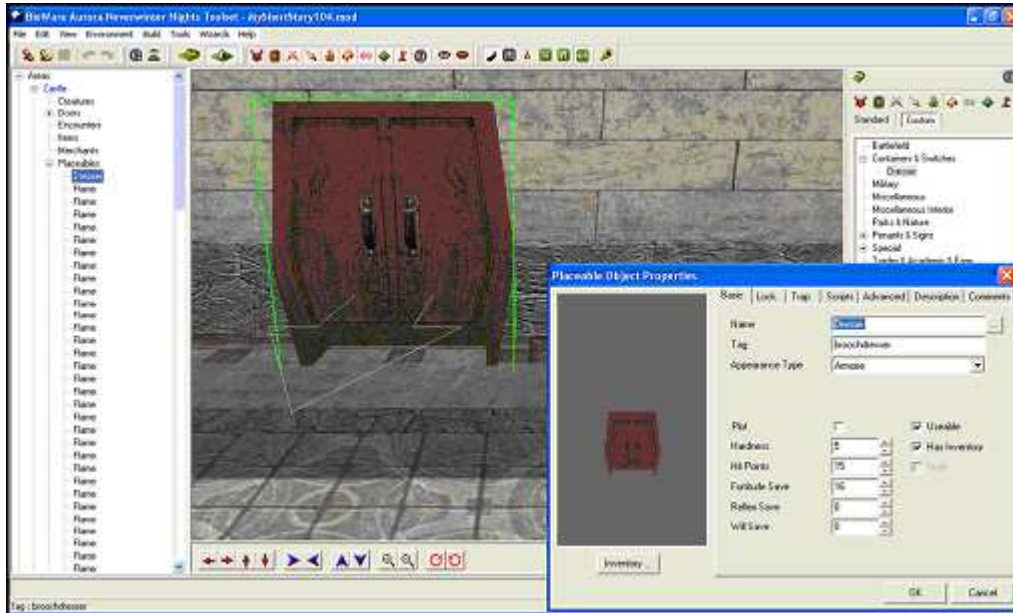


Figure A.1: Creating and placing a container using the Aurora Toolset.

tainer that the pattern applies to, the creature that gets spawned when an item is removed from the container, and the visual effect (if any) that occurs during the spawning. Fig. A.2 shows the generated description of this pattern and one of the options being set by the author.

The author has opened a story file (*MyShortStory*) created using the Aurora Toolset and has created an instance of an *encounter* pattern (identified by the stylized E), *Container disturb - spawn creature*. The author has selected the pattern and four tabs have appeared. If the Description tab was selected (it is not), the author would see a summary of the intent of the pattern. The other three tabs are option tabs (*The Container*, *Creature Blueprint* and *Spawn Effect*). The author has adapted this pattern to the story by selecting the *Creature Blueprint* option using a dialog box to select a particular creature named *Avadel*. The author has previously adapted the pattern by setting *The Container* option to the *Dresser* and the *Spawn Effect* to be a *Pulse, Holy* visual effect, using similar dialogs.

Every encounter pattern contains one or more *situations* (stylized S). This encounter pattern is shown in Fig. A.2 and it has been opened to reveal its three situations. One applies when an item is added to the container, one applies when

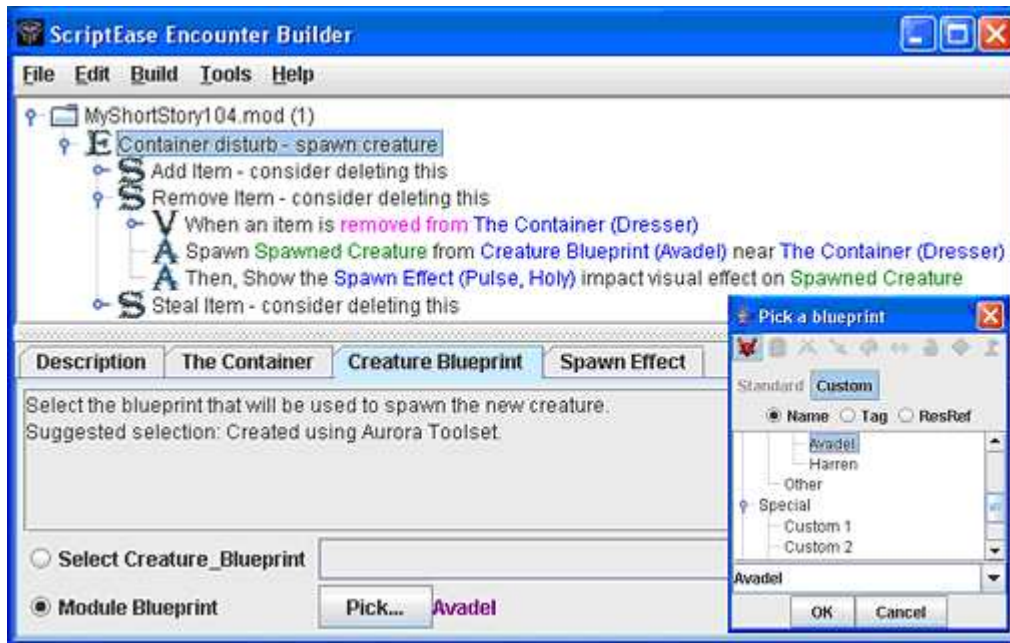
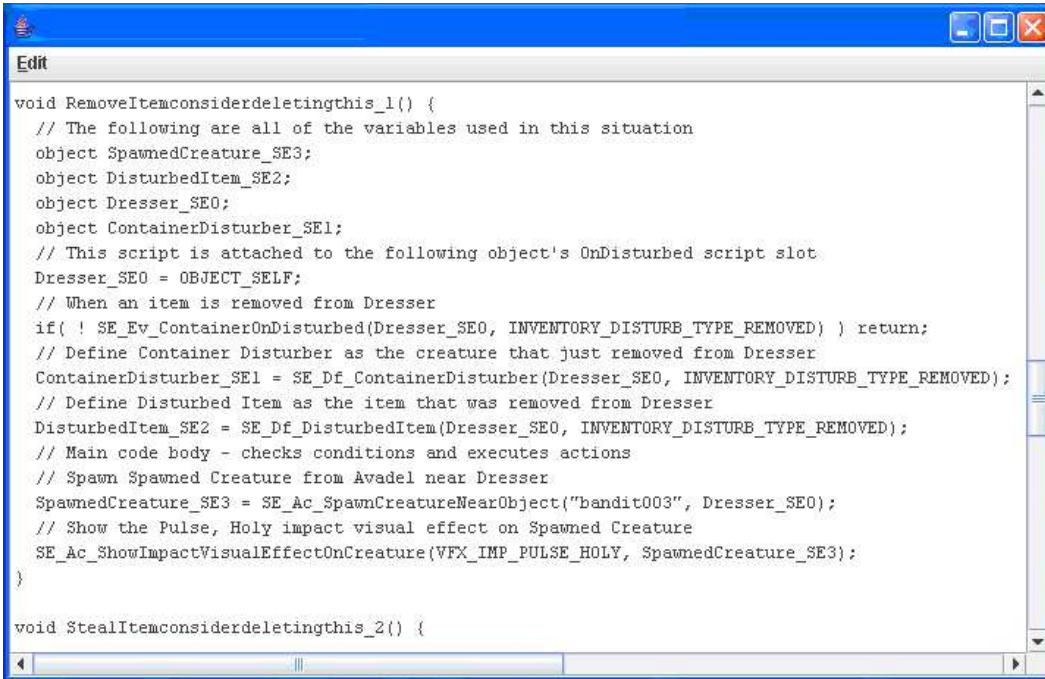


Figure A.2: A generative pattern, its description and a dialog being used to set an option.

an item is removed and one applies when an item is stolen. In Fig. A.2, the *Remove* situation is expanded to show its components. Every situation contains one *Event* (stylized V) that describes the circumstances under which the situation applies. The *Remove* situation applies when *an item is removed from The Container (Dresser)*. Every situation contains one or more *actions* (stylized A) that will be performed when the situation applies. The *Remove* situation contains two actions, one to spawn a creature and the other to show a visual effect. A situation can also contain two other kinds of components, definitions and conditions, which will be discussed in Section 4. Note the use of colors in the text to clearly indicate the parts of the natural language description that are part of the pattern (black), options (blue), conditions (red), and definitions (green).

As the author adapts the pattern by setting options, the description is updated to reflect the choices. The description in Fig. A.2 has been adapted for the author's story as the pattern options were set. If the author selects *Generate Code* from the menu, the NWScript code for the pattern would be automatically generated and inserted into the story file at the appropriate place. Fig. A.3 shows a portion of the 58

lines of BioWare Corp.'s NWScript code that was generated for this adapted pattern description. A game author would never need to see the automatically generated commented code. However, a programmer on the team may choose to view the generated scripts and to optionally edit the scripts manually.



```
void RemoveItemconsiderdeletingthis_1() {
    // The following are all of the variables used in this situation
    object SpawnedCreature_SE3;
    object DisturbedItem_SE2;
    object Dresser_SE0;
    object ContainerDisturber_SE1;
    // This script is attached to the following object's OnDisturbed script slot
    Dresser_SE0 = OBJECT_SELF;
    // When an item is removed from Dresser
    if( ! SE_Ev_ContainerOnDisturbed(Dresser_SE0, INVENTORY_DISTURB_TYPE_REMOVED) ) return;
    // Define Container Disturber as the creature that just removed from Dresser
    ContainerDisturber_SE1 = SE_Df_ContainerDisturber(Dresser_SE0, INVENTORY_DISTURB_TYPE_REMOVED);
    // Define Disturbed Item as the item that was removed from Dresser
    DisturbedItem_SE2 = SE_Df_DisturbedItem(Dresser_SE0, INVENTORY_DISTURB_TYPE_REMOVED);
    // Main code body - checks conditions and executes actions
    // Spawn Spawned Creature from Avadel near Dresser
    SpawnedCreature_SE3 = SE_Ac_SpawnCreatureNearObject("bandit003", Dresser_SE0);
    // Show the Pulse, Holy impact visual effect on Spawned Creature
    SE_Ac_ShowImpactVisualEffectOnCreature(VFX_IMP_PULSE_HOLY, SpawnedCreature_SE3);
}

void StealItemconsiderdeletingthis_2() {
```

Figure A.3: A portion of the code generated for the pattern in Fig. A.2.

In this story, the author wants the actions to occur only when an item is removed from the *Dresser*. Although adding and stealing from a container are other possibilities supported by the (general) pattern, the author does not want to use them. Therefore, in addition to setting the pattern options, the author can further adapt the pattern description by highlighting the *Add* and *Steal* situations, one at a time, and selecting the *Delete* option from a pop-up menu, before generating the scripting code.

Pattern descriptions can be adapted (customized) by setting options, adding, deleting or replacing pattern components. We have already seen that deleting situations is easy to do. Adding an action to a situation is also simple in ScriptEase. For example, the author could add an action to the existing pattern so that the caption *Run for you life!* is displayed above the spawned creature's head. The author clicks

to highlight the *Remove* situation, selects *add an action* from a pop-up menu and navigates a set of hierarchical menus to find the desired action. Fig. A.4 shows the results of the author’s efforts. Note that the *Add* and *Steal* situations have been deleted. The resulting pattern description would generate 38 lines of NWScript code (not shown). After generating the scripting code, the author can “test-drive” the story by opening it in *Neverwinter Nights (NWN)*. This allows the author to incrementally change the story and immediately test the new story to verify its correctness.

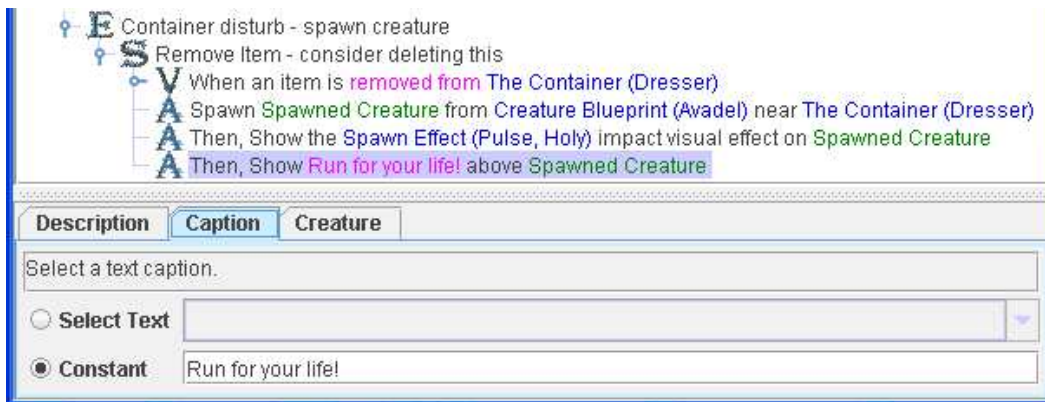


Figure A.4: Adapting a pattern by adding an action.

The *Container disturb - spawn creature* example shows how easy it is for an author (non-programmer) to use ScriptEase to create an interactive game story. The ScriptEase three-step approach (instantiate, adapt, generate) provides many benefits for simplifying the creation of interactive stories:

1. All authoring is done using familiar story-element patterns.
2. The author does not need to know anything about programming or scripting languages.
3. The author sees a natural language description of the story.
4. The author adapts the story using a simple menu-driven interface.
5. Many common programming errors are eliminated. The patterns have been tested and debugged by the pattern designer, before the author uses them, not

by the author during story writing.

6. Enabling an author to directly generate an interactive story eliminates the programmer as an intermediary and as a potential source of errors.
7. Since there can be thousands of objects that are scripted in a story, each requires a unique label at the script level. ScriptEase manages these transparently using natural language pronouns to refer to objects in context. Hiding these unique labels from the author reduces complexity and allows the author to work at a higher level of abstraction. By reducing the effort required to add scripts, the author has more time to add additional interactive (scripted) NPCs and objects, thereby increasing the richness of the story.
8. Adapted patterns are used to generate scripting code. This code does not have to be viewed by the author. However, it is available for further adaptation by programmers.”

Appendix B

Behaviour Pattern Catalogue Description

This appendix briefly describes the behaviour patterns comprising the behaviour pattern catalogue that we created for the user study of Section 5.1. We have since modified the catalogue by taking advantage of the cue mechanism to combine similar behaviours that are triggered by different cues. In all our tasks, an NPC speaks using two actions: uttering a random text from a dialogue file with different one-liners that represent variations on a topic and performing a “Talk” animation. Each non-object option (e.g., text, number, etc.) has a default setting. The user may decide to change these values to adapt the pattern, but this is not necessary to use the patterns.

B.1 Approacher

- **Description:** The NPC turns to face a target, it speaks, it moves (walks or runs) to the target, and it turns to face the target again.
- **Options:**
 - Target: the object that the NPC approaches.
 - Text: the text spoken by the NPC while approaching.

B.2 Attacker

- **Description:** The NPC turns to face a target, it speaks, and it attacks the target.
- **Options:**
 - Target: the object that the NPC attacks.
 - Text: the text spoken by the NPC while attacking.

B.3 Beckoner

- **Description:** The NPC turns to face a target while speaking, if the distance between the NPC and the target is within a range specified by the user.
- **Options:**
 - Target: the object that the NPC beckons.
 - Text: the text spoken by the NPC while beckoning.
 - Range: a threshold value for the distance between the NPC and the target.

B.4 Beseecher

- **Description:** The NPC turns to face a target, it moves to the target, it turns to face the target again, and it speaks, if the distance between the original location of the NPC at the beginning of the game and the current location of the target is within a range specified by the user.
- **Options:**
 - Target: the object that the NPC beseeches.
 - Text: the text spoken by the NPC while beseeching.
 - Range: a threshold value for the distance between the NPC and the target.

B.5 Challenger

- **Description:** The NPC turns to face a target, it moves to the target, it turns to face the target again, and it initiates a dialogue with the target, if the distance between the original location of the NPC at the beginning of the game and the current location of the target is within a range specified by the user.
- **Options:**
 - Target: the creature that the NPC challenges.
 - Dialogue: the dialogue file initiated by the NPC with the target.
 - Range: a threshold value for the distance between the NPC and the target.

B.6 Checker

- **Description:** The NPC turns to face a target, it speaks, it moves to the target, it turns to face the target again, it opens the target, it performs an animation to look inside the target for an item, and it closes the target.
- **Options:**
 - Target: the placeable that the NPC checks.
 - Item: the item for which the NPC checks the target.
 - Text: the text spoken by the NPC while preparing to check if the item is in the target.
 - Gone Text: the text spoken by the NPC when discovering that the item is missing from the target placeable.

B.7 Destroyer

- **Description:** The NPC turns to face a target, it speaks, it casts a spell on the target, and it destroys the target.

- **Options:**
 - Target: the object that the NPC destroys.
 - Text: the text spoken by the NPC while destroying.
 - Spell: the spell cast by the NPC while destroying.

B.8 Dispossessor

- **Description:** The NPC takes an item from a target (creature or placeable) while speaking. The NPC turns to face a target placeable, it speaks, it moves to the target, it turns to face the target again, it performs an animation that mimics the action of taking an item from the target, and it finally takes the item from the target.
- **Options:**
 - Target: the placeable that the NPC dispossesses.
 - Text: the text spoken by the NPC while taking the item from the target.
 - Item: the item that the NPC takes from the target.

B.9 Exclaimer

- **Description:** The NPC turns to face a target and it speaks (performing a “Talk” animation).
- **Options:**
 - Target: the object to which the NPC speaks.
 - Text: the text spoken by the NPC.

B.10 Exclaimer with Animation

- **Description:** The NPC turns to face a target and it speaks performing a random talk animation.

- **Options:**
 - Target: the object to which the NPC speaks.
 - Text: the text spoken by the NPC.

B.11 Expert

- **Description:** The NPC turns to face a target, it speaks, it moves to the target, it turns to face the target again, and it performs a skill.
- **Options:**
 - Target: the object in front of which the NPC performs the skill.
 - Text: the text spoken by the NPC while performing the skill.
 - Skill: the skill that the NPC performs.

B.12 Follower

- **Description:** The NPC speaks and it follows a target (using a follow NWScript action) within a distance specified by the user.
- **Options:**
 - Target: the object that the NPC follows.
 - Text: the text spoken by the NPC while following.
 - Range: the follow distance of the NPC from the target.

B.13 Guard

- **Description:** The NPC guards an item located in a placeable. The NPC patrols, it checks the target for an item, it rests on a seat, it initiates conversations with fellow guards, it reacts to conversations initiated by other NPCs, it watches for intruders that move within a watch range from the target, it warns any intruder that moves within a warn range from the target, and it attacks intruders that move within an attack range from the target.

- **Options:**
 - Target: the placeable that the NPC guards.
 - Item: the item that the NPC guards.
 - Seat: the placeable that the NPC uses to sit.
 - Watch Range: the threshold of the watch distance between an intruder and the target.
 - Warn Range: the threshold of the warn distance between an intruder and the target.
 - Attack Range: the threshold of the attack distance between an intruder and the target.

B.14 Interactor

- **Description:** The NPC speaks, it turns to face a target, it moves to the target, it turns to face the target again, and it interacts (performs an interact NWScript action) with the target. If the NPC can interact with the target, an appropriate action will be executed. For example, if the target is a container (a placeable with an inventory), the interaction will open the container if the container is closed and it will close it otherwise.
- **Options:**
 - Target: the object with which the NPC interacts.
 - Text: the text spoken by the NPC before interacting.

B.15 Loiterer

- **Description:** The NPC speaks and it moves around its original location in a random direction and for a random distance within a range specified by the user.
- **Options:**

- Text: the text spoken by the NPC while moving.
- Range: the maximum distance of movement from the NPC's original location.
- Movement: binary option that specifies whether the NPC walks or runs. The default value is set to walk.

B.16 Manipulator

- **Description:** The NPC turns to face a target, it moves to the target, it turns to face the target again, and it performs an animation while speaking.
- **Options:**
 - Target: the object in front of which the NPC performs an animation.
 - Text: the text spoken by the NPC while performing the animation.
 - Pose: the animation that the NPC performs.
 - Duration: the length of the animation performed by the NPC.

B.17 Patroller

- **Description:** The NPC patrols randomly around a set of patrol posts with the same tag as a target while speaking.
- **Options:**
 - Target: the object that provides the tag of the patrol posts to which the NPC moves.
 - Text: the text spoken by the NPC while patrolling.

B.18 Poser

- **Description:** The NPC performs an animation while speaking.
- **Options:**

- Pose: the animation that the NPC performs.
- Text: the text spoken by the NPC while performing the animation.
- Duration: the length of the animation performed by the NPC.

B.19 Rester

- **Description:** The NPC turns to face a target, it speaks, it moves near the target, and it sits on the target.
- **Options:**
 - Target: the placeable on which the NPC sits.
 - Text: the text spoken by the NPC while sitting.

B.20 Returner

- **Description:** The NPC speaks, it turns to face its original location where it was spawned at the beginning of the game, and it moves to its original location.
- **Options:**
 - Text: the text spoken by the NPC while returning.

B.21 Spawner

- **Description:** The NPC speaks, it casts a spell at the location of a target, it displays a visual effect at the location of the target, and it spawns a target creature.
- **Options:**
 - Target: the creature that the NPC spawns.
 - Text: the text spoken by the NPC while spawning the target creature.
 - Spell: the spell cast by the NPC while spawning the target creature.

- Effect: the visual effect displayed while spawning the target creature.

B.22 Spellcaster

- **Description:** The NPC turns to face a target, it speaks, it casts a spell on the target, and it displays a visual effect on the target.
- **Options:**
 - Target: the object on which the NPC casts a spell.
 - Text: the text spoken by the NPC while casting a spell.
 - Spell: the spell cast by the NPC.
 - Effect: the visual effect displayed while casting a spell.

B.23 Striker

- **Description:** The NPC turns to face a target, it speaks, it moves near the target, and it strikes the target repeatedly.
- **Options:**
 - Target: the object that the NPC strikes.
 - Text: the text spoken by the NPC while striking.

B.24 User

- **Description:** The NPC turns to face a target, it speaks, it moves near the target, it turns to face the target again, it performs an animation to simulate the use of the target, it actually uses the target, it performs an animation, it waits, and it uses the target again. If the placeable has an inventory, the use NWScript action opens or closes the placeable, depending on the current state of the placeable.
- **Options:**

- Target: the placeable that the NPC uses.
- Text: the text spoken by the NPC while using the target.

B.25 Vanisher

- **Description:** The NPC turns to face a target, it speaks, and it destructs itself when the distance between the NPC and the target is within a range specified by the user.
- **Options:**
 - Target: the object that causes the NPC to vanish.
 - Text: the text spoken by the NPC while vanishing.
 - Range: a threshold value for the distance between the NPC and the target.

B.26 Wanderer

- **Description:** The NPC speaks and it moves in a random direction and for a random distance within a range specified by the user from its current position.
- **Options:**
 - Text: the text spoken by the NPC while wandering.
 - Range: the maximum distance of movement of the NPC from its current position.
 - Movement: binary option that specifies whether the NPC walks or runs. The default value is set to walk.

B.27 Withdrawer

- **Description:** The NPC removes an item from a target container (a placeable with an inventory) while speaking. The NPC turns to face a target, it speaks,

it moves near the target, it turns to face the target again, it performs an animation that simulates the use of the target, it performs an animation that opens the target, it performs an animation that simulates the action of taking the item, it actually takes the item, and it performs an animation that closes the target. This pattern is similar to the **Dispossessor** pattern. In addition, the NPC performs two more animations that mimic the opening and the closing of the target container.

- **Options:**

- Target: the placeable from which the NPC takes an item.
- Text: the text spoken by the NPC while taking the item from the placeable.
- Item: the item that the NPC takes from the placeable.

Appendix C

Changes to the Arena Module

We modified four NWScript files in Spronck’s arena module v.3 [77] to use the ALeRT algorithm in this framework: *ud_igor_01*, *os_learn_01*, *in_learn_generic*, and *ou_learnlever_02*. In these files, along with our changes (marked with a bold font) that are function calls to our auxiliary file, *i_se_rl_modific*, we display existing framework code to provide context for these changes.

We alternated the creation of the NPCs to eliminate unfair advantages caused by the order of spawning in the game and, subsequently, by the order of attack. In addition, at the end of a phase, the fighters are replaced by new fighters with a different equipment configuration than their previous counterparts, as illustrated in Figure C.1.

```
#include "i_se_rl_modific"
//ActionDoCommand(createOpponent(sWhite,
    GetLocation(oWPWhite)));
//ActionDoCommand(createOpponent(sBlack,
    GetLocation(oWPBlack)));
SCEZ_RL_AlternateNPCCreation(OBJECT_SELF, sWhite,
sBlack, oWPWhite, oWPBlack, nRound);
updateLampposts();
```

Figure C.1: *ud_igor_01* script

We created a new signpost for the new fighter NPC, as illustrated in Figure C.2, to store the rule information for Spronck’s NPC on the previous NPC’s signpost after the phase change. This ensures that learning continues after the phase change, as illustrated in Figure C.3.

We created a new lever switch for the new fighter NPC, as shown in Figure

```

#include "i_se_rl_modific"
//object oStorage = GetObjectByTag("SIGN-" +
    GetResRef(OBJECT_SELF));
object oStorage = SCEZ_RL_GetStorage(OBJECT_SELF);
if (!GetIsObjectValid(oStorage)) return;

```

Figure C.2: os_learn_01 script

```

#include "i_se_rl_modific"
//object oStorage = GetObjectByTag("SIGN-" +
    GetResRef(OBJECT_SELF));
object oStorage = SCEZ_RL_GetStorage(OBJECT_SELF);
if (GetIsObjectValid(oStorage))
    return DetermineCombatRoundLearning(oStorage, iCombat);

```

Figure C.3: in_learn_generic script

C.4, to ensure that Spronck’s NPC uses learning. In Spronck’s system, an NPC is learning only if there is a learning lever provided for that NPC and if the lever is switched on.

```

#include "i_se_rl_modific"
oSwitch = GetObjectByTag(S + "013");
switchLever(oSwitch, iOn);
oSwitch = GetObjectByTag(S + SE_RL_NEW_COMBATANT_ID);
switchLever(oSwitch, iOn);
updateLampposts();

```

Figure C.4: ou_learnlever_02 script

Figure C.5 illustrates the constants that we defined in the `i_se_rl_modific` file in order to implement our changes to Spronck’s code for supporting the integration of our ALeRT algorithm and experiments.

Figures C.6 and C.7 show the learning scripts used by Nera, our agent controlled by the ALeRT algorithm and by Blanche, the opponent controlled by Spronck’s scripts.

```
const int SE_RLEPISODES = 500;  
const string SE_RL_NEW_COMBATANT_ID = "014"; //New Fighter
```

Figure C.5: The constants defined in the `i_se_rl_modific` file.

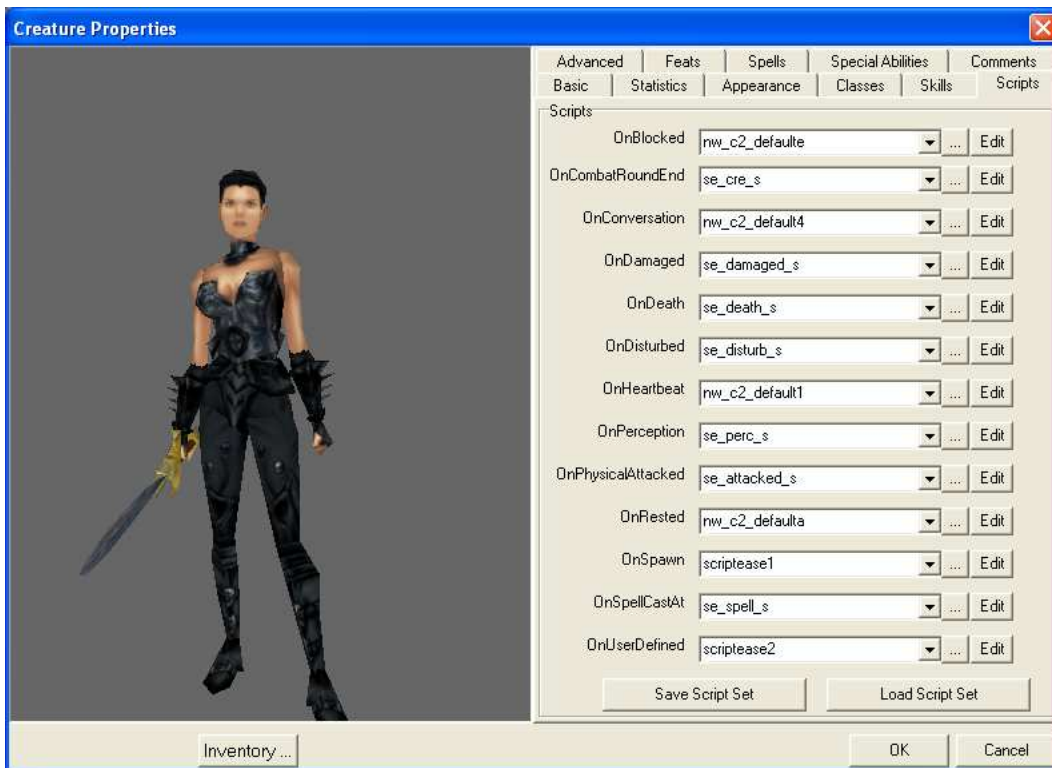


Figure C.6: Nera's Scripts: ALeRT learning algorithm.

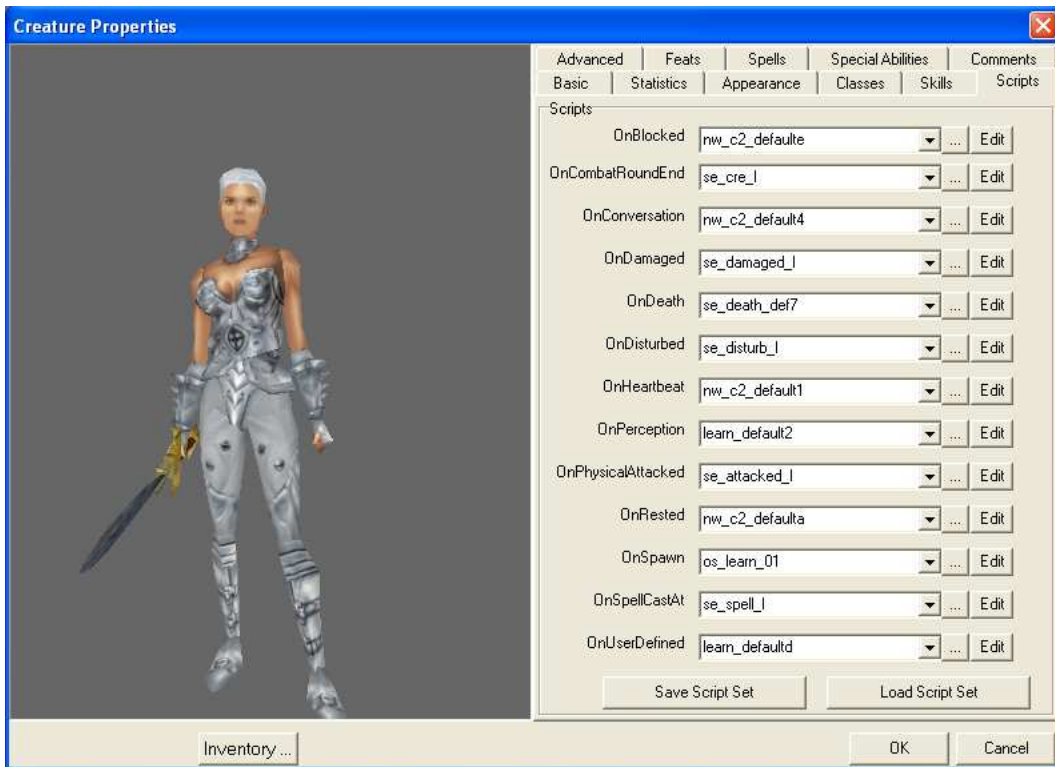


Figure C.7: Blanche's Scripts: Spronck's rule-based learning algorithm.