



Platform for Decoupling Experience Managers and Environments

by
Giulio Mori

Dissertation submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

May 17, 2023

Thesis Committee:

Supervisors:

David Thue, Ph.D.

Adjunct Assistant Professor, Reykjavík University, Iceland
Assistant Professor, Carleton University, Canada

Stephan Schiffel, Ph.D.

Assistant Professor, Reykjavík University, Iceland

Ph.D. Committee:

Hannes Högni Vilhjálmsson, Ph.D.

Professor, Reykjavík University, Iceland

Stephen Ware, Ph.D.

Assistant Professor, University of Kentucky, USA

External Examiner:

Jichen Zhu, Ph.D.

Associate Professor, IT University of Copenhagen, Denmark

© Giulio Mori

May 17, 2023

Publishing Information


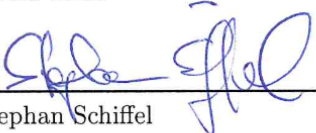

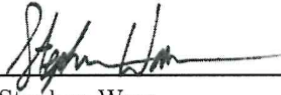

ISBN Print version: 978-9935-539-20-5

ISBN Electronic version: 978-9935-539-21-2

Author's ORCID: 0000-0001-7685-9637

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirements for the degree of Doctor of Philosophy.

Date of Signature

	23/09/2023
David Thue	
	04/10/2023
Stephan Schiffel	
	28/09/2023
Hannes Högni Vilhjálmsson	
	28/09/2023
Stephen Ware	
	27/09/2023
Jichen Zhu	

REYKJAVIK UNIVERSITY

Platform for Decoupling Experience Managers and Environments

Giulio Mori

May 17, 2023

Abstract

Experience Management employs Artificial Intelligence technologies to enhance people’s interactive application experiences by dynamically modifying the environment during the experience. In game-related research, there is a prevailing trend where each experience manager is tightly integrated with the specific environment it can manipulate. This integration poses a challenge in comparing different managers within a single environment or a single manager across multiple environments.

In this dissertation, I propose a solution to address this issue by introducing EM-Glue, an intermediary software platform that decouples experience managers from the environments they can modify. Prior to presenting the solution, I provide a comprehensive problem description and conduct a literature review to explore the current state of the field. Subsequently, I outline the platform’s structural design, including a communication protocol facilitating interaction between managers and environments, as well as the regular communication process.

Additionally, I develop a use case to evaluate the effectiveness of the proposed solution. This involves employing an environment and two experience managers: the *Camelot Wrapper*, a software I constructed to extend the interactive visualization engine *Camelot* and connect it to the platform, *PaSSAGE*, an existing experience manager adapted for use with the platform, and a random experience manager. The evaluation results demonstrate the platform’s ability to decouple experience managers from environments, enabling future work to compare experience managers across multiple environments.

Vettvangur til að Aftengja Reynslustjórnendur og Umhverfi

Giulio Mori

May 17, 2023

Útdráttur

Upplifunarstjórnun notar gervigreindartækni til að auka gagnvirka notkunarupplifun fólks með því að breyta umhverfinu á kraftmikinn hátt meðan á upplifuninni stendur. Í leikjatengdum rannsóknum er ríkjandi þróun þar sem hver upplifunarstjóri er þétt samþættur því sérstaka umhverfi sem hann getur stjórnað. Þessi samþætting veldur áskorun við að bera saman mismunandi stjórnendur innan eins umhverfis eða eins stjórnanda í mörgum umhverfi.

Í þessari ritgerð legg ég til lausn til að taka á þessu vandamáli með því að kynna EM-Glue, milliliðahugbúnaðarvettvang sem aftengir reynslustjórnendur frá umhverfinu sem þeir geta breytt. Áður en ég kynni lausnina gef ég yfirgripsmikla vandamálalýsingu og geri úttekt á bókmenntum til að kanna núverandi stöðu málaflokksins. Í kjölfarið lýsi ég uppbyggingarhönnun pallsins, þar á meðal samskiptareglur sem auðvelda samskipti stjórnenda og umhverfisins, svo og reglubundið samskiptaferli.

Að auki þróa ég notkunartilvik til að meta árangur fyrirhugaðrar lausnar. Þetta felur í sér að nota umhverfi og tvo upplifunarstjóra: *Camelot Wrapper*, hugbúnað sem ég smíðaði til að framlengja gagnvirku sjónmyndunarvélina *Camelot* og tengja hana við vettvanginn, *PaSSAGE*, núverandi reynslustjóra aðlagð til notkunar með pallinum og tilviljunarkenndan reynslustjóra. Niðurstöður matsins sýna getu vettvangsins til að aftengja reynslustjórnendur frá umhverfi, sem gerir framtíðarvinnu kleift að bera saman reynslustjórnendur í mörgum umhverfi.

Preface

This dissertation is an original work by Giulio Mori. I am grateful to have received financial support for this work from both The Icelandic Centre for Research - Rannís (Project Grant #184937-053) and the Reykjavik University Research Fund.

Acknowledgments

I would like to express my deepest gratitude and appreciation to the individuals who have contributed significantly to the completion of my Ph.D. dissertation. Their guidance, support, and expertise have been invaluable throughout this journey.

First and foremost, I am immensely grateful to my supervisors, Dr. David Thue and Dr. Stephan Schiffel. Your unwavering commitment, extensive knowledge, and continuous encouragement have shaped the trajectory of my research. Your guidance has not only refined my academic skills but also instilled in me a passion for scientific inquiry. I am grateful for the trust you placed in me and for the countless hours dedicated to reviewing my work, providing invaluable feedback, and pushing me to achieve my best. Your mentorship has been transformative, and I am grateful for the opportunity to learn from both of you.

I would also like to extend my heartfelt appreciation to the members of my Ph.D. committee, Dr. Hannes Högni Vilhjálmsson and Dr. Stephen Ware. Your insightful comments, suggestions, and constructive criticism have played a crucial role in shaping the direction of my research. I am grateful for the time and effort you invested in evaluating my work and for the thought-provoking discussions that have broadened my perspective.

I would like to express my sincere gratitude to Dr. Jichen Zhu, the external examiner of my dissertation. Your expertise and thorough evaluation of my research have provided invaluable insights and strengthened the rigor of my work. I appreciate the time and consideration you dedicated to reviewing my dissertation.

I am also thankful to the faculty, staff, and researchers at Reykjavík University for fostering an intellectually stimulating environment. The collaborative atmosphere and resources available to me have been instrumental in the completion of my research.

Furthermore, I would like to acknowledge my friends and family for their unwavering support and understanding. Your encouragement, patience, and belief in my abilities have sustained me throughout this challenging process. Your presence in my life has been a constant source of motivation and inspiration.

I am at a loss for words when it comes to describing the incredible support that Eleonora Lanni, my girlfriend, has provided me throughout this journey. Her unwavering belief in my abilities, encouragement during moments of doubt, and constant presence by my side has been nothing short of extraordinary. Eleonora's unwavering support has given me the strength to overcome challenges, pursue my dreams, and achieve milestones I once thought were impossible. Her patience,

understanding, and unwavering faith in me have been the guiding light that has illuminated my path. I am forever grateful for Eleonora's love and support, and I am truly blessed to have her by my side.

Lastly, I would like to express my gratitude to all the individuals who have contributed to my academic journey in various ways, whether through insightful discussions, research collaborations, or personal encouragement. Your influence and support have been instrumental in shaping my growth as a researcher.

Completing this Ph.D. dissertation has been an intellectually stimulating and rewarding experience, and it would not have been possible without the support and contributions of the individuals mentioned above. I am truly grateful for their guidance, encouragement, and belief in my abilities.

List of Tables

4.1	Overview of the papers analyzed in this chapter. The column “J/D Perspective” indicates if the system is using a joint or disjoint perspective [129] in the development of the experience manager.	44
4.2	Overview of all the papers cited during the analysis of the decision constraint function block divided by categories.	49
4.3	Overview of all the papers cited during the analysis of the objective function divided by categories.	55
4.4	Overview of all the papers cited during the analysis of the rollout function divided by categories.	60
4.5	Overview of all the papers cited during the analysis of the estimated player policy divided by categories.	66
4.6	Table legend for abbreviations of table 4.7.	71
4.7	Summary of all the components of each paper.	72
7.1	Matrix of compatibility between the different categories of decision constraint functions from the environment and experience manager. The values in the table can range between “Very High”, “High”, “Medium”, and “Low”. “Very High” indicates that it would be potentially easy to adapt and connect the experience manager to the environment using EM-Glue. “Low” indicates that it would be potentially difficult to adapt and connect the experience manager to the environment using EM-Glue.	176
7.2	Example of the first table of the datasheet taken from the GitHub repository of the Camelot Wrapper.	180
7.3	Example of the second table of the datasheet taken from the GitHub repository of the Camelot Wrapper.	180
7.4	Example of the third table of the datasheet taken from the GitHub repository of the Camelot Wrapper.	181

List of Figures

1.1	A graph illustrating an abstract view of how the player experience can change. The graph is an interactive environment, and every circle represents a state. The double circle represents the player's initial state, and each arrow illustrates how the corresponding player action results in a new state. The states and actions the player perceived and performed as part of their experience are denoted by thick outlines and arrows.	4
1.2	A diagram showing the separation of environment and experience managers.	9
1.3	This diagram shows an hypothetical abstract example of how a game can work with a tight integration between the environment and the experience manager.	10
1.4	This diagram shows an hypothetical abstract example of how a game can work with a separation between the environment and the experience manager.	11
4.1	Flowchart of the literature review process.	32
4.2	A schematic diagram of a GEM manager's policy. Numbers identify GEM's building blocks. Rounded boxes show functions, italics show data, and arrows show function inputs and outputs. The flow of game mechanics is highlighted with thicker arrows. See Figure 4.3 for more details.	45
4.3	A schematic of how GEM's rollout function, objective function, and estimated player policy are used to assess candidate game mechanics (see Figure 4.2 for context).	46
5.1	An overview of our EM-Glue's design. The Experience Manager and the Environment are external modules. The Environment sends messages to update the EM with what is happening in the environment and the EM sends the action that it wants to apply.	80
5.2	An example of a communication between an experience manager and an environment via EM-Glue using the RESTful APIs. This example uses the same messages from Figure 5.1. The icons used are licensed from flaticon.com.	84

5.3	A summary of EM-Glue’s handshake protocol steps. (#) indicates the order of the protocol steps, each involving one message. Each coloured area represents a phase of the protocol.	87
5.4	Architecture diagram of the EM-Glue platform.	91
5.5	Database schema.	92
6.1	A schematic diagram of the Camelot Wrapper’s main components. Double boxes indicate that the component is executed in a separate thread. The arrows indicate the flow of information between components.	106
6.2	A diagram showing the PDDL data framework used to transform the PDDL specification into a Python framework within the Camelot Wrapper.	116
6.3	Algorithm that translates all the items listed in the PDDL problem initial state into Camelot instructions.	122
6.4	Algorithm used to execute a PDDL action message.	125
6.5	Algorithm used to update the world state using a location message.	130
6.6	Algorithm used to update the world state using a success message.	131
6.7	Algorithm used to handle the scene execution.	141
6.8	Screenshot of the game <i>Annara’s Tale</i> taken from Figure 5.9 of Thue (2015) dissertation [127]. Top left: the player’s character, Annara. Top right: Maedorn Forest. Bottom left: dialogue with a non-player character. Bottom right: combat.	143
6.9	A schematic diagram of the PaSSAGE main components. Double boxes indicate that the component is executed in a separate thread. The arrows indicate the flow of information between components.	145
6.10	Algorithm used to handle the main loop of execution in PaSSAGE.	150
6.11	Algorithm used to handle the main loop of execution in the Random experience manager.	153
6.12	Screenshot of the implementation of a conversation in <i>Annara’s Tale</i> taken from the Aurora Neverwinter Toolset. Box #1 shows the conversation. Box #2 shows the action that the environment will execute when the selected line of conversation is chosen.	155
6.13	Visualization of the “Eldon’s Watch” map of the original implementation of PaSSAGE taken from the Aurora Neverwinter Toolset. Box #1 shows the definition of the trigger script. Box #2 shows where the trigger is placed in the map.	158
6.14	Code from the <code>trg_cue_cta</code> script that hosts the start of the encounter selection process of the “Eldon’s Watch” map of the original implementation of PaSSAGE.	158
6.15	Implementation of the <code>init</code> script that defines which type of the player model is more likely to enjoy the encounter <code>mercy</code>	159
6.16	Implementation of the function that selects the encounter that is most likely to be enjoyed by the player based on the player model and the annotations of the encounters.	160

6.17	Screenshots of the execution of the case study. The first column shows the execution of the PaSSAGE script. The second column shows the execution of the environment with the dialogues. Red boxes shows highlights that are relevant to the discussion. On the top right corner of each image there is a number that indicates the order of the execution and I will refer to it in the text.	162
6.18	Database dump of the execution of the case study.	163
6.19	Screenshots of the execution of the case study showing the random experience manager. The first column shows the execution of the random experience manager script. The second column shows the execution of the environment with the dialogues. Red boxes shows highlights that are relevant to the discussion. On the top right corner of each image there is a number that indicates the order of the execution and I will refer to it in the text.	165
7.1	A side by side comparison of the datasheet of the Camelot Wrapper and the PaSSAGE experience manager. The column on the left refers to the datasheet of the Camelot Wrapper, while the column on the right refers to the datasheet of the PaSSAGE experience manager. We can note that the “new entity” message is not present in the datasheet of the PaSSAGE experience manager. This is because the PaSSAGE experience manager does not support the creation of new entities, but the Camelot Wrapper does. In this case, this is not a problem because this message can only occur in response to actions that the experience manager sends to the environment, and PaSSAGE does not send such actions.	182

Listings

5.1	A call of an action using PDDL.	89
5.2	A message that communicates an update of the environment state. .	90
5.3	Class <code>Message</code> of the <code>models.py</code> script. I used classes like this to create the tables of the SQLite database.	92
5.4	Classes <code>MessageBase</code> and <code>MessageCreate</code> of the <code>schemas.py</code> script. These are used to define the valid data shape accepted by the API server.	93
5.5	Query read example of the <code>crud.py</code> script.	93
5.6	Query create example of the <code>crud.py</code> script.	94
5.7	Example of an API endpoint using FastAPI.	95
5.8	Messages that are exchanged during the handshake protocol in JSON format.	95
5.9	The script that governs the handshake protocol for the experience manager.	96
5.10	Function that handles the communication during <code>PHASE_1</code> of the handshake protocol.	97
5.11	Function that handles the communication during <code>PHASE_3</code> and <code>PHASE_4</code> of the environment's handshake protocol.	97
5.12	API call that is used to add a new message from the experience manager during the normal communication.	99
5.13	API call that is used to get the new messages available to the experience manager during the normal communication.	99
5.14	EM-Glue Manager function that handles the handshake protocol. . .	100
5.15	EM-Glue Manager function that handles the <code>PHASE_3</code> of the handshake protocol.	101
5.16	EM-Glue Manager function that handles the <code>PHASE_4</code> of the handshake protocol.	101
6.1	Thread that handles the incoming messages from EM-Glue.	107
6.2	Function used to send a message to EM-Glue.	108
6.3	The <code>send_message</code> method that is used to send messages to EM-Glue.	108
6.4	Camelot Sender thread implementation.	110
6.5	Camelot Receiver thread implementation.	111
6.6	Function that handles the I/O operations with Camelot.	111
6.7	An example of a message that comes from Camelot.	112
6.8	Function that sorts the messages that come from Camelot.	112

6.9	Example of a PDDL entity declaration	114
6.10	Example of a PDDL predicate declaration.	114
6.11	Example of a PDDL action declaration.	114
6.12	Example of a PDDL object declaration.	115
6.13	Example of a PDDL initial state declaration.	115
6.14	Example of a JSON entry file that maps predicates to Camelot instructions.	120
6.15	Function that translates all the items listed in the PDDL problem into “create” Camelot instructions.	121
6.16	Example of a PDDL message that needs to be executed.	123
6.17	Declaration of the <code>move-between-location</code> PDDL action.	123
6.18	Example of an entry of the JSON file for converting PDDL actions into Camelot instructions.	124
6.19	Function that translates a PDDL action message into the action class of the PDDL data framework.	125
6.20	Function that translates an action into Camelot instructions.	126
6.21	An example of a message that comes from Camelot.	127
6.22	Example of another location message that Camelot shares.	128
6.23	Example of a success message that Camelot shares.	128
6.24	Example of an entry of the JSON file that maps predicates to Camelot instructions.	128
6.25	Function that creates and sends Camelot instructions.	131
6.26	Function that starts the communication with EM-Glue.	133
6.27	Function that starts the setup of the game in Camelot.	133
6.28	Function that executes phase three and four of the handshake protocol in the Camelot Wrapper.	134
6.29	Function that handles the input messages from Camelot.	134
6.30	Example of a <i>Yarn Spinner</i> node with options.	136
6.31	Part of the implementation of the <code>__init__</code> method of the <code>Conversation</code> class.	137
6.32	Implementation of the <code>prepare</code> method used for conversations.	137
6.33	Implementation of the method to prepare the messages for the conversation in Camelot.	138
6.34	Implementation of the method used to start a conversation in Camelot.	139
6.35	Implementation of the method that generates the instructions that need to be send to Camelot when a scene is executing.	140
6.36	The function that updates world state in PaSSAGE.	146
6.37	Example of a message that updates the player model.	147
6.38	The <code>PlayerModel</code> class.	147
6.39	Function that handles the handshake protocol in the experience manager.	148
6.40	Function that selects the encounter to execute.	150
A.1	Yarn Spinner conversation.	209
B.1	Example of a desclaration of a Scene in the Scene Controller module of the Camelot Wrapper.	213

Glossary

design pattern A comprehensive, repeatable solution to a frequent issue in software design. 17

drama management an omniscient AI system that monitors the fictional world and influences what happens next, following authorial constraints. 5

encounter an event that an author has annotated with information concerning which player types it would be suitable for [131]. 5, 142

environment a collection of states that an intelligent agent can observe, a variety of actions that an intelligent agent may do, and dynamics – which explain how the intelligent agent’s actions in each state of the environment result in the emergence of new states. 3

experience management the process of optimizing a player’s experience in an interactive environment by changing that environment while the experience is underway. 5

experience manager An AI system whose objective is improve people’s experiences within an application, according to one or more given metrics, by changing the environment while the experience is underway. 5

interactive environment An environment that gives the player the option to choose which action to perform to proceed in the experience. 3

manager action Action that the experience manager uses to change an environment. 5

PDDL domain It defines the essential elements of the problem’s domain and the actions that are available to the agent. 86

PDDL problem It specifies a particular instance of the problem to be solved. 86

quest is a partially ordered set of tasks that the player must complete to get one or more rewards [155]. 6

Acronyms

AI Artificial intelligence. 1

DODM Declarative Optimization-based Drama Management. 24

EM Experience Manager. 5

GEM Generalized Experience Management. 44, 174

GUI Graphical User Interface. 1

GVGP General Video Game Playing. 27

NPC Non-Player Character. 1

PDDL Planning Domain Definition Language. 27, 75

SBDM Search-Based Drama Management. 23

Contents

Abstract	iv
List of Tables	xi
List of Figures	xv
Listings	xviii
Glossary	xix
Acronym	xxi
1 Introduction	1
1.1 Background	2
1.1.1 Environments	3
1.1.2 Experience Management	4
1.2 Problem and Motivation	6
1.3 Domain and Research Focus	12
2 Problem Formulation	15
2.1 Definition of the Core Challenges	17
2.2 Defining the Theoretical Framework	18
2.3 Criteria for Success	19
2.3.1 Reliability and Consistency	20
2.3.2 Support for Interchangeability	21
2.3.3 Practical Considerations	22
3 Related Work	23
3.1 Theoretical Frameworks	23
3.2 Implementation Frameworks	25
3.3 Knowledge Representation	27
3.4 Environments	28
4 Analysis of EM Techniques	31
4.1 Paper Selection Methodology	31
4.1.1 Model for Paper Classification	33
4.2 Analysis of Selected Papers	37
4.2.1 Summary	43
4.3 GEM Framework	44
4.4 Decision Constraint Function	48
4.4.1 Analysis	48

4.4.2	Discussion	52
4.5	Objective Function	53
4.5.1	Analysis	54
4.5.2	Discussion	58
4.6	Rollout Function	60
4.6.1	Analysis	60
4.6.2	Discussion	64
4.7	Estimated player policy	65
4.7.1	Analysis	65
4.7.2	Discussion	69
4.8	Discussion	70
4.9	Takeaways	73
5	Proposed Approach	79
5.1	General Overview	79
5.1.1	Environment	80
5.1.2	Experience Manager	81
5.1.3	EM-Glue	81
5.2	Communication Module Infrastructure	82
5.2.1	Communication Technology	82
5.2.2	Message Exchange Using RESTful APIs	83
5.3	Communication Protocol	85
5.3.1	Communication Language	85
5.3.2	Handshake protocol	86
5.3.3	Regular Communication	89
5.4	Implementation	90
5.4.1	SQLite Database	91
5.4.2	API Server	94
5.4.3	EM-Glue Manager	100
6	Case Study	103
6.1	Camelot Wrapper	103
6.1.1	API Connector	107
6.1.2	Camelot Receiver and Sender Threads	109
6.1.3	Input Multiplexer Thread	112
6.1.4	PDDL Domain and Problem	113
6.1.5	PDDL Data Framework	115
6.1.6	Camelot World State	118
6.1.7	Camelot Action	131
6.1.8	Error Manager	132
6.1.9	Game Controller	132
6.1.10	Conversation Controller	135
6.1.11	Scene Controller	139
6.2	PaSSAGE	142
6.2.1	Implementation	145
6.2.2	Lessons Learned	150

6.3	Random Experience Manager	152
6.3.1	Implementation	152
6.4	Evaluation	154
6.4.1	Joint vs Disjoint PaSSAGE	154
6.4.2	PaSSAGE vs Random EM	163
7	Discussion	167
7.1	Research Questions	168
7.2	Contributions and Benefits	183
7.3	Limitations	185
7.4	Future Work	187
8	Conclusion	189
	Bibliography	193
A	Yarn Spinner Conversation	209
B	Scene Declaration	213

Chapter 1

Introduction

A video game is an electronic game that involves a user interaction with a graphical user interface (GUI) via an input device (e.g., joystick, mouse and keyboard) to generate visual feedback from a display device. Video games are a form of entertainment that has been around for decades, and they have evolved from simple games to complex simulations of real-world scenarios. The fact that video games can simulate real-world scenarios has made them a popular tool for research in many fields, especially in computer science, particularly in the field of Artificial Intelligence (AI). Research on video games and AI can span many different aspects of a game, from developing Non-Player Characters (NPC) that can interact with the player in a realistic manner, to developing intelligent agents that can adapt the experience to designer-specified goals. These intelligent agents are essential components of many gaming experiences, and they can have a profound impact on the game's overall design and player experience. The relationship between these intelligent agents and the game in which they operate is of great importance. Game developers must consider how such agents will operate within the game, what goals they will pursue, and how they will interact with the player and other elements within the game. This relationship is also important for researchers studying AI in games, as it provides insight into how AI that can adapt the player's experience can be effectively integrated into games. In this work, I focus on the relationship between the intelligent agents that manages the player's experience (also called *manager* [102]) and the game in which they operate.

Let us make an example of a game that has an intelligent agent that manages the player's experience. For instance, we can think about the game *Left 4 Dead* [66], where the player is a survivor of a zombie apocalypse and must fight against hordes of zombies to survive. The goal of this game is to survive as long as possible by reaching the next safe area, and the player can achieve this goal by killing zombies and finding supplies to heal themselves. This is made more difficult by the presence of a AI manager, who tracks each player's experience, and adjusts the difficulty of the game by adding or removing items and zombies with the goal of vary the intensity of the experience over time and creating a new experience with each playthrough. Imagine that you are the developer of this game. You have an AI

manager that it is really effective and you would like to use it in another title. However, most likely, your team developed the AI manager in a way that is tightly integrated with the game, thus making it unsuitable for use in other games. As a result, you need to re-implement the AI manager in the new game, which is time-consuming and might not lead to the same behaviour. What could you have done differently to be able to use the same AI manager in different games without having to re-implement it?

Nowadays, reuse of software components is a common practice in software engineering, but historically this is not always been the case. In the early days of software engineering, software was developed in a linear fashion, and the software was not expected to be reused. However, as software became more complex, the need for reuse became more apparent and software engineering started to evolve towards a more modular approach. This evolution led to the development of software components as frameworks, which are pieces of software that can be reused in different contexts. In my research, I aim to apply this concept to the development of intelligent agents that manage the player's experience and the games in which they operate, enabling the reuse of these agents in different games.

In this dissertation, I present EM-Glue, a framework that allows intelligent agents that manage player experiences to be used in different games. I tackle this problem by proposing a protocol that allows the intelligent agent to interact with the game in a way that is independent of the game's implementation. This protocol along with my implementation of EM-Glue are the primary contributions of this dissertation; other contributions include the literature review presented in Chapter 4 and the case study presented in Chapter 6.

This dissertation is organized as follows. In this chapter, I provide the necessary background to understand the rest of the dissertation. I clarify the context of this work and outline the research questions that I will try to answer in this dissertation. In Chapter 2, I present the problem formulation, where I describe the problem I am trying to solve in more specific terms. Chapter 3 discusses other related work that pertains to the problem I am trying to solve. In Chapter 4, I present a literature review of existing intelligent agents that manage the player's experience, to help understand how they were developed. Chapter 5 introduces the framework I have developed to achieve the reuse of intelligent agents. Chapter 6 presents a case study that demonstrates how the framework can be used in a practical scenario. In Chapter 7, I discuss the answers to the research questions and present the limitations and future work using the framework. Finally, in Chapter 8, I present the conclusions of this dissertation.

1.1 Background

This section defines the main terms that are used in this dissertation and provides a brief overview of the history related to these terms. First, I introduce the term "environment" in Section 1.1.1. Then, I introduce the term "experience management" in Section 1.1.2.

1.1.1 Environments

Russell and Norvig (2009) defined environment in the context of intelligent agents as “An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators” [109]. They then defined an environment as something an agent can perceive using sensors and act upon via actuators. This definition could be reflected in an example where we are intelligent agents, we can perceive the world around us using our senses, and we act on this environment via our body. Every time we act in the environment and perceive the state of things, we have an experience. So, a sequence of moments that we live in our daily life is an experience, and we live that experience in an environment.

If we adapt the definition that Russell and Norvig wrote to the context of this dissertation, we can say that three things, in general, make up an *environment*: a collection of states that an intelligent agent can observe, a variety of actions that an intelligent agent may do, and dynamics – which explain how the intelligent agent’s actions in each state of the environment result in the emergence of new states [127]. If we analyze the situation that we are living in now, the environment is where we are located at the moment (e.g., an office); the state is composed of all the things that surround us (e.g., the items and their location) and what we are doing (e.g., reading this dissertation), we have a finite number of available actions (e.g., we can pick up a pen, but we cannot fly a plane), and based on the action we perform next, we will change the state. We can apply this reasoning also while we play video games because we are observing the states of the virtual world via the computer’s screen, the game provides us with all the available actions, and we use input devices to apply actions to the virtual world. In this case, the environment is the virtual world where the game is being played, and we are the intelligent agents that play that video game, or in other words, we are the players. As a result, the player and the NPCs are agents that interact with the environment, and they are part of the environment. Throughout this dissertation, when I write about an environment, I refer to a virtual environment in the context of a video game.

Whenever the environment gives the player the option to decide which action to perform to proceed in the experience, we can consider that environment as an *interactive environment*. For example, the interactive environment represented in Figure 1.1 is composed of four states $\{1, 2, 3, 4\}$ (state 1 is the starting state), three player actions $\{A, B, C\}$, and dynamics such as action A leads from state 1 to 2, action B from state 2 to 3, and action C from state 2 to 4.

In an interactive environment, how the experience plays out depends on three factors: the starting state, the player’s actions, and the dynamics of the environment. The starting state is the state where the player starts the experience, the player’s action is the action that the player performs to change the state, and the dynamics of the environment are the rules that define how the player’s action changes the state.

Snodgrass et al. (2019) identifies that two parts compose an environment: a *Physical* environment which contains all the characters and objects, and the *Narratological* environment which is characterized by the story or narrative that the player participates in during gameplay [119]. The physical and narratological en-

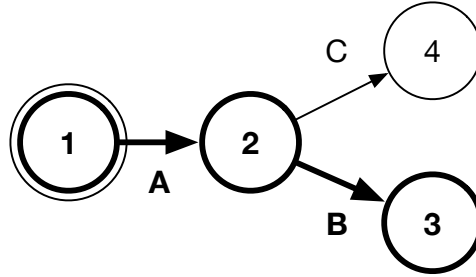


Figure 1.1: A graph illustrating an abstract view of how the player experience can change. The graph is an interactive environment, and every circle represents a state. The double circle represents the player’s initial state, and each arrow illustrates how the corresponding player action results in a new state. The states and actions the player perceived and performed as part of their experience are denoted by thick outlines and arrows.

vironment are interconnected and influence each other. In fact, by interacting with the physical environment, the player can influence the narrative; for example, by killing a character, the player can change the story. In the same way, the progress of the narrative can influence the physical environment. When the player is progressing the game and following a story, the system in charge of the narratological environment can change the physical environment to reflect the progress of the story. For example, the story can require the player to go to a specific location, and the system can change the physical environment to make the player’s goal more visible.

1.1.2 Experience Management

Experience Management is a subfield of artificial intelligence that studies intelligent systems with the name of “experience managers”, which objective is to improve people’s experiences within an application according to one or more given metrics. In the context of game-related research, researchers have often referred to experience management as drama management. One of the first definitions can be traced back to Laurel (1986). She designed an AI system called the PLAYWRIGHT which, utilizing a designer-provided knowledge base and the history of the interactive environment, would generate a temporally ordered sequence that describes the next moments of the interactive simulation. Weyhrauch (1997) developed Laurel’s work by designing an AI manager named MOE. With this work, he extended the PLAYWRIGHT’s capabilities with two key features. First, he represented drama management as an optimization problem, where the manager determines the quality of any given narrative using one or more measures or estimates, such as the player’s level of engagement. Second, he included the idea that a manager might want to change not only the immediate course of the story, but also its behaviour at some potential future states. Weyhrauch considered drama management similarly to game-tree search: a player performs an action within the environment, and the

drama manager responds with some moves by performing a game-tree-like search to maximize an evaluation function. The evaluation function was the encoding of the authors’ belief of what an excellent interactive experience looks like, and it provided the drama manager with objectives to achieve. Weyhrauch’s work led to a representation of EM called Search-Based Drama Management (SBDM) [82, 79]. More generally, *drama management* usually entails an omniscient AI system that monitors the fictional world and influences what happens next, following authorial constraints [103, 153].

Other possible ways to represent experience management systems can be found in prior work on dynamic difficulty adjustment [44], adaptive game mechanics [52], player-adaptive games [38], procedural game adaptation [128], and generalized experience management [127]. “Experience management” can be seen as the hypernym of these terms, since it derives from the generalization of these lines of research where all the fields mentioned above target a specific type of experience. Riedl et al. (2008) first defined the term *experience manager* (EM) as an “agent that attempts to coerce a virtual world so that a user’s interactive experience conforms to a set of provided properties” [102].

I define *experience management* as Thue (2015) did: “the process of optimizing a player’s experience in an interactive environment by changing that environment while the experience is underway” [127]. This means that these experience managers will change the environment by executing actions with the objective of optimizing a user’s experience while the game is being played. An action that the manager uses to change the environment is called *manager action*. The EM has to be guided during the optimization process to choose the best possible manager actions to achieve a specific objective by using metrics that focus on a particular aspect of an experience. For example, suppose the manager’s objective is to improve the player’s enjoyment of a video game. In that case, a possible solution would be that each manager’s action that the EM can execute is annotated with data to estimate the player’s enjoyment so that the manager can make an informed choice.

PaSSAGE [131] is an example of an experience manager that changes the dynamics of the environment to adapt the player’s experience. It uses a player model that can keep track of the player’s preferences into five categories based on the actions that they choose to play. The player model is formed during the dialogue sessions and based on the replies the player gives to the other NPCs in the game. Once the player model has some data, the manager decides which encounter the player should play next based on the player preferences. An *encounter* is an event that an author has annotated with information concerning which player types it would be suitable for [131].

Experience managers can influence the player’s experience in a variety of ways that depend on how they influence an interactive environment. As mentioned in the previous section, an environment can be influenced in three ways: (i) by changing the starting state, (ii) by changing the player’s available actions or (iii) by changing the dynamics of the environment. In this dissertation, I will focus on experience managers that influence the environment by changing the player’s available actions. These experience managers have many different ways with which they can influence

the player’s experience. One instance is when they directly modify the environment by performing actions that alter its state, such as shutting a door that would lead to a dead end. Through this method, the manager affects the player’s experience by limiting their options and directing them towards a different part of the game. They can also influence the player’s experience more indirectly by changing the availability of particular actions or quests. For instance, the manager can alter the player’s available quests by hiding quests that are not aligned with the player’s preferences in a secluded area of the game world. A *quest* is a partially ordered set of tasks that the player must complete to get one or more rewards [155].

1.2 Problem and Motivation

Over the years, many experience managers have been developed to propose solutions to improve people’s experience inside an environment according to different metrics. However, in literature, we can find a trend where researchers designed experience managers and environments in a way that is difficult to distinguish which part of the code base is the environment and which one is the experience manager. In fact, many of experience managers are tightly coupled with the environment they are designed for, and they are not easily portable to other environments. This is a problem because it makes it difficult to compare the performance of different experience managers, and reuse experience managers in different environments.

Roberts and Isbell were the first researchers that brought to light this issue. In their paper, they made a qualitative analysis based on a list of desiderata of different approaches to drama management [103]. To the best of my knowledge, this was the first attempt of evaluating and comparing different systems to one another. They also shed light on two problems concerning the evaluation processes in drama management: dependency to the content, and integration between drama managers and games. First, the evaluation of such systems is highly dependent on the content of the narrative/game. When surveying people to understand the potential effectiveness of a certain system, if the narrative is boring, the answers of the person being tested will be influenced by the narrative. This situation can lead to problems in the evaluation. An example is in Ramirez and Bulitko’s (2015) work [93], where they tried to assess the effects of adding player modelling to a planning-based experience manager. They repeated the “user study 2” twice without finding statistical significance in the results, concluding that a severely unbalanced distribution of the story content is a factor in the user reports of the perceived fun and agency [93] and it can influence the evaluation. The outcome was that they needed to repeat the evaluation for the third time but with a scaled-down narrative space to reduce the player’s focus on the story.

The second problem that Roberts and Isbell described is that some of the systems they surveyed are “integrally tied to a particular game system” [103]. In fact, a common practice among both academia and industry professionals has been to build a new, ad hoc testbed for each new experience manager that they create [147, 60, 84, 8, 148, 57, 99, 82, 80, 26, 10, 115, 114, 102, 5, 9, 104, 131, 133, 134, 90, 107, 136, 121, 145, 95, 92, 93, 42, 105]. This issue is connected to the first problem described, but also raises another, concerning the reproducibility and

replicability of the results.

I do not know the exact reason why this tight integration between an EM and its environment is so popular in the literature. Still, I can speculate that it derives from the fact that, usually, experience managers need to have fine-grained control of a game environment to adapt the user experience. Experience managers are designed to improve a particular part of an experience (e.g., drama), and the game environment plays an essential role in the user's perception of the particular aspect that the EM is targeting. So, researchers, in some cases, choose to have full control of the environment and also to control what the user uses to interact with the EM. As a result, this choice leads to an unification between the EM and the environment particularly in the code base. In my opinion, another aspect also plays a role in this diffused integration: the absence of tools that facilitate the making of experience managers in a way that keeps them detached from the environment without too much development work.

At first glance, this integration does not seem like a problem because the field has achieved great results from its developments. However, if we analyze it more closely, we start seeing some complications, especially related to the evaluation of such systems [103, 77]. These problems can be summarized in three macro categories: (i) the lack of generalization during the evaluation of experience managers, (ii) the difficulty of comparing multiple experience managers, and (iii) the lack of experience managers available for the community to use.

First, since the results of testing EMs depend on the content of the environment used [103], testing in a single environment gives insufficient data to ensure that any findings will be replicable in a different environment. To obtain more generalizable results, it would be helpful to test an EM in a diverse set of environments and combine that data to better understand how the manager works. However, when an EM is tightly integrated with an environment, it is difficult to use that EM in another environment. In fact, to do so, we would need to modify (or in the worst case, rewrite) the EM's code since the tight integration causes the EM to be dependent on the environment's code. This same reasoning applies the other way around: if we want to use an environment with another EM, we need to modify the environment's code to make it compatible with the new EM.

Second, the tight integration of each EM and its environment makes it challenging to experimentally compare multiple EMs, since doing so requires separating $n - 1$ of them from their initial environments and connecting them to the n^{th} EM's environment for testing. This process is time-consuming and requires a lot of effort to be done for each EM because of the reason that I mentioned before: if we want to use an EM in a different environment or an environment with different EMs, we need to modify some parts of the code base to make them compatible. An example is Ramirez and Bulitko's (2015) work, where they had to re-implement the EMs of both the *Automated Story Director* [102] and *PaSSAGE* [131] to use them in a novel environment. This makes performing such comparisons difficult, discouraging researchers from undertaking the work to do them and making it challenging to know whether any new EM advances the state of the art.

Third, another barrier to comparing multiple EMs is that very few are released after the publication of their related research papers. There can be many reasons

why EMs are not released, but we can speculate that one of them is the tight integration between the EM and the environment. This reason is connected to the fact that releasing an EM would mean releasing the environment as well and this is not always possible. If a manager is not released with the paper, another researcher that wants to use it in their own work for comparison must reimplement it interpreting the paper's description of the manager's behaviour. With a reimplementation comes the risk of a wrong interpretation of the paper's text, which can introduce bugs and errors that may affect the results of the comparison. Lately, we started seeing signs towards a more open approach, where researchers release their EMs and environments to the community [139, 156]. However, even if these systems are released, they are still tightly integrated with their environments, which makes it difficult to use them in other environments.

These three barriers to comparing multiple EMs and generalizing results while testing EMs could be solved by creating a separation between experience managers and environments. This separation would allow researchers to test EMs in different environments and compare them without having to modify the code of any of its parts. However, this would require that multiple elements are in place to make this separation possible. Before I start the discussion of what we need to separate EMs and environments, I should first clarify what I mean by separation.

Thue and Bulitko (2018) can help us identify this separation with their work on the *joint/disjoint perspective*. The disjoint perspective views the game and the manager as separate elements, with the manager changing the environment as the player's interaction with the game progress. The joint perspective views the game and the manager as a single entity, which the player interacts with as they would under the disjoint perspective [129]. In their work, Thue and Bulitko show that in literature the most common way of talking about EMs and its environment is using the disjoint perspective. However, in practical terms, during the development researchers decide to implement EMs and environments using a joint perspective (as I show in Chapter 4).

Decoupling experience managers and environments is the process of reducing the dependencies that an experience manager has with the environment where it was tested. Using the context of Thue and Bulitko's (2018) work, we can say that decoupling EMs and environments is the process of converting experience managers and environments from the joint perspective to the disjoint perspective. Figure 1.2 gives a high-level representation of what this means showing the difference between a game developed using a tight integration and one where manager and environment are separated.

On the left side of Figure 1.2 we have a representation of a game with the environment and the experience manager are tight together. This integration can also be referred to as the *joint perspective* of Thue and Bulitko's (2018) work. When an experience manager is developed with this integration, the two components intersect since the EM directly controls parts of the environment and the environment controls part of the experience manager. In this intersection, it is difficult to understand which parts of the functionalities belong to the EM and which to the environment. So, if we needed to export this EM to another environment, a good part of the code would require adaptation to work. Meanwhile, on the right side of

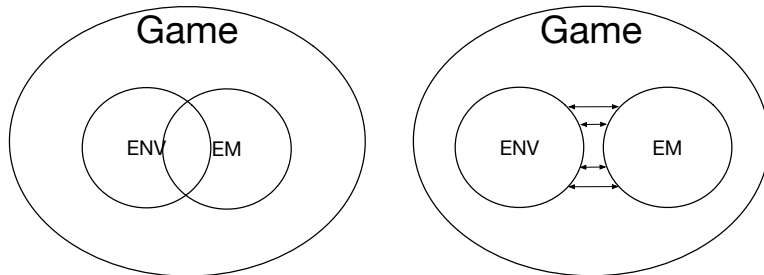


Figure 1.2: A diagram showing the separation of environment and experience managers.

Figure 1.2 there is a high-level representation of a game where the environment and experience manager are separated (Thue and Bulitko’s disjoint perspective). We can notice the difference between the two parts of the figure in that the intersection of the two components is substituted by arrows that indicate communication. Suppose the communication between EM and the environment follows some specified rules. In that case, the process of changing environments or EMs could become a matter of just setting up a new connection between the two parts.

To understand how both methods work, I now show an example of the two levels of integration in a hypothetical use case to understand how both methods work. Some parts of the text are underlined to highlight that they are the components of Figure 1.3 and 1.4. Imagine a game where the environment is a game engine (e.g., Unity) that controls the GUI, NPCs, and keeps track of what the user is doing via a game state. Meanwhile, the experience manager (EM) chooses how the story should progress based on the world state, and it has the power to instantiate new objects or NPCs that are available in the assets. During gameplay, the player decides to kill an important NPC for the story, and in response to this player action, the EM decides to spawn a new NPC that takes the role of the dead actor.

Figure 1.3 shows the joint perspective of the example. In a use case where the environment is developed with tight integration with the EM, the EM has direct access to the world state and directly notices the death of the NPC based on the player’s actions. In response, the EM creates a new NPC using the game assets and assigns the role of the dead character. This process is possible because the EM has direct access to the environment and the world state. This tight integration makes it difficult to understand which are the roles of the EM and the environment because actions that should belong to the environment (e.g., instantiating a new NPC) can be performed directly by the manager. Suppose that in the future we would want to extract this manager to use in another environment. In that case, we would need to modify the code of the manager to make it compatible with the new environment (e.g., the NPC that needs to be instantiated has a different name).

Figure 1.4 shows the disjoint perspective of the example. In this use case where the environment and the EM are detached, the EM receives from the environment the communication via the updated world state that the NPC is dead with all the data that the EM needs to understand what happened. In response, the EM asks

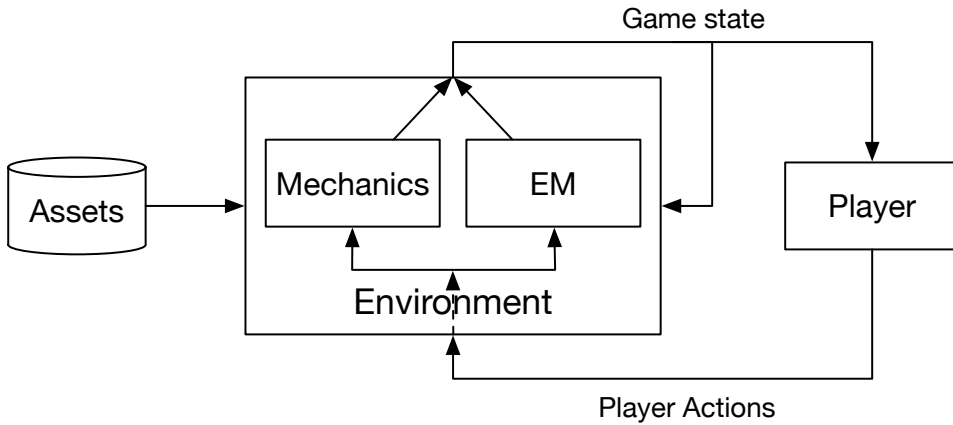


Figure 1.3: This diagram shows an hypothetical abstract example of how a game can work with a tight integration between the environment and the experience manager.

the environment to create a new NPC with an *EM action*, and it internally assigns the role of the previous character to the newly created one. When environment and EM are separated, we add some steps to the process since the environment needs to communicate with manager an updated world state, and then the manager needs to ask the environment to perform an action. However, with this separation environment and EM are more independent with clear roles and duties, and the process of changing environments or EMs becomes easier.

This separation between experience managers and environment can be both conceptual and practical. To start, we can define the conceptual separation between the two components as follows.

Definition 1.2.1 (Conceptually decoupled). An experience manager and an environment are *conceptually* decoupled if the author of the system views the two components as separate entities, where the manager modifies the environment while the experience happens.

Definition 1.2.1 follows closely the definition of Thue and Bultko’s disjoint perspective [129]. In fact, Thue and Bultko defines the disjoint perspective as a conceptual separation between the game and the manager. However, in practice, the designers of experience managers do not always implement this separation in practice even if they define it theoretically. So, we can define the practical separation between the two components as follows.

Definition 1.2.2 (Practically decoupled). An experience manager and an environment are *practically* decoupled if the author of the system implements the two components as separate software entities.

The objective of Definition 1.2.2 is to distinguish between conceptual and practical separation. In the literature, it is relatively common to find systems that are

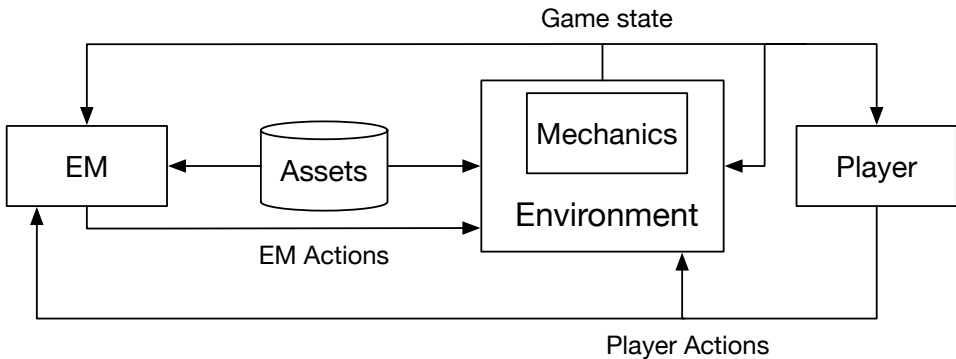


Figure 1.4: This diagram shows an hypothetical abstract example of how a game can work with a separation between the environment and the experience manager.

conceptually decoupled, but in practice, it is difficult to understand the separation (if any) between the implementation of the two components.

As we can infer from the examples, the key element that facilitates the separation of the EMs and environments is the communication between the two parts. The communication should be defined such that the EM can understand the information that the environment sends and viceversa. If we can create a set of rules that both EMs and environments can follow during communication, then we obtain separation and interchangeability between them. In fact, if an environment follows a set of rules to communicate, then any EM that follows the same rules can understand the information that the environment sends and act upon that information to adapt the experience.

In theory, once these rules are defined and accessible to the community, researchers can start to develop EMs and environments that use them. This would allow the researchers to create new EMs and environments that can be used interchangeably. However, in practice, just defining the rules to follow is not enough to achieve this goal. There is also the need to create a set of tools that facilitate the community to develop EMs and environments towards the adoption of these rules. If the process of adopting the approach to separate EMs and environments is too complex, then the researchers in the community will prefer to continue developing their own EMs with the tight integration with environments. So, with the word *facilitate* I refer to assisting the researchers in the community by providing open source tools, guides, and tutorials that simplify the process of adopting the approach.

This problem of separation is relevant to the game development and research community because developing EMs and environments with interchangeability in mind can help to create a more robust and flexible ecosystem of tools that can be used to create games. This is particularly important because the game industry is fast-paced and requires the creation of new games in a short amount of time. The ability to reuse and interchange experience managers and environments can reduce the time needed to create new games. Another benefit of a decoupled ecosystem is

that one can test multiple experience managers in the same environment (with no added development time) to see which one performs better. This is useful because if we understand which experience manager works better in the specific environment, the player will have a better experience thus increasing the game's player retention.

The interchangeability of EMs and environments in academia is also important for different reasons. First, having a decoupled ecosystem of EMs and environments can encourage the comparisons between different experience managers in a single environment since it would require less development work compared to the current standard. This is useful because it can help evaluate the performance of the experience manager developed by showing how it compares to other experience managers. Another benefit is that researchers can spend more time on experience managers and less time developing the environment [117]. This can reduce the time needed to develop an experience manager and thus encourage more research in experience management. Lastly, researchers can test an EM in multiple environments to see how it performs in different settings, opening new lines of research that were mainly unexplored before.

1.3 Domain and Research Focus

The main focus of this dissertation is on experience managers, especially how they relate to computer video games. Video games are the perfect test bench for experience managers because they are complex systems that can simulate dynamics in real-life situations. Using a virtual environment as host for an experience manager allows us to test the AI manager's behaviour in a controlled environment where we can have a complete awareness of the environment. Video games are also beneficial as an evaluation tool when investigating the effectiveness of experience managers for multiple reasons. First, they can be designed with a specific purpose in mind, which allows us to test an experience manager in a specific context. Game designers, when creating a game testbench, frequently strive to target specific impacts on players (such as creating sentiments of enjoyment, irritation, or excitement), making them suitable models for AI managers. Second, the virtual nature of video games makes it simpler to use as test-bed of these AI systems because there is no need to create interfaces for robotic arms, manual switches, or other "real world" devices, and this helps streamline the logistics of carrying out this kind of research. Finally, the potential advantages of using AI experience management in video games are also worthwhile from a business standpoint, since raising player enjoyment might have a direct, favourable effect on a game's reception, lifespan, and sales. Different commercial video games include an AI manager in their content to create a more engaging experience for the player [65, 66, 64]. For example, in *Mario Kart 64* [65] the AI manager is responsible for the control of the NPCs in the game and the spawning of the boosters, and in *Left 4 Dead* [66] the AI manager is responsible for the behaviour of the zombies. However, usually the scope of such systems is limited to specific functionalities, since the AI manager is often developed as a part of the game engine, and it is not possible to reuse it in other games.

In doing this research, I was particularly interested in the relationship between the interactive environment and the system in charge of managing player expe-

riences in settings where the manager may watch the environment's states and actions as they take place. These managers study and understand the preferences of each player, which will help them become better at enhancing each player's experience. Because of this, I posit that every experience we discuss throughout this dissertation follows these three characteristics: (i) it takes place in an interactive environment; (ii) the manager may monitor the states and activities that take place there; and (iii) the manager can shape how the experience takes place.

With this dissertation, I want to investigate if it is possible to create a set of rules and protocols that facilitate the separation of the experience manager from the environment with the objective of interchangeability between them. The benefits of this approach are twofold. First, it allows us to create more flexible EM's system that can be adapted to different environments. This can be useful for game developers that want to create a game with a specific experience manager but do not want to develop it from scratch. Second, it enables the possibility to make comparisons between different experience managers in the same environment with less development work. This feature could be used in the initial stages of planning a game to test different experience managers and choose the one that best fits the game.

The questions that I want to answer with this dissertation are the following:

1. What are the key obstacles that must be overcome to accomplish the separation between experience managers and environments? With this question, other sub-questions arise:
 - (a) What are the constraints that need to be taken into account when developing the communication framework between the experience manager and the environment?
 - (b) What are the characteristics that an experience manager needs to have to be able to work in different environments?
 - (c) What are the characteristics that an environment needs to have to be able to work with different experience managers?
2. Based on the characteristics and constraints that I have identified in the previous question, can the communication be facilitated in a way that does not require changing the internals of the experience manager or environment? If the answer is yes, then other sub-questions arise:
 - (a) What data needs to be exchanged between the experience manager and the environment?
 - (b) What are the key components necessary to create a protocol that is general across experience managers and environments?
3. How does using the platform impact the constraints experienced by developers during the implementation process? Also with this question, other sub-questions arise:
 - (a) What are the steps that developers of an existing experience manager, initially developed using a joint perspective, need to follow to achieve a

disjoint perspective implementation while adhering to the framework's rules and protocols?

- (b) What are the steps that developers of an environment need to follow to achieve a disjoint perspective implementation while adhering to the framework's rules and protocols?
4. How can a designer anticipate what work is needed to connect an experience manager to an environment using the framework? Also with this question, another sub-question arises:
- (a) How can a designer recognize when an experience manager and an environment can be connected using the framework?

Chapter 2

Problem Formulation

The problem I want to address with this dissertation is that the tight integration in the development of experience managers and environments does not allow interchangeability. By interchangeability, I mean that any experience manager should be able to be connected and used in any environment without the need to change the code of the experience manager or the environment.

First-order logic can help us to define interchangeability formally. We can divide the definition of interchangeability into two parts: for experience managers and for environments. Suppose that M represents the set of all experience managers, E represents the set of all environments, and $connect(m, e)$ means that an experience manager ($m \in M$) and an environment ($e \in E$) can be connected without the need to change the code of the experience manager or the environment. Then, we can define interchangeability for an experience manager m in the context of all the environments E , $Int_{m,E}$, as follows:

$$Int_{m,E}(m) \iff \forall e \in E, connect(m, e) \quad (2.1)$$

We can read Equation 2.1 along these lines: a given experience manager m is interchangeable with the environments in E if and only if it can be connected to all environments in E without the need to change the code of the experience manager or the environments.

Similarly to Equation 2.1, we can define the function interchangeability of an environment e in the context of all experience managers M , $Int_{e,M}$, as follows:

$$Int_{e,M}(e) \iff \forall m \in M, connect(m, e) \quad (2.2)$$

We can read Equation 2.2 as follows: a given environment e is interchangeable with the experience managers in M if and only if it can be connected to all experience manager in M without the need to change the code of the experience manager or the environment.

If we combine the interchangeability for experience managers and the interchangeability for environments, we can define full-interchangeability $FInt_{M,E}$ in this way:

$$FInt_{M,E} \iff \forall m \in M, Int_{m,E}(m) \iff \forall e \in E, Int_{e,M}(e) \quad (2.3)$$

Equation 2.3 states that full-interchangeability $FInt_{M,E}$ is true if and only if all experience managers in M are interchangeable with all environments in E (as defined by Equation 2.1) and all environments in E are interchangeable with all experience managers in M (as defined by Equation 2.2). In other words, it states that full-interchangeability is achieved if every experience manager can be connected to any environment and every environment can be connected to any experience manager, without the need to change the code of either the experience manager or the environment. Equation 2.3 is composed of two implications because the fact that any manager is interchangeable with the set of all environments implies that any environment is interchangeable with the set of all managers, and vice versa.

Full-interchangeability represents an ideal: it could be achieved in a perfect world with unlimited time and resources. However, in practice, we can only achieve this degree of freedom with *some* environments and managers that exist or will be developed in the future, since achieving full-interchangeability would require adapting the code of all managers and environments to work with the interchangeability model, which is not feasible. Nevertheless, we can start studying the problem and find a solution that can be applied to a constrained subset of managers ($M_c \subset M$) and environments ($E_c \subset E$). So, we can change Equations 2.1, 2.2, and 2.3 to define partial interchangeability $PInt$ in this way:

$$PInt_{m,E_c}(m) \iff \forall e \in E_c, connect(m, e) \quad (2.4)$$

$$PInt_{e,M_c}(e) \iff \forall m \in M_c, connect(m, e) \quad (2.5)$$

$$PInt_{M_c,E_c} \iff \forall m \in M_c, PInt_{m,E_c}(m) \iff \forall e \in E_c, PInt_{e,M_c}(e) \quad (2.6)$$

To simplify my writing, in the rest of the dissertation, I will use interchangeability to refer to partial interchangeability (Equations 2.4, 2.5, and 2.6) since it is the most realistic goal that can be achieved in the near future.

To achieve the goal of partial interchangeability, there are two challenges that need to be addressed:

- **Clear boundaries:** the experience managers and the environments should be developed as standalone modules that can be shared and used in other settings. The term “standalone” refers to EMs and environments that can be used independently of which other part is used. Suppose there are two experience managers A and B and two environments C and D . If $A, B, C,$ and D are standalone, then it should be possible to use A with C and D and B with C and D without changing the code of $A, B, C,$ or D . Additionally, managers and environments need to be modular and flexible, allowing them to be easily connected and used in different contexts without changes to their code.
- **Structured external communication:** the external communication between the experience manager and environment should follow a structured set of protocols. Each message sent between the two should be defined in a way that both parties can understand. These protocols and rules must be shared in a

comprehensive and accessible way (e.g., open source schematic, well-known technologies). This is particularly important because an EM or environment that does not follow a shared protocol for its external communication cannot be used with other components that use the shared protocol without additional work.

The first challenge is not trivial because it requires that the experience manager and the environment are decoupled from each other using Thue and Bulitko's disjoint perspective both theoretically and practically. However, as discussed in the previous chapter, the tight integration in the code base between the experience manager and the environment is a common practice in industry and academia. So, to achieve this first challenge, a new design pattern is needed to shift the trend towards decoupled experience managers and environments.

The second challenge is also not trivial because creating a protocol that regulates the diverse use cases that experience managers and environments can have requires an in-depth analysis of the literature and trade-offs of the kind of aspects that the protocol can regulate. An in-depth literature analysis is necessary to identify the most common ways that experience managers are developed. This is useful for understanding the widespread components that developers use to create experience managers and identifying the characteristics that the protocols should have.

Proposing a possible solution to this complex problem requires a multi-step approach. First, in Section 2.1, I define the core challenges that I need to solve for achieving the decoupling and interchangeability of experience managers and environments. Second, in Section 2.2, I specify the theoretical framework behind a design pattern that can decouple experience managers and environments. Third, in Section 2.3, I define the criteria that the design pattern and framework needs to meet to be considered successful.

2.1 Definition of the Core Challenges

Creating a general solution for decoupling experience managers and environments requires overcoming some challenges. First, there is no common way of implementing experience managers or environments. In literature, we can find different theoretical frameworks for representing experience managers [81, 79, 103, 127, 129]. However, there is no work I could find related to implementation frameworks specific to EMs (more details in Chapter 3). To achieve the goal of interchangeability, a design pattern needs to be defined so that it can be used during development. A *design pattern* is a comprehensive, repeatable solution to a frequent issue in software design [32]. In this case, the issue that the design pattern needs to address is the tight coupling between experience managers and environments. The design pattern needs to be general enough to be used in different scenarios that cover a subset of common use cases.

The second challenge that I need to address is the different ranges of integration between EMs and environments, which makes it hard to understand the role of the manager in controlling the environment. We can classify interactive environments

into two groups: environments that require a manager's intervention to proceed and those with built-in logic and can (potentially) be used without a manager. An environment of the first type requires the manager to make all the decisions needed to have a meaningful experience. The manager's duty starts with setting up the environment with all the things that an experience needs by creating the locations, instantiating the characters, spawning the items, and organizing everything logically for the player. Once the initialization phase is finished, the manager needs to manage other aspects of the game, such as the interaction between characters and items (e.g., the player wants to open a door, the manager has to open that door) and the interaction between characters (e.g., controlling the NPC). Once all these aspects are sorted out, the manager can start high-level reasoning on improving the user's experience in the game. Meanwhile, in an environment where all these logics are built within the environment itself, the manager needs to focus only on the high-level reasoning for improving the player's experience. An experience manager designed to work in one environment radically differs from an experience manager designed for another. To resolve this challenge, some trade-offs need to be made when defining the scope of the design pattern that will be proposed.

In general, if we want a solution for the interchangeability of a subset of EMs and environments, we need to establish some constraints to narrow down this subset from the overall set of possible EMs and environments. These constraints define a set of characteristics that the EMs and environments need to work with the design pattern. However, to determine these characteristics, we need to understand some aspects of the development of EMs and environments such as: the most common components used in the implementation, the kind of tasks that they need to perform, and the data needed to carry out these tasks. To answer these needs and analyze the current state of the art in the development of EMs and environments, I worked on a literature review that is presented in Chapter 4.

2.2 Defining the Theoretical Framework

A thorough theoretical foundation is necessary to create a strong, adaptable, and interoperable ecosystem of experience managers and environments. This foundation would define the rules and protocols the two components must follow to operate effectively in a decoupled ecosystem. To this end, I propose the following four key components of the theoretical framework: data standards, communication protocols, and testing and validation procedures. Together, these elements can provide a solid foundation for achieving interchangeability and promoting the growth of an interchangeable ecosystem of experience managers and environments.

The first key component of a theoretical framework is a set of shared data standards that experience managers and environments which want to operate interchangeably must adhere to. These standards need to specify the format and structure of the data that the two components can exchange, as well as any rules for data validation and error handling. A data standard is necessary because it ensures that experience managers and environments can exchange data consistently and predictably. Without a data standard, different components may use different formats and structures for their data, which could lead to confusion and errors

when attempting to exchange information. A data standard provides a common framework for data representation and communication, which allows experience managers and environments to interoperate effectively and efficiently. Additionally, a data standard can ensure the integrity and quality of the data being exchanged by providing rules for data validation and error handling.

Once the data standards are defined, the next step is to characterize a communication protocol that experience managers and environments can use to communicate with each other. These protocols would specify the rules for sending and receiving messages and the procedures needed for establishing and maintaining a connection. A communication protocol is important because it enables experience managers and environments to exchange information and data consistently and reliably. Without a communication protocol, the two components would not have a standardized way of sending and receiving messages, which could lead to confusion and errors when attempting to communicate with each other. The protocol provides a common communication framework, allowing experience managers and environments to interoperate effectively and efficiently.

The final key component of the theoretical framework is a set of procedures for testing and validating the interchangeability of experience managers and environments. These procedures could include tests to ensure that EMs and environment can communicate and exchange data effectively, as well as tests to evaluate the performance and reliability of the system. Testing and validation procedures are important because they help ensure that experience managers and environments are interchangeable and can operate effectively in a decoupled ecosystem.

We also need a good set of documentation and support resources that explain how to use the interchangeability protocols and standards and provide guidance on troubleshooting any issues that may arise. This would ensure that users of the system have the information and support they need to develop the interchangeability approach effectively.

A solution to the problem of interchangeability of EMs and environments will require addressing all of the core challenges discussed in this section. By establishing clear answers to all the points, we can ensure that experience managers and environments can communicate and exchange data consistently and reliably, adapt to different scenarios, and handle errors and unexpected conditions. In Chapter 5, I will explore these issues in greater depth and propose a framework for achieving interchangeability of EMs and environments.

2.3 Criteria for Success

To claim success in developing a framework for achieving interchangeability of experience managers and environments, it is important to demonstrate the feasibility of the approach by implementing a prototype system that uses the framework to achieve interchangeability. Developing an example to demonstrate the theoretical framework in practice provides a concrete illustration of how the framework can be applied in a real-world scenario. By creating an example, I can show how the different components of the framework work together to achieve the interchangeability of experience managers and environments and understand the challenges and

limitations of using this approach. The framework acts as an intermediary layer between the experience manager and environment, offering a common representation of communication and data exchange. This representation enables experience managers and environments to interact and exchange information in a consistent and predictable manner, similar to how they would in a joint perspective, but with the advantage of a shared and decoupled architecture. An additional benefit of implementing an example is that I can illustrate how developers can use the framework to create decoupled experience managers and environments. This demonstration is important because it helps developers to understand and utilize the framework and provides a reference for their work. Overall, developing an example to demonstrate the theoretical framework in practice is a crucial step in demonstrating the functionality and promoting the growth and potential of the ecosystem.

For the framework to achieve its goal of facilitating the interchangeability of experience managers and environments through a shared protocol for communication and data exchange, I need to prove two key elements. First, the framework must reliably and consistently share data, and second, the framework must support interchangeability.

2.3.1 Reliability and Consistency

There are two ways to develop an example with the objective of demonstrating the framework's reliability and consistency during the operations of data exchange: (i) by implementing a new experience manager and environment or (ii) by adapting an existing experience manager and environment to work using the theoretical framework.

The benefit of the first approach is that it allows the creation of an example specifically designed to showcase the capabilities of the theoretical framework. By implementing a new experience manager and environment from scratch, I can ensure that the example is tailored to highlight the key features and benefits of the framework and that the constraints of existing components do not limit it. The disadvantage of this approach is that there is a risk of increased development time. Implementing a new experience manager and environment from scratch can be time-consuming and resource-intensive. This can increase the overall development time and cost of the example, which may be a disadvantage because the goal is to demonstrate the theoretical framework in practice, not to create a new experience manager and environment.

The benefits of the second approach are reduced development time and showing how the theoretical framework can be applied to existing experience managers and environments. This approach would benefit from reduced development time because replicating an existing experience manager and environment does not require a redesign of the two elements from scratch. This approach also has the benefit of showing how the theoretical framework can be applied to existing systems, which shows the flexibility of the framework to adapt to an existing experience manager and environment to let them work interchangeably.

This second approach also allows the developer to choose between implementing a fully equivalent or partially equivalent experience manager or environment,

relative to those being adapted. A fully equivalent reimplementation of an existing experience manager or environment would be a direct copy of the original system, with no changes to its functionality or behavior. To be fully equivalent, the converted system would need to exhibit equivalent behavior compared to the original system, including performing the same tasks, making the same decisions, and obtaining the same information to make these decisions. A partially equivalent reimplementation of an existing experience manager or environment would be a modified version of the original system, where some parts of the functionalities are kept the same, and others are changed. A partially equivalent implementation might be desirable for a developer who wants to generalize the experience manager implementation to support other types of environment. For example, we can imagine that a developer might want to convert an experience manager that was originally designed to work by learning the player model from a conversation, into an experience manager that can learn the player model from the quest selection. In this case, the algorithm that governs the decisions of the EM might be kept the same, but the way the player model is learned would be changed. The framework should offer developers the freedom to choose between implementing a fully equivalent or partially equivalent experience manager or environment, to best meet their needs.

To prove that data sharing is reliable and consistent, I decided to reimplement an experience manager and environment that were originally designed using a joint perspective and transform them into separate components using a framework. I will create a fully equivalent version of the experience manager, that can perform the same tasks and make the same decisions based on the same information. By comparing the behavior of the original and converted experience manager when given the same input, I aim to show that the data sharing process is consistent and reliable since the converted experience manager behaves identically to the original experience manager. However, for the environment, I will only create a partially equivalent version of the original game. It will still provide the necessary information to the experience manager up to the first decision point, but will not necessarily maintain the same fidelity in audio or video aspects of the game. The reason behind this choice is that I want to demonstrate the reliability and consistency of the data sharing process, not the fidelity of the environment. Reimplementing the environment to be fully equivalent to the original game would require a significant amount of time and resources, without an added benefit to the demonstration. Assuming that the converted experience manager behaves identically to the original up to the first decision point, given the same input, I will be able to conclude that the framework ensures reliable and consistent data sharing between the experience manager and environment, even if they were originally designed using a joint perspective.

2.3.2 Support for Interchangeability

To demonstrate the interchangeability feature of the framework, I need to create an additional experience manager or environment that can use the framework and showcase its ability to work interchangeably with the experience manager and environment implemented to test the data sharing. There are two ways that can be

used to demonstrate the interchangeability feature of the framework in practice: (i) by creating a new environment and (ii) by creating a new experience manager.

The first option is to create a new environment that utilizes the framework. However, as I mentioned in Section 3.2, in the literature there is already a framework (*Mimesis*) that supports the development of a specific experience manager which can function in various environments [150]. In addition, creating new environments can be a complex and time consuming process based on the complexity of the environment.

The second possible approach is to build a new experience manager that is compatible with the framework. This approach has two key advantages. Firstly, it would significantly reduce the development time compared to creating a new environment from scratch. Secondly, it would showcase the framework's ability to support multiple experience managers. As far as I know, there are no frameworks in literature that support the development of experience managers that can work interchangeably with different environments. Therefore, it would be beneficial to concentrate on creating a new experience manager that can operate using the framework. This approach would be highly innovative since it would demonstrate the framework's potential to support the interchangeability of experience managers.

2.3.3 Practical Considerations

Designing and developing these examples to demonstrate the theoretical framework in practice is a complex process that requires careful planning and consideration. This planning is not only necessary to ensure that these examples are effective in showcasing the capabilities of the theoretical framework, but also to identify the steps required to implement them successfully. To successfully implement the theoretical framework, the examples must showcase the following characteristics: (i) they must show the capability to communicate and exchange data between the experience manager and environment using the communication protocol, (ii) they must pass the procedures developed for testing and validations, and (iii) the player must be able to play a videogame using an environment where some part of the experience is managed by an experience manager (e.g., choosing the best story-path based on the player preferences).

Chapter 3

Related Work

This chapter provides an overview of the related work on decoupling experience managers from their environments. It is structured as follows: Section 3.1 describes the theoretical frameworks that have been used to guide the design of experience managers in the past. Section 3.2 discusses implementation frameworks that are similar to the proposed approach in this dissertation. Section 3.3 examines the formal languages used for knowledge representation in experience management and related fields. Finally, Section 3.4 explores the existing environments that could be used to test the proposed platform. Note that in this chapter, I will not describe any experience manager because an in depth analysis of the state of the art in the field of experience management is located in Chapter 4.

3.1 Theoretical Frameworks

A theoretical framework, in the context of experience managers, is a set of principles and methodologies that guide the design and implementation of AI systems that manage user experiences in interactive systems. They provide a systematic approach to address the challenges of designing, implementing, and evaluating such systems. Theoretical frameworks are important because they provide a common language for researchers to discuss and compare their work. In the context of this dissertation, to achieve the separation and interchangeability between experience managers and environments, I need to undertake a comprehensive literature review to identify challenges and constraints that I need to address to pursue my goal. This literature review will need to be done in the context of a theoretical framework that allows me to compare and contrast the different approaches of experience management.

Search-based drama management (SBDM) is a framework for the design of interactive drama systems that uses a search-based approach to guide a player's journey in a game by forecasting potential stories and adjusting the game world accordingly. Search-based drama management was first proposed by Bates (1992) [12] and developed by Weyhrauch (1997) [147]. Lamstein and Mateas (2004) and Nelson and Mateas (2005) revived the technique and applied it to the interactive fiction

Anchorhead [47, 78]. In SBDM, stories are modelled as plot points that can happen and an evaluation function designed by the author rates the quality of the sequence of plot points. The drama manager uses certain actions (DM actions) to influence the game world and lead the player toward a story that is rated highly by the evaluation function. Examples of DM actions are: causing an NPC to move to a certain location, altering the environment by causing certain parts of the game world to be visible or invisible, or letting the player find an item by placing it near their current location. The best action to take is chosen by searching over possible choices of DM actions and determining which one will lead to the most interesting plot. All of this occurs in an abstract model that communicates back and forth with the actual game. Planning-based experience managers can also be considered part of the SBDM framework since they use planning (that is a form of search) approach to guide the player’s journey in a game.

Declarative optimization-based drama management (DODM) is a method for guiding players in video games by predicting potential storylines and adjusting the game world accordingly [82]. This approach, as in SBDM, sees stories as a collection of plot points, which are evaluated by a function set by the game’s author. DODM is a generalization of SBDM since SBDM employs a variant of a game-tree search while DODM is agnostic about the projection technique. The SBDM and DODM specifications do not explore the prospect of separation and interchangeability between the drama manager and the game environment. They are also specific to interactive drama system that use a search-based approach to guide the player. Therefore, I cannot use them as a theoretical framework for my work.

Another theoretical framework was made by Roberts and Isbell (2008) where they classified a wide variety of managers using a set of 10 criteria (called “desiderata”) for evaluating the behaviours and affordances of experience managers [103]. These ten desiderata include speed, coordination, replayability, authorial control, player autonomy, ease of authoring, adaptability, soundness, invisibility, and measurability. Each system was analyzed using all ten desiderata, and each desiderata was classified into one of three categories that define if the authors designed the system for that desiderata or not. After the analysis, they concluded that is not clear if these systems can help the typical author since, to be used, they require highly technical skills that the author might not have. They also shed light on two problems concerning the evaluation processes in drama management: dependency on the content, and integration between drama managers and games. We discussed these problems previously in Section 1.2. While these desiderata are important considerations to evaluate when designing an AI manager, they are used only in the context of drama managers and they are not designed to address the challenges of separation and interchangeability between experience managers and environments. To achieve my objectives in this dissertation, I need to find a theoretical framework that addresses these challenges.

Snodgrass et al. (2019) presented a theoretical design framework called *PEAS* that stands for the four high-level personalizable aspects of games: Player, Environment, Agents, and Systems [119]. The purpose of this framework is to assist researchers when deciding which aspects of a game to personalize by using a set

of three guiding questions: why, how and what. The why question is intended to elicit the reasons why a game system is customized as well as the reasons why certain game elements are chosen to be customized. The how question is made to discover the method through which the game’s creator will customize the specific game elements they have chosen, and how the system will adjust to the player. The what question is intended to provoke thought about how the personalisation will be implemented within the gaming system. This framework is designed to be used by game designers and researchers to help them decide which aspects of a game to personalize during the design process. For this reason, it is not a theoretical framework that can be used to analyze existing AI managers and understand their relationship with the game environment.

Thue (2015) presented a theoretical framework called “Generalized Experience Management” (GEM). GEM views experience management as a process of changing how an interactive system works at the same time that one or more users have an experience with that system. When an AI manager is used for narrative experience management, the manager dynamically changes how the narrative world works as one or more users experience that world¹. GEM represents an experience manager as a foundation and a collection of interrelated “building blocks”, each of which addresses a distinct sub-problem of experience management. Thue (2015) initially presented it in an extended mathematical format [127], then I summarized it in a more conceptual form in Section 4.3. This framework is focused on the experience manager design, and it can help me to gain the necessary understanding of how experience managers are developed and how they interact with the game environment. For this reason, this is the framework that I decided to use during my research to analyze existing AI managers and understand their relationship with the game environment.

3.2 Implementation Frameworks

One of the first attempts to separate EMs and game environments was *Mimesis* developed by Young et al. (2004) [150]. The authors’ objective was to provide conventional game engines with a way to connect (via socket-based APIs) to intelligent components that could create novel and effective action sequences. My approach differs from Young et al.’s because they allow only the *Mimesis* EM to connect to different potential environments. Although *Mimesis* could be configured with additional components to extend its functionalities, it does not support a full replacement of its EM. For this reason, it does not support the full-interchangeability between EMs and environments that I am aiming for. Moreover, Young et al. distinguished between the degrees to which game engines and intelligent agents (such as EMs) are linked in their design. They identified three categories: *mutually specific*, which focuses on developing new features for a certain game engine employing a specific set of intelligent reasoning capabilities; *AI specific*, for when a specific set of AI tools has been created to work in several gaming environments;

¹Note that changing how the world *works* (how it transitions from one world state to the next) is sufficient to also capture changing how the world *is* (the content of the world state) since every next state is the result of a world transition.

and *game-specific*, for when an intelligent agent has been developed to work into a specific game engine. My solution can be thought of as generalizing the mutually specific approach, since we enable connections between a (potentially) wide range of reasoning tools and a (potentially) wide range of game engines.

Another attempt at separating intelligent reasoning systems from the game engine was made by Aha and Molineaux (2004) with the “Testbed for Integrating and Evaluating Learning Techniques” (TIELT) [2]. In TIELT, the reasoning system and game engine communicate through a set of five knowledge bases that act as intermediaries. The first one is the *Game Interface Description* which defines how the communication between the reasoning and gaming systems works. This communication is based on two types of messages: action messages which TIELT can use to affect the game, and percept messages which provide information to the reasoning system. Then, there is the *Game Model Description* which provides an explicit, abstract description of a game. It is composed of an initial state that lists all potential player-interesting items, operators that explain how to play the game, and rules that specify how the game may be played with the effects that are likely to occur after a specific game state. The *Learning Interface Description* and *Task Descriptions* also use message templates to define communication with the reasoning systems. The last of the intermediaries is the *Evaluation Methodology Description* which allows a researcher to define exactly how to conduct an evaluation. TIELT was developed to enable intelligent systems to learn to play games in a particular genre: real-time strategy. In fact, Aha et al. (2005) used TIELT to evaluate the performance of a case-based approach that learns to select which tactic to use at each state in *Wargus*, a real-time strategy game that is a clone of the popular commercial game *Warcraft II* [3]. My dissertation does not focus on a specific type of game, but rather on the general problem of how to separate experience managers from the environments where they operate. In the case study presented in Chapter 6, I used EM-Glue with a narrative-based game, which is a different type of game from the real-time strategy games used in TIELT. For example, if we imagine to connect a game engine that supports dialogues in TIELT, as far my knowledge goes, we would not be able to learn the player model from that dialogue because TIELT’s Model Updater component (the component that reads what is happening in the game and updates the world state) does not support this feature. In fact, TIELT hosts all the components needed to abstract the game state and share this information with the intelligent system. Meanwhile, my approach requires that the game engine provides this abstraction and shares it with the EM making it more flexible and adaptable to different types of games. Nevertheless, certain overarching principles and design choices are common to both my approach and TIELT. For example, both use a third component to mediate between the reasoning system and the game engine, rely on a declarative model of actions to define possible moves, and use a declarative model of the game state to define the initial state and updates.

Cesar Jr et al. (2011) presented a framework for communication between a planning system (the “planner”) and a plan execution system (the “executor”) [17]. This framework consists of the architecture that encapsulates the communication between the planner and the executor to enable the two subsystems to operate

independently, as well as the interface rules and algorithms for how the planner and the executor subsystems communicate through this architecture. My approach differs from the work of Cesar Jr et al. because they focus on experience managers composed of planners, while my approach targets experience managers regardless of which components they use.

Szilas et al. (2011) also sought to create a common architecture to separate environments from intelligent systems. *OPARIS*, an architecture for Interactive Storytelling, provides a structure of modules that communicate via socket APIs, toward facilitating the integration of various different Interactive Storytelling components using a common architecture [122]. The functionalities are divided into modules that are independent software components that communicate with the platform via a set of messages. My approach differs from theirs because *OPARIS* targets Interactive Storytelling systems and it focuses on narratives, whereas my framework focuses on EMs and aims to remain agnostic about the content of each environment.

3.3 Knowledge Representation

GDL is a logic-based language that describes the rules of a game (legal moves, state transitions, goals of the players, etc.) in a declarative way. GDL allows the players to interpret the rules of a game, both simulating the game and analyzing it to find structures that might help develop a strategy to play the game successfully. The first version of GDL is limited to discrete, deterministic, and perfect-information games with an arbitrary number of players. However, researchers have created new versions to allow the representation of non-deterministic, imperfect information games [113], as well as introspection of the players' knowledge [126]. GDL is simple to interpret yet powerful enough to describe arbitrary games [125], which is why the GGP field is using it extensively. In addition, having a common description language for game rules made it possible to hold annual competitions among General Game Playing agents developed by researchers around the world [33]. These competitions have led to considerable interest in the field, both from researchers and for education [124].

Similar to GGP, General Video Game Playing (GVGP) [86] intends to develop techniques for playing (arbitrary) video games properly. In distinction to GGP, GVGP focuses more on single-player real-time games, where the game's rules are not necessarily known to the player. Instead, players access a forward game model that allows the game's simulation. The availability of a program that simulates the games made GVGP very popular and led to a substantial volume of research despite the short time since the initiation of the field. GVGP does have a description language for general video games [112], but its focus on relatively simple video games makes it not general enough for arbitrary experience management tasks.

Another domain that benefited from a common language for describing problems is automated planning. The International Planning Competition has been held nine times since 1998 (and the tenth will be held in 2023) and uses the Planning Domain Definition Language (PDDL) to describe and communicate planning

problems to the participating agents [67]. Having a common language and regular competition with a wide selection of different problems has led to steady and measurable progress in automated planning over the years. Researchers also used PDDL besides its planning purposes, for its description capabilities to describe the domain of a story problem [90, 135, 142, 25, 91]. An example of an experience manager that uses PDDL to represent domains and stories is the system developed by Porteous et al. (2010), in which they built a PDDL-based narrative replanner to produce multiple variants of narratives and control story pacing [90]. PDDL can be considered as a candidate to be used as data standard (as described in Section 2.2) for the framework I propose in this dissertation. It has all the characteristics that a data standard in this context should have and it is a widely used language in the AI community and it is already used for similar scopes in the experience management community.

The experience management community has also created languages that serve as a means to represent knowledge and convey various aspects of an experience. One example is the “A Behavior Language” (ABL) designed by Mateas and Stern (2004) and used in the interactive drama *Façade* [61]. ABL was developed with the goal of enabling authors to create believable artificial agents. Each agent has a library of pre-written behaviours, and each behaviour is made up of a series of operations that may be carried out sequentially or concurrently in order to achieve a certain objective. In ABL, a goal is a representation of an action (e.g., approaching the user, or delivering a line of conversation), and each goal is given one or more behaviours to carry out its assignment. Preconditions are used to determine behaviour applicability by matching against working memory elements that make up the agent’s subjective knowledge about the world. However, in the context of a data standard to be used in the framework that I propose in this dissertation, ABL is not a good candidate because it is not a general-purpose language since it focuses on believable agents and it is not widely used in the experience management community.

Another example of a language that was developed in the experience management community is Ceptre [58]. Ceptre is a rule-based specification language developed by Martens (2015), which employs logic to express the rules of a game. The system states (configurations) of a Ceptre program are represented as multisets of logical predicates, and it offers rules that may be applied to those multisets to replace certain facts with others. An unordered set of rules, type and predicate definitions, and an initial state description make up the structure of a Ceptre program. Ceptre is another candidate language that can be used as a data standard in the framework that I propose in this dissertation. It has the characteristics necessary for a data standard in this context, but it is not widely used in the experience management community.

3.4 Environments

One environment that was designed to be used in a way that remains decoupled from experience managers is *The Best Laid Plans* [142]. It was developed by Ware and Young (2016) to facilitate the development of experience managers with focus

on planning. They designed it as a research prototype to show the potential of quick narrative planning for interactive virtual environments as well as a tool for evaluating plan-based computational models of narrative. To isolate the virtual environment from the story planner, the game uses a conventional client/server architecture and the PDDL language as complete description of the story world. This environment achieves the goal of separation with regards to the experience manager, but it does not provide the freedom that an environment should have to be used in any way the user wants. In fact, it uses a specific game type where the player creates plans for the characters to follow, then the planner uses this plan to generate a story coherent with the player’s plan. This environment is not suitable for other types of experience managers except for those that use planning to generate stories, thus not suitable to be used with my approach.

Another example of an environment is *Camelot* [111, 117], a visualization engine developed by the Narrative Intelligence Lab at the University of Kentucky. *Camelot* provides a sandbox to visualize and test different narrative systems. It accepts as input a series of text commands, and it visually presents a 3D environment with characters, locations, objects, and items that respond to the commands. It allows researchers of EMs to build a simple testbed without the need to start from scratch in creating a new environment. An example is the testbed developed by Ware et al. (2022), where they used *Camelot* to create and facilitate an environment composed of four locations and three NPCs. Since *Camelot* accepts a set of instructions that are specifically designed for the software, developing an EM for *Camelot* requires a high dependency on the *Camelot* commands. With this work, I want to drop this dependency, allowing an environment to be used by many different EMs (which *Camelot* supports already), while still allowing those EMs to be tested using many different environments (which *Camelot* makes difficult). *Camelot* supports the creation of environments that have the theme and gameplay of a medieval computer role-playing game, but it does not support a complete change of game genre (e.g., to a spaceship game).

Another example that has the potential to be used as evaluation environment is the commercial computer game *RimWorld* [64], which allows the players to choose between three experience managers to play with. However, to the best of my knowledge, no comparisons of the performance of *RimWorld*’s managers have been published, and the game itself is not available as a platform for evaluating new managers.

A testbed that could be used as visualization tool for experience managers is *FarmQuest* [156] developed by Yu et al. (2022). *FarmQuest* is a video game with game loop that requires the player to interact with an experience manager. The experience manager selects which are the quests that are most suitable for the player, and the player is provided with the option to select which one to choose. The game is designed to be used as a testbed for experience managers that use quests to influence the game experience, and it was recently used by Yu et al. (2022) to evaluate three different experience managers: a random manager, a manager that uses a reinforcement learning algorithm, and an implementation of *PaSSAGE* [131]. *FarmQuest* is a good candidate to be used as a visualization tool for experience managers, but its focus on quest management makes it unsuitable to be used with

general-purpose experience managers.

Chapter 4

Analysis of EM Techniques

To define a way to achieve interchangeability of experience managers and environments, it is important to have a comprehensive understanding of the various techniques used in developing EMs. This knowledge will enable me to identify the necessary requirements by drawing on past experiences in EM development. By analyzing the strengths and limitations of different techniques, I can make well-informed decisions when proposing an approach that is best suited to this particular application. In literature, many AI managers have been created using a variety of AI techniques, including planning, curve fitting, filtering, and more. In spite of all this work, however, the field has lacked a critical component for ensuring methodical progress: a way to meaningfully compare the inner workings of different AI managers. In this chapter, I show that such comparisons can be made in the context of an existing conceptual framework, and I support this claim by analyzing and comparing the inner workings of diverse managers.

4.1 Paper Selection Methodology

In this chapter, I want to answer these two questions: *What are the main techniques that researchers have used to develop experience managers?*, and *With which of Thue and Bulitko's perspectives were the managers that I analyzed developed?*. It is important to answer these questions for two reasons. Firstly, understanding the techniques used to develop experience managers can provide insights into the underlying methods used to create these systems. In general, this knowledge can be useful for researchers who are interested in developing similar systems or who want to evaluate the effectiveness of existing experience managers. In the context of this thesis, this knowledge can help us to understand the techniques used to develop experience managers and to identify the most valuable aspects to consider when designing a tool that supports partial-interchangeability. Secondly, identifying the perspective used during the development of experience managers can help to understand the state of the separation of experience managers and environments in the literature. This can help me to gain insights into the current trend in the literature and to understand if the partial-interchangeability between experience

managers and environments is a goal that can be pursued.

The first step to answer these questions is to gather a set of papers that describe how experience managers work. There are many ways to structure a literature review. For my work, I drew inspiration from the guide by Silva and Neiva [118] and followed the steps described in Figure 4.1.

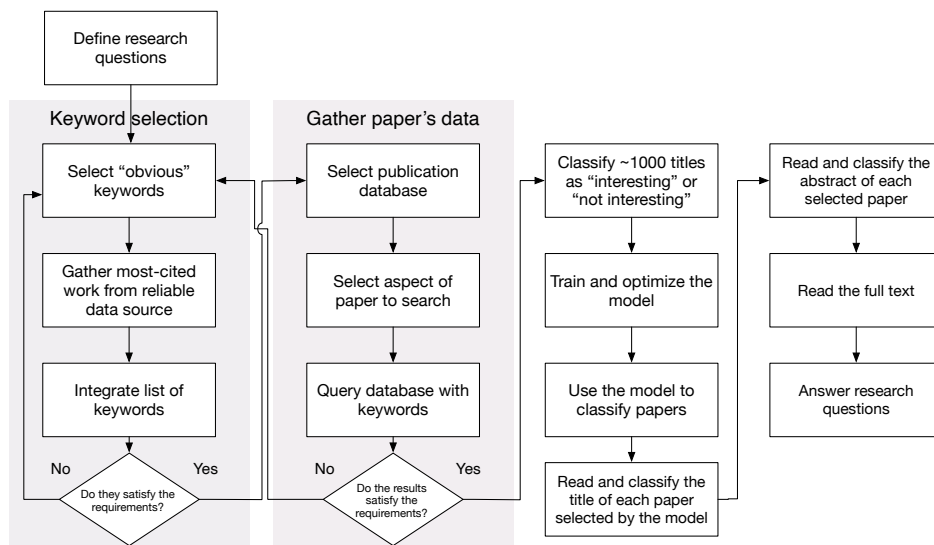


Figure 4.1: Flowchart of the literature review process.

The first step was to start the keyword selection process by querying Google Scholar with the following keywords: experience, drama, management, and evaluation. I chose Google Scholar to gather influential papers because it offers a broad collection of scientific material. I stopped after extracting ten unique keywords from 30 different papers.

To expand the initial list, I used two approaches. First, I split each title into words and placed them into a list. I then counted the occurrence of each word in the list. Doing so allowed me to find the recurring words more frequently and assess whether they were related to the scope. If they were, I included them in the set of keywords. Secondly, I analyzed the paper's text and included terminology that I deemed related to the research questions. The final keywords were: interactive, management, drama, narrative, experience, game, player, evaluation, story, and model.

I used these keywords to query Scopus [63], which I choose as primary data source to gather all the papers needed. Scopus is a comprehensive source for scientific publications, covering material from a large number of data sources. This means that when searching on Scopus, we get results from several conferences and journals at once. The content is carefully curated and selected by experts, who ensure that only the most reliable scientific articles and contents are available. Finally, Scopus has a clearly-designed application programming interface (API) that allows complex queries, exports large batches of results, and can be used

remotely through programming languages (e.g., Python). I explored several other alternatives, but all of them were lacking at least one of the features mentioned above. For example, Google Scholar, while having a broad collection of scientific material, is not as curated, and does not offer APIs.

To perform the search, I created a Python script that first calculated all pairs of keywords and then searched Scopus querying a different pair of keywords every time (e.g., experience AND management, drama AND management, etc.). I connected to the Scopus API via `pybliometrics` [106], a Python library designed to query Scopus in a structured way. Since the search process took a long time (up to 30 minutes for some keyword pairs), I stored the raw results of the queries in a file, and then moved the data to a MySQL database. During this process, I removed duplicates and added some columns to use later for tracking when I read a title or abstract. This search resulted in 26,046 papers to classify and was made in September 2019.

One can view the initial classification step of literature review as a deterministic function, which separates the state of the art of a particular topic into either *interesting* and *not-interesting* material, according to the researcher. Since classifying this high number of papers is a long and repetitive task, I came up with a machine learning model that uses binary decision trees to classify papers as *interesting* or *not-interesting*. This machine learning model is described in more detail in Section 4.1.1. Then, I created a tool to make the classification process easy. The tool presents the user with a title; the user reads it and decides if it is interesting for the literature review. However, instead of presenting the title without any reasoning, the tool first makes a classification using the model trained on previously labelled titles. If the model classifies the title as interesting, the tool presents it to the user. Thanks to the model, I did not read 18,312 classified as *not-interesting*.

After I finished re-classifying the paper's titles, I had 416 paper titles classified as *interesting*. Then, I analyzed the abstracts of these resources. If the abstract was interesting, I briefly scanned the full text for further selection, looking to see if the paper described what I needed to answer the research questions. After this process, I ended up with 67 papers that were further analyzed to extract the candidates' systems for this literature review. During this process, I excluded papers from the same authors, selecting only the most recent or comprehensive work. For example, if an author published a conference paper and a journal paper covering the same system, I selected the journal paper. Then, as the last step of selection, I fully read the papers and selected just the ones that were most relevant to the research questions by describing the implementation of the experience managers, ending up with 24 papers in total.

4.1.1 Model for Paper Classification

Machine learning offers several models that can be employed to solve classification problems, including support vector machines [21], decision trees [110], random forests [51], and neural networks [39]. Natural Language Processing research currently seeks to develop complex deep neural networks to improve language recognition and classification [151]. However, using deep learning models comes with a

problem: it is difficult for researchers to understand the path that the model took to yield a certain result [7, 53]. For this reason, I opted for a relatively simple approach using Decision Tree Classifiers. Machine learning models that use decision tree classifiers might not have the best in class performance, but they are simple to use, extremely fast, and straightforward to interpret, visualize, and understand.

Decision Tree Classifiers, like any other supervised learning approach, need samples for training their classification function. This means that I needed to perform an initial survey of the literature and add an *interesting* or *not interesting* label to each paper in a chosen sample. Since this initial collection of research material should be a representative sample of the entire literature, it needs to be unbiased. For example, the results obtained from querying the data source may be ordered according to arbitrary criteria (e.g., citation count). In my case, since I queried Scopus successively using a pair of keywords at each time, I needed to avoid the classifier giving more value to some keywords simply because I added them before others in our database. I mitigated this problem by randomly shuffling the entries retrieved from our MySQL database before beginning the labelling process.

Training Set Preparation. A classifier generally becomes more accurate when more samples are used to train it. So, to define the size of the training set, I employed an iterative approach: (i) read a portion of the titles and classify them as *interesting* or *not-interesting*, (ii) train the model with the result of the previous step and use it to classify the remaining material, (iii) read the material that the model has classified as *interesting*, (iv) correct any misclassified title, and (v) integrate the material into the training set. Each time this process is repeated, the training set grows and the classification can improve. One final consideration that I used when building the training set is the balance of that set. To train well, machine learning classification approaches require examples from every class that they are meant to recognize. If the unrelated papers greatly outnumber the related papers, the model will be biased towards that specific class. Some techniques can be employed during training to mitigate this issue (e.g., oversampling or giving more importance to a class during training).

I built the training data set by retrieving and labelling 1000 random papers from the set of 26,046. To make my life easier during the classification process, with the help of a fellow Ph.D. student Michelangelo Diamanti, we designed a simple Python tool [24] that displays the titles in a window with two buttons to label the title as *interesting* or *not interesting*. This is the same tool that was mentioned in the previous section, but without the model.

Once I finished labelling the papers for the training set, I analyzed the results to understand if the data was balanced. The result of the analysis was that the training set had more titles labelled as *not interesting*. To improve the balance, I added titles that we knew were interesting from my own collection. I was aiming to have around 150-200 titles labelled as *interesting* so that the decision tree would receive sufficient examples for that category.

Model Development. This model classifies papers based on their titles. A title is an extremely condensed version of the research being discussed, and it is

composed of a limited set of keywords which are highly significant for defining the paper’s content. However, it is not possible to use raw titles for training a Decision Tree Classifier, as they must first be pre-processed. The goals of pre-processing are: (i) to remove useless words from the titles (e.g., “the” or “and”), (ii) to split titles into keywords, and (iii) to encode the relevant keywords in a suitable format usable by the model. These three steps are respectively called noise reduction, tokenization, and feature extraction.

To reduce the noise in the data, I used two approaches. The first one was a pre-processing step via regular expressions. In this phase, I applied four conditions to the text. First, I removed all the non-alphabetic characters. Next, I removed all the single character words in the phrase. After that, I substituted multiple spaces with a single space. Finally, I converted every title into lower case. In the second approach, I used a technique called lemmatization to reduce the words to their primary form. To do so, I used the WordNetLemmatizer from the Python library *natural language toolkit* (nltk) [54].

For the feature extraction process, I used the CountVectorizer from the Python library *scikit-learn* [83]. The CountVectorizer also includes a process of tokenization, so I did not need to execute it separately. This type of vectorization provides a simple way to build a vocabulary of known words and encode new documents using that vocabulary. The result is an encoded vector with the length of the vocabulary’s size and an integer count of each word’s appearances in the document. When using a CountVectorizer, one can control many different parameters to adjust its behaviour. In our case, I decided to change four types of parameters:

- *ngram_range* with value “(1,2)” – to allow the inclusion in the dictionary of both unigrams and bigrams.
- *stop_words* with value “stopwords.words('english')” – to reduce noise by removing english stopwords.
- *token_pattern* with value “ r'\b[\w]4,(?<![\d])\b” – to reduce noise by removing words with fewer than four characters or digits.
- *min_df* with value “2” – to ignore terms that occurred fewer than two times.

Given this vectorization process, I began to develop our model for classification. I decided to use cross-validation [88], and thus split the dataset in 80% training and 20% test. Then, I trained a binary Decision Tree Classifier using the implementation from Python’s “scikit-learn”. When building and exporting a model that also uses a feature extraction process, it is good practice to chain together the feature extraction with the classification algorithm so that the output of the first goes directly to the model; this gives a consistent flow of data from raw inputs to classification results. To do so, I used the “Pipeline” tool in “scikit-learn”.

Model Optimization. The last step in the process of training a machine learning model involves finding the set of parameters that yields the best results according to one’s chosen evaluation metrics. In this case, one arguably important metric is the false negative rate of *interesting* titles. This number identifies the amount of

titles classified as *non interesting* by the decision tree, but originally deemed *interesting* by our initial assessment. It is very important that this value is minimized, to avoid discarding pieces of literature that might actually be important to review.

Keeping this goal in mind, it is possible to tweak the parameters of the model and analyze its performance through the following evaluation metrics:

- **Accuracy:** the ratio between the number of correct predictions and the total number of predictions made by the model with a particular label of classification.
- **Confusion matrix (CM):** describes the performance of the model by showing the combination between predicted and actual values. The CM allows the analysis of true positives/negatives and false positives/negatives. True positives/negatives occur when the model predicts a particular value and the prediction is correct. False positives/negatives occur when the model predicts a particular value and the prediction is incorrect.
- **Receiver Operating Characteristic (ROC):** a graph of the True Positive Rate (TPR)¹ versus the False Positive Rate (FPR)², showing the performance of a classification model at all classification thresholds.
- **Area Under the ROC Curve (AUC):** measures the two-dimensional area underneath the entire ROC curve. It measures how well predictions are ranked. An AUC value of 1 means that all the predictions were correct, while 0 means they were all wrong.
- **Precision:** the ratio between true positives and all samples classified as positive by the model. Describes the percentage of positive classifications that were actually correct.
- **Recall:** the number of true positives divided by the sum of the true positives and the false negatives.
- **F1-score:** the Harmonic mean between precision and recall.

Through trial and error, it is possible to tweak the model's parameters to optimize the aforementioned metrics. For example, it is important find a set of parameters which maximise the model's *accuracy* and increase the number of correct predictions. Nevertheless, those same parameter values might increase the amount of false negatives in the *interesting* class. In the worst case scenario, the model could correctly classify all the *non interesting* titles while misclassifying every *interesting* title, and still yield a high accuracy value. For this reason, while tweaking the parameters for optimizing a certain metric, one should also be mindful of the effect that they have on the other metrics. It is important to check the *confusion matrix* for assessing the raw number of false positive/negatives. Moreover, one should try

¹True positive rate corresponds to the proportion of positive data points that are correctly considered as positive.

²False positive rate corresponds to the proportion of negative data points that are mistakenly considered as positive.

to maximise the *ROC*, *AUC*, and *Recall* values, which indicate the ratio between correctly classified and misclassified data points. Finally, one metric which combines accuracy with false negatives is the *F1-score*. By maximising this metric, one should have the highest amount of correctly classified titles, while minimizing the chance of misclassifying interesting titles. Since this process is time consuming, it possible to automate it by brute-force testing various combinations of parameters. At the same time, one should first assess, for each parameter, a range that of values that could yield promising results, and thereby reduce the number of combinations that need to be explored.

To optimize the model, I needed to tune the parameters available in the Decision Tree Classifier. To do so, I changed six parameters from their default function: criterion, splitter, class weight, min sample split, min sample leaf, and max depth. Every parameter can influence the other; thus, I needed to find the best possible combination of values. I began by changing one parameter and re-training the model to understand how to change the values of the parameters listed above. Through this initial manual assessment, I found ranges for the parameter values that yielded the best results: (i) *Max Depth*: between 7 and 12, (ii) *Min Samples Split*: between 0-10% of the data set, and (iii) *Min Samples leaf*: between 0-10% of the data set. Since the Decision Tree is quick to train and test, I decided to try many possible combinations of these parameters. *Max Depth* was increased by 1 each time, while *Min Samples Split* and *Min Samples leaf* were increased by 0.5%. By employing these restrictions, the explored parameters ranged around the value I expected to fit. Eventually, I was able to find out which parameter combination yielded the best results without worrying about a combinatorial explosion. During this process, I fixed the *split criterion* to “gini”, *class weight* to “balanced”, and *splitter* to “best”, for two reasons: to limit the number of combinations, and because the alternatives were performing worse. To do this brute force parameter analysis, I created a Python script [24] that, for each combination of the parameter/values, trained a binary decision tree and saved the results of the evaluation metrics in a CSV file to create a report. By analyzing the report that was produced, I decided which were the best values for the model’s parameters: *criterion* with value “gini”, *splitter* with value “best”, *min_samples_split* with value 7.5%, *max_depth* with value 7, *class_weight* with value “balanced”, and *min_samples_leaf* with value 0.5%. Finally, I tested the model using the metrics listed above (with 10-fold cross validation) with the following results: *accuracy* with value 0.74, *precision* with value 0.68, *recall* with value 0.88, and *roc_auc* with value 0.83.

4.2 Analysis of Selected Papers

This section presents the 24 papers selected for this analysis, organized by publication year. For each paper, I summarized the main focus of the experience manager, and I determined whether the system uses a joint or disjoint perspective [129]. This analysis is based on the description and implementation of the system and is preceded by the “**J/D:**” text that stands for “Joint/Disjoint perspective”. A system is using a disjoint perspective if one of the following criteria are met: (i) there is a clear mention of separation in the implementation of the experience manager

and environment in the paper’s text, (ii) the experience manager uses as environment a system that was made to be used by multiple experience managers (e.g., *Camelot* [117]), (iii) the environment was used on other unrelated projects, (iv) if access to the source code is available, the codebase should show a clear separation between the experience manager and environment code. If none of these criteria are met, then system is classified using a joint perspective.

We start the analysis with the work from Magerko (2005) where they introduced the *Interactive Drama Architecture* (IDA). IDA is a system for creating interactive narratives and storytelling experiences. IDA aims to provide an immersive and interactive storytelling experience while maintaining the coherence and structure of the plot and uses a combination of reactive and preemptive guidance techniques to achieve this balance [55].

J/D: Without access to the implementation of IDA, it is difficult to determine if the system is using a joint or disjoint perspective in its development. However, the text describing the system shows that the system uses a joint perspective in the implementation since there is no mention of the separation of the experience manager and the environment.

Nelson and Mateas (2005) and Nelson et al. (2006) applied search-based drama management (SBDM) [147] and declarative optimization-based drama management (DODM) [82] to the interactive fiction piece *Anchorhead*, with the objective of further investigating the algorithmic and authorship issues involved in the use of these techniques. I decided to include both papers in this analysis even if they are connected because the first focuses on SBDM and the second on DODM (even if it has references to the first one). The drama manager’s objective is to create a dynamic and engaging story that keeps the reader or player engaged and invested in the outcome. This may involve balancing the reader’s needs and preferences with the constraints and plot points of the story to create a cohesive and believable narrative.

J/D: In this case, no implementation details or mentions of separation between environment and manager are provided in the two papers’ texts. Using a purpose-built text-based environment suggests that the systems use a joint perspective in their development with options to choose which DM to use.

Mehta et al. (2007) performed a qualitative study of the interactive drama *Façade* [60], where the DM makes decisions about how the story should unfold and how the characters should respond to the player’s actions. The study aims to understand the relationship between the decisions made by the drama manager and the player’s perception of the game and to use this understanding to inform the design of future interactive dramas [68].

J/D: For their study, Mehta et al. used *Façade* by Mateas and Stern (2003). So, the evaluation of Thue and Bulitko’s perspective is based on the implementation details of *Façade*. Judging how *Façade* is implemented without access to the source code is complex. In Mateas and Stern’s (2003) paper, we can find a schematic of the interactive drama architecture (Figure 2 of the paper [60]) and mentions in the text of communication between the system’s parts that suggests that they used a disjoint perspective. However, the developer of the *Façade* project used a joint perspective because there is no explicit mention of separation of manager and

environment and the environment was not used in other projects.

Thue et al. (2007) presented a system called PaSSAGE, an interactive storytelling system that bases its storytelling decisions on an automatically learned player model. This player model understands the preferred play style of the current player and uses this knowledge to adapt the content of an interactive story dynamically.

J/D: In this case, I had the opportunity to access and analyze the implementation of the system. By accessing the source code, I could see that there was no clear separation of the experience manager and environment modules. In fact, in most of the scripts, there was a mix between manager actions and code that would directly change the environment in response to the action. From this analysis, it was clear that Thue et al. used a joint perspective in the development of the experience manager.

Peirce et al. (2008) designed a system called ALIGN (Adaptive Learning In Games through Non- invasion) that uses an experience manager to adapt the difficulty level of an educational game based on the learner’s performance. The EM aims to provide a personalized learning experience tailored to the individual learner’s needs and abilities without disrupting the gaming experience.

J/D: In the paper’s text, we can find references to the conceptual separation of EM and environment. However, since they say that it is conceptual, I infer that the system was implemented using a joint perspective.

Cheong and Young (2008) introduced the Suspenser Framework that can determine the narrative contents (i.e., the *sjuzhet*) of a story to evoke a high level of suspense in the reader [18]. To do this, it receives in input a *fabula*, a point in the story where the reader’s suspense is measured, and the desired story length to determine the *sjuzhet* based on the likelihood of different outcomes occurring in the story world.

J/D: The objective of this system is to generate stories, and the environment where the user can read the generated stories is not mentioned in the paper. Thus, the system uses a joint perspective in the development.

Corradini et al. (2009) developed an interactive story-based game based on the interactive fiction game *Anchorhead* with the inclusion of a drama manager [20]. The DM aims to improve the player’s subjective experience and enjoyment of the game by helping them progress through the game using hints and understanding the story more thoroughly.

J/D: As with *Façade* [60], this paper includes an architecture diagram (Figure 2 of the paper [20]) that seems to suggest that the system is using a disjoint perspective. However, since there is no direct mention of the separation of manager and environment in the text and since the environment was not used in other projects, it is more likely that it was developed using a joint perspective.

Sullivan et al. (2009) presented EMPATH, a *Zelda*-style adventure game developed with a drama manager that uses a modified version of DODM to monitor the gameplay and intervene to shape the global experience [120]. The drama manager uses an evaluation function to cause specific plot points to happen, provide hints that make it more likely that a plot point will happen, deny a plot point so that it cannot happen, or un-deny a previously denied plot point.

J/D: In this paper, no implementation details or mentions of separation between the environment and the manager are provided in the text. The use of a purpose-built environment suggests that the system uses a joint perspective in its development, with the option to play the game with or without the manager.

Sharma et al. (2010) developed a subset of the game Anchorhead that is a text-based interactive game with a complicated story, several plots and subplots [116]. To help manage the experience, they included a drama manager called C-DraGer (Case-based Drama manaGer) that observes the player's interaction during the complete course of the game. C-DraGer is responsible for producing interesting story arcs using both search and case-based reasoning.

J/D: The paper includes a diagram (Figure 3 of the paper [116]) that appears to show that the system was developed using a disjoint perspective. However, there is no mention in the text of the separation between manager and environment, and the environment was not used in other projects. Therefore, it seems more likely that the system was developed using a joint perspective.

Arinbjarnar and Kudenko (2010) presented a system called Directed Emergent Drama (DED) whose objective is to create a dynamic and engaging narrative experience within a game by directing the actions of characters in a way that supports the overall dramatic structure and character development [6]. The DED system uses a director, schemas, actors, and a player to integrate drama into gameplay in a way that supports gameplay rather than conflicting with it.

J/D: The focus of this paper is on the DED architecture, and the authors do not provide implementation details or a description of the environment. However, in their dissertation, Arinbjarnar show that they implemented the DED system in four diverse games over two environments [4]. Therefore, it seems that the system was developed using a disjoint perspective.

Tomaszewski (2011) presented a game called *Demeter: Blood in the Sky* that uses the Marlinspike drama manager to manage the experience of the player [137]. Marlinspike handles the player's experience by controlling NPCs by selecting scenes.

J/D: From the description of the implementation of Demeter, it is clear that the system is using a joint perspective in its development.

Lee et al. (2014) implemented a machine learning model to guide the player's investigative activities and control the narrative's pace and progression during the game *Crystal Island*. *Crystal Island* [108] is an educational adventure game that features an interactive science mystery set on a recently discovered tropical island.

J/D: The focus of this paper is on the machine learning model, and the authors do not provide implementation details of the environment other than saying that the model was integrated with the game. However, since *Crystal Island* does not provide a standardized way to integrate EMs, the system is likely using a joint perspective in its development.

Endrass et al. (2014) used the *Virtual Beer Garden* application [22] to model different modes of dialogue interaction (round-based and continuous interaction) on the user experience. Their interactive storytelling system aims to create a dramatic, nonlinear experience for the user by allowing them to influence the progress and outcome of the story through their interactions and contributions.

J/D: The paper does not provide implementation details of the interactive nar-

rative system, but it focuses on high-level design decisions. Their environment is based on the Advanced Agent Animation [22] framework. This framework provides a set of tools to separate the environment and the experience manager. Therefore, this suggests that the system uses a disjoint perspective in its development.

Harrison and Roberts (2014) created a game called *Sidequest: The Game* to serve as a test environment to test how dynamic game adaptation, paired with game analytics, can be used to increase the percentage of players that complete a single game session (session-level retention) in a casual game [40]. They used game analytics to abstract the game state into a smaller space and then used this space to alter the game environment to target game states that can improve session-level retention and avoid states likely to lead to player quitting.

J/D: The paper provides little information about how the environment and the manager are implemented. However, the fact that the system was developed using a purpose-built test environment suggests they used a joint perspective during the development.

Poo Hernandez et al. (2015) presented the implementation of the EM called *PACE* (Player Appraisal Controlling Emotions) [42] by adding it to a novel narrative-based video game called *iGiselle* [89]. *PACE*'s objective is to keep a player on a target emotional trajectory during the game playthrough by selecting the next bit of narrative from the candidates generated by an AI planner.

J/D: The paper shows that a purpose-built environment was used to test the system. This suggests that the system was developed using a joint perspective.

Yu and Riedl (2015) designed a personalized drama manager to create an enjoyable and coherent narrative experience for the player [154]. The DM does this by monitoring the fictional world and determining what will happen next in the player's story experience, often through coordinating and instructing virtual characters in response to the player's actions.

J/D: The environment used to test this drama manager is an open-source tool called *Undum* [70]. However, the paper does not provide implementation details of the environment, and the tool does not provide a standardized way to separate the environment from the experience manager. These two facts suggest that the system was developed using a joint perspective.

Ramirez and Bulitko (2015) presented a system called Player-specific Automated Storytelling (*PAST*) [94]. *PAST* aimed to perform narrative mediation similarly to Riedl et al. (2003)'s work in *Mimesis*, but with the addition of a learned player model inspired by Thue et al.'s (2007) *PaSSAGE*. When the player performed an exceptional action, *PAST* used a modified version of the Automated Story Director's planner [100] to generate a new story plan tailored to the player model.

J/D: The paper does not describe the implementation of the manager or environment. However, the fact that the system was developed using a purpose-built text-based environment suggests they used a joint perspective.

Ware and Young (2016) presented the game *The Best Laid Plans* an interactive narrative point-and-click adventure game that uses fast narrative planning techniques to generate stories in real time [142]. The planner uses models of agent intentionality and conflict to create believable character behaviour and obstacles

for the protagonist to overcome.

J/D: *The Best Laid Plans* environment presented in this paper as a simple client/server architecture that can be used to decouple the environment from the experience manager. This suggests that the system was developed using a disjoint perspective.

Wang et al. (2017) implemented a deep reinforcement learning algorithm to personalize interactive narratives in an open-world game environment [140]. The authors evaluate the effectiveness of this approach by using it to personalize an educational interactive narrative called *Crystal Island* [108] and compare it to linear RL techniques. They also introduce a bipartite player simulation model that uses classifiers to generate synthetic data on player actions and outcomes, which can be used to train the RL algorithm.

J/D: As with Lee et al.’s (2014) paper, the authors used *Crystal Island* as the environment to showcase their model. This suggests that the system was developed using a disjoint perspective.

De Lima et al. (2018) proposed a new approach to creating interactive storylines in games based on player behaviour and personality modeling [23]. This approach involves using the Big Five factors [36] to model the player’s personality and an artificial neural network to predict player behaviours based on statistical features extracted from gameplay. A partial-order planning algorithm then uses the personality and behaviour models to create hierarchical quests during gameplay.

J/D: While the diagram in Figure 2 of the paper [23] suggests that the system was developed using a disjoint perspective, with a clear separation between the manager and the environment, the text of the paper does not mention this separation. The environment was not used in other projects, suggesting that the system was likely developed using a joint perspective.

Braunschweiler et al. (2018) presented an experience manager system whose main objective is to ensure that the story progression balances user agency with the author’s intended narrative, using the user model and the online planning system to guide the selection of story events and to mediate user actions [16]. The experience manager updates the user model based on the user’s engagement with the characters and objects in the story and uses the current state of the user model and the story domain representation to determine an optimal path through the space of available stories. It then generates story sequences in real-time to create a path to that optimal storyline, considering the user’s interactions and the desired narrative quality.

J/D: To test their system, they implemented a purpose-built environment that uses augmented reality and virtual reality technologies. However, since they are using a custom environment, we can assume they are using a joint perspective during development.

Ware et al. (2019) approached the problem of letting the experience manager choose which action an NPC should perform next by pruning the story graph. This approach consists of removing NPC actions from the graph until each NPC has at most one action to perform in any given state, resulting in a clear policy for the experience manager to follow [143]. This pruning technique is intended to maintain story quality while allowing for flexibility in player actions.

J/D: To test their system, the authors used the Camelot environment [117] that I discussed in Section 3.4. This choice suggests that the system was developed using a disjoint perspective.

Farrell et al. (2019) presented *Indexter*, a computational model of narrative event salience that can be used to predict and influence player behaviour in interactive narrative systems by manipulating the salience of past events [28]. *Indexter* is based on the idea that events in a narrative are stored and retrieved in memory along certain indices, such as protagonist, time, space, causality, and intentionality. They propose mapping these indices onto narrative planning events, which allows a system to calculate the salience of any past event as a function of the indices it shares with the current event.

J/D: To test *Indexter*, they used *Twine* [45], an open-source tool for telling interactive, nonlinear stories. However, they manually translated the stories that *Indexter* designed to work with *Twine*. The manual translation from the result of the computation to what *Twine* accepts indicates that the system was developed using a disjoint perspective since there is no connection between the two.

4.2.1 Summary

In this section, I analyzed the experience manager systems presented in the literature and classified them according to whether each system used a joint or disjoint perspective [129] during the development. In Table 4.1, I summarized the results of this analysis. This table serves two purposes: first, it provides a summary view of the experience managers analyzed in the context of joint/disjoint perspective; and second, it provides a reference for the reader to quickly identify the experience managers during the analysis in later sections.

From this analysis we can conclude that the majority of the systems use a joint perspective in the development of the experience manager and environments. This is not surprising, as the joint perspective in the development is the most common approach in the literature. However, it is interesting to note that there are some systems that used a disjoint perspective. This is a good indication that the joint perspective is not the only possible approach, and that there is room for further research in this area.

Moreover, when further analyzing the systems that use the disjoint perspective in the development, we can extrapolate that the current version of separation between experience managers and environments it is made by the environment itself. With this methodology, the environment can be used by multiple experience managers. However, in case we want to use the same experience manager with different environments, we would need to rewrite the experience manager (almost) from scratch. The ability to reuse an environment with different experience managers a great step in the right direction, as it would allow us to test different experience managers in the same environment. Another added benefit is the ability to do rapid prototyping of new experience managers, as we would not need to create a new environment for each new experience manager.

My objective with this dissertation is to explore the possibility of using a disjoint perspective in the development of experience managers and environments

Citation	Author(s)	Year	System Name	J/D perspective
[55]	Magerko	2005	Interactive Drama Architecture (IDA)	Joint
[78]	Nelson and Mateas	2005	Anchorhead with SBDM	Joint
[82]	Nelson et al.	2006	Anchorhead with DODM	Joint
[68]	Mehta et al.	2007	Façade	Joint
[131]	Thue et al.	2007	PaSSAGE	Joint
[85]	Peirce et al.	2008	ALIGN	Joint
[18]	Cheong and Young	2008	Suspenser Framework	Joint
[20]	Corradini et al.	2009	Anchorhead	Joint
[120]	Sullivan et al.	2009	EMPath	Joint
[116]	Sharma et al.	2010	C-DraGer	Joint
[6]	Arinbjarnar and Kudenko	2010	Directed Emergent Drama (DED)	Disjoint
[137]	Tomaszewski	2011	Marlinspike	Joint
[50]	Lee et al.	2014	ML in Crystal Island	Joint
[27]	Endrass et al.	2014	Virtual Beer Garden	Disjoint
[40]	Harrison and Roberts	2014	Sidequest: The Game	Joint
[89]	Poo Hernandez et al.	2015	PACE	Joint
[154]	Yu and Riedl	2015	Personalized Drama Manager (PDM)	Joint
[94]	Ramirez and Bulitko	2015	Player-specific Automated Storytelling (PAST)	Joint
[142]	Ware and Young	2016	The Best Laid Plans	Disjoint
[140]	Wang et al.	2017	Deep RL in Crystal Island	Disjoint
[23]	De Lima et al.	2018	-	Joint
[16]	Braunschweiler et al.	2018	-	Joint
[143]	Ware et al.	2019	-	Disjoint
[28]	Farrell et al.	2019	Indexter	Disjoint

Table 4.1: Overview of the papers analyzed in this chapter. The column “J/D Perspective” indicates if the system is using a joint or disjoint perspective [129] in the development of the experience manager.

not only from the environment side, but also from the experience manager side. Partial-interchangeability between experience managers and environments would take the disjoint perspective to the next level, as it would allow us to reuse the same experience manager with different environments, and vice versa.

4.3 GEM Framework

To answer the first research question stated in Section 4.1, I structure the discussion using Thue’s (2015) “Generalized Experience Management” (GEM) framework [127], as its generality has been demonstrated across several different man-

agers [127]. GEM defines experience management as the task of optimizing a player’s experience in an interactive environment by adjusting that environment while the player is experiencing it. GEM is a framework in the sense that it combines several conventional notions of EM into a well-defined formal structure, providing both a base and a collection of conceptual “building blocks”.³

The base of GEM can be summarized as follows. Given an interactive environment (e.g., a computer game), a player’s experience (called a *trajectory*) is a rotating sequence of the game’s *states* (which the player perceives), *actions* (which the player performs), and potentially new variations of the game’s *mechanics* (which determine future states). A gameplay *history* is a trajectory that starts from the beginning of the experience and ends at the most recent action the player performed. GEM allows an experience manager to modify a game’s mechanics, which makes it simpler to represent some managers using the GEM framework [127].

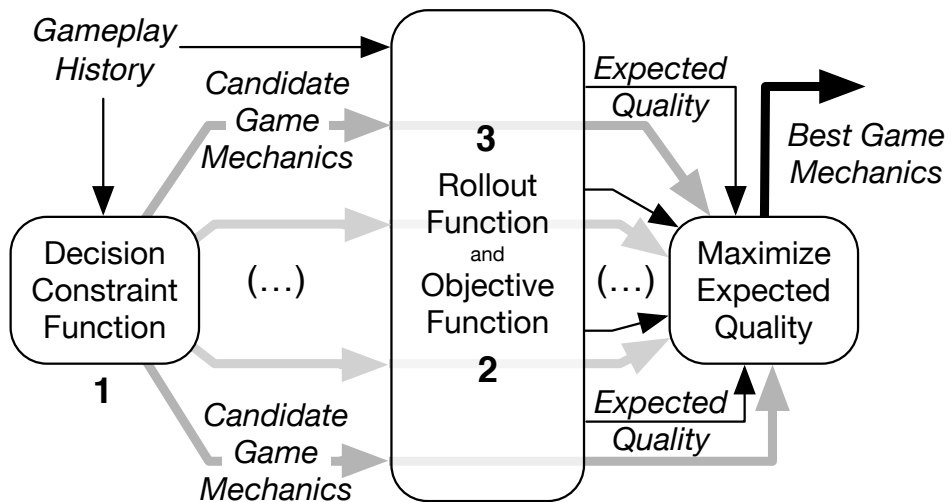


Figure 4.2: A schematic diagram of a GEM manager’s policy. Numbers identify GEM’s building blocks. Rounded boxes show functions, italics show data, and arrows show function inputs and outputs. The flow of game mechanics is highlighted with thicker arrows. See Figure 4.3 for more details.

As sketched in Figure 4.2, a GEM manager’s policy works to maximize the *expected quality* of the player’s experience by assessing several candidates for the game’s mechanics and choosing the best one. Figure 4.3 shows the assessment step in more detail, in which the potential futures that could result from each given candidate for the game’s mechanics are used to compute an expected quality. Quality is highest when the experience’s effect on the player is closest to what the manager’s designers intended.

GEM’s building blocks form the basis of our forthcoming discussion, and we

³To keep focus on the relevant *concepts* of GEM, we will not use GEM’s mathematical definitions in this work.

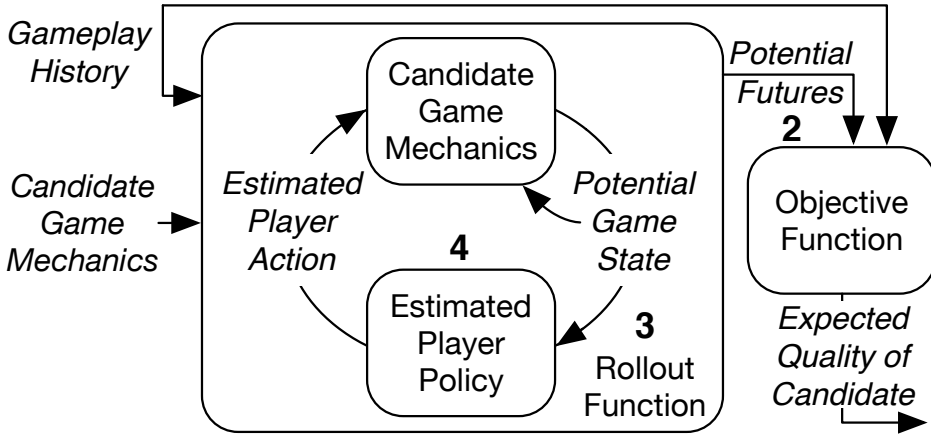


Figure 4.3: A schematic of how GEM’s rollout function, objective function, and estimated player policy are used to assess candidate game mechanics (see Figure 4.2 for context).

review their definitions briefly here:

- **Block #1 - Decision Constraint Function:** This allows the manager’s designer(s) to prevent it from considering different options for changing a game’s mechanics. For example, *Left 4 Dead*’s AI DIRECTOR was constrained to choose between only two versions of the game’s mechanics (spawning more and more zombies, and not) [66, 15]. Decision constraints can reduce the computational requirements of EM by decreasing the number of possible experiences that the manager needs to analyze.
- **Block #2 - Objective Function:** When multiple options for the game’s operation are available for the manager to consider, the manager might consult a designer-provided objective function. This function estimates the quality of a given trajectory, and can thus be used by the manager to estimate the results of its modifications. For example, *Façade*’s drama manager estimated how closely the player’s current experience of dramatic tension was following an author-defined curve over time [59, 62].
- **Block #3 - Rollout Function:** To obtain more useful estimates from an available objective function, a manager might use a rollout function to estimate potential futures of the player’s experience (for one or many steps). Weyhrauch (1997) illustrated this block with the drama manager MOE, which aimed to build a tree of possible futures to consider when selecting its next dramatic move.
- **Block #4 - Estimated Player Policy:** To increase the reliability of an available rollout function, the manager might use an estimated player policy to estimate which action(s) the player might perform next, given their prior gameplay history. An example of this block can be found in a study by Min

et al. (2016), where they attempted to model how players would form and pursue new goals based on their prior experiences in the game.

- **Block #5 - Feature Vector:** Given a trajectory of a player’s prior (or potential future) experience in the game, it is common for managers to extract higher-level information that can aid in their reasoning process. A feature vector is a collection of functions, each of which is responsible for extracting one piece of information from a given trajectory. For example, Barber and Kudenko (2007) used the player’s gameplay history to estimate several features in the form of a player model, including the player’s selfishness and faithfulness.

Figures 4.2 and 4.3 show schematic diagrams of the GEM framework in terms of how a GEM manager’s policy might operate. Given a gameplay history as input (Figure 4.2), the policy first uses the decision constraint function (Block #1) to obtain a set of possible candidates for game mechanics. Next, each candidate is sent to the rollout (#3) and objective (#2) functions for assessment, along with the gameplay history (Figure 4.3). Given a gameplay history and a candidate for the game’s mechanics, the rollout function generates a set of potential futures; it uses the estimated player policy (#4) to estimate subsequent player actions, and the candidate game mechanics (now as a function, rather than data) to compute subsequent game states. Alongside the player’s gameplay history, the set of potential futures is assessed by the objective function (#2) to calculate the expected quality that will result from choosing the given candidate game mechanics. If a manager is created without an estimated player policy, then the rollout function will be limited to producing potential futures that only extend one state past the given gameplay history. If a manager is created without a rollout function, then the objective function will estimate the candidate’s expected quality using only the gameplay history. After obtaining an expected quality for each candidate game mechanics (Figure 4.2), the manager chooses a candidate that maximizes the expected quality and applies it to the game. All of GEM’s building blocks are complementary, and they can be used to categorize the techniques that various researchers have developed in pursuit of better experience managers.

Block #5 (a feature vector) is left out of the figures to simplify their presentation. In practice, for any function that accepts a trajectory (such as a gameplay history or a potential future) as one of its inputs, a feature vector can be used when computing the function’s result.

In the context of this literature review, for each research work I analyzed, I examined whether (and if so, how) that work implemented each of the first four GEM building blocks. I decided to exclude the Feature Vector block because, after an in-depth analysis of the framework, I concluded that the functionalities that this block provides to the experience manager can be too wide to be classified for the context of this literature review. Furthermore, since it is used to help other blocks in extracting high-level information from the simulation, it can be embedded directly in the other blocks. The analysis is structured as follows: for each block, I introduce the most common components I found during the review, then I list in which paper(s) I found them and how they were implemented.

4.4 Decision Constraint Function

In this section, I analyze the papers introduced in Section 4.2, specifically concerning GEM’s Decision Constraint Function (Section 4.3, Block #1). I read the full text for each of the papers to identify whether the described experience manager contained a component that constrained the manager’s available decisions. If it did, I analyzed how the authors designed that component.

4.4.1 Analysis

To facilitate clear comparisons, I briefly describe a set of common design patterns that I found during the analysis. There are four different ways in which decision constraint functions were designed: *plot points*, *preconditions and effects*, *graph representation*, and *hand-made functions*.

Plot points represent significant game events such as the player fighting with an NPC or finding a combination for a chest [103]. Thue and Carstendottir (2018) described that there are two concurrent definitions of a plot point in the literature: *player-focused*, and *character-focused* plot points [130]. A player-focused plot point represents something that the player does that is narratively important. A character-focused plot point represents something that characters (NPCs or players) do that is narratively important. Throughout this section, when a paper uses plot points to constrain the experience manager’s decisions, I will specify whether they are player-focused or character-focused.

Preconditions and effects help describe author-defined actions that can happen in the game world based on how they change an abstract representation of the game’s state. An action’s preconditions must be verified on the game state. If all these conditions are met, the action can be performed in the current state. The effects of an action are a set of changes to the game state that happens after the action is executed. This allows the experience manager to maintain an abstract representation of the game’s current state by keeping track of what changes in the environment every time something relevant happens. For example, consider the action “move(player, roomA, roomB)”. This action represents when the player moves from roomA to roomB. The preconditions are that the player has to be in roomA, and the player has to be alive. After the player acts, the effects are that the player leaves roomA and enters roomB.

Another typical design pattern for decision constraints is the use of graphs. Graphs can be combined with other methods to describe how the narrative world can evolve to different possible states over time. Graphs are composed of nodes and edges that connect pairs of nodes. In this analysis, researchers have used graphs in two ways. First, a node represents a world state, while an edge represents an action that can change the state of the world to another state. Second, a node represents an action or an event that can occur in the game, and an edge represents a transition (as an ordering constraint) to a subsequent action (node) that can happen. Graphs can be used to analyze how a story evolves in the future, such as whether a particular set of actions leads to loops of execution of the same actions repetitively.

The last common design pattern found in the analysis of decision constraint functions is using hand-made functions to restrict the set of possible candidate options the manager can choose from. When an author uses a hand-made function, they intend to manually specify which options should be available to the manager at each point during the experience.

Table 4.2 summarizes each system’s design patterns to implement the decision constraint function. The analysis is organized by following the general structure of the table.

Papers	Plot point		Preconditions and Effects	Graph Representation	Hand-made Function	Other
	Player Focused	Character Focused				
Nelson and Mateas (2005) [78]	✓					
Lee et al. (2014) [50]		✓				
Mehta et al. (2007) [68] Tomaszewski (2011) [137] Ware and Young (2016) [142] De Lima et al. (2018) [23] Farrell et al. (2019) [28]			✓			
Endrass et al. (2014) [27] Yu and Riedl (2015) [154]				✓		
Peirce et al. (2008) [85] Ramirez and Bultko (2015) [94] Thue et al. (2007) [131]					✓	
Corradini et al. (2009) [20]		✓	✓			
Nelson et al. (2006) [82] Sullivan et al. (2009) [120]	✓			✓		
Magerko (2005) [55] Sharma et al. (2010) [116]		✓	✓	✓		
Poo Hernandez et al. (2015) [89] Braunschweiler et al. (2018) [16] Ware et al. (2019) [143]			✓	✓		
Harrison and Roberts (2014) [40] Arinbjarnar and Kudenko (2010) [6]						✓

Table 4.2: Overview of all the papers cited during the analysis of the decision constraint function block divided by categories.

4.4.1.1 Plot Points

Nelson and Mateas (2005) used the SBDM framework [147], and one of the first tasks when using this framework is to abstract the story’s content into discrete player-focused plot points [78]. Each plot point is assigned ordering constraints to constrain the drama manager’s decisions.

Character-focused plot points have also been used to represent narrative progress in Lee et al.’s (2014) work [50]. Specifically, they used the narrative progress as part of the observation that a dynamic Bayesian network employed to decide the intervention strategy. The narrative progress consists of a discrete variable representing Crystal Island’s plot structure and is used to limit the options that the EM can consider.

4.4.1.2 Preconditions and Effects

In Mehta et al.’s (2007) work, the AI manager in Façade dynamically determines a sequence of dramatic situations (called “beats”) between two virtual actors and the player [62, 68]. The beat preconditions implement a decision constraint function, as shown in previous work [77].

With *Marlinspike*, Tomaszewski (2011) used scenes with preconditions to limit the number of scenes that the manager can choose from [137]. To play a certain scene, all the preconditions must be satisfied. A scene is an extension of the story; it directs NPCs on how to act and affects world objects.

In *The Best Laid Plans*, Ware and Young (2016) used actions with preconditions and effects to limit the possible options that the experience manager has to choose. Preconditions and effects were also used in the description of “steps” that define a sequence of actions that need to be executed together.

Farrell et al.’s (2019) work is based on STRIPS, meaning that each event has preconditions that must be true before execution and effects that modify the world state. In addition, they included some parameters that each event should include, such as space (where an event can be executed) and time (when an event can be executed). The use of preconditions and effects are used to limit the manager’s options.

De Lima et al. (2018) implemented the decision constraint function using precondition and effects on the quest planner that is part of the quest manager. They used this technique for operators and actions to limit the possible quests from which the manager must choose.

4.4.1.3 Graph Representation

Endrass et al. (2014) [27] implemented a decision constraint function using a graph-like representation with nodes and edges called a sceneflow. A sceneflow, a hierarchical and concurrent state chart, specifies the logical and temporal sequence in which scenes are run. In executable programs built in a domain-specific scripting language, nodes are structured hierarchically and represent distinct dialogue contexts or phases of discourse. Transitions between nodes in the sceneflow are called edges, and they are frequently protected by temporal restrictions or conditional expressions. In this case, the graph representation limits what the manager can work with at any time because each edge uses temporal constraints or conditional expression.

Yu and Riedl (2015) use as a decision constraint function a story graph to reduce the set of actions that the manager has to choose from before it calculates the one with the highest expected enjoyment [154]. This system has already been represented using GEM blocks in prior work [77].

4.4.1.4 Hand-made Function

The decision constraint function in Peirce et al.’s (2008) system is conceived as a two-stage process based on game feasibility and game appropriateness [85]. The game feasibility constraints eliminate all adaptive elements that are not available

due to the current game state and context. The appropriateness constraint allows adaptive elements to be constrained based on desirable NPC behaviour by not allowing them to repeat or contradict themselves. This process can be considered as an approach that uses a hand-made function to limit the manager’s options.

Ramirez and Bilitko (2015) implemented the decision constraint function with a hand-made function, based on the gameplay history, to identify exceptional player actions and give *PAST*’s planner access to a planning domain [94].

A hand-made function with preconditions and effects is the approach that Thue et al. (2007) used in their system *PaSSAGE* [131]. When analyzing *PaSSAGE* using GEM blocks, Thue (2015) said that the author is ensuring that only certain actions are available at any given point in time depending on the player history [127].

4.4.1.5 Plot Point and Preconditions and Effects

Corradini et al. (2009) used an approach where player-focused plot points were defined with a set of preconditions, a trigger (e.g., a player action), and a set of effects corresponding to responses from characters in the game [20].

4.4.1.6 Plot Points and Graph Representation

Nelson et al. (2006) used plot points to represent story events that the drama manager should know about [82]. From the text of the paper, the plot points they use are player-focused since all the actions represent a model of what the player can do. DODM also uses directed acyclic graphs as a method of ordering constraints, with nodes representing plot points and edges specifying a partial ordering.

Player-focused plot points and graph representation were similarly used by Sullivan et al. (2009). They organized plot points into a directed acyclic graph to capture ordering dependencies enforced by the game world’s logic. For each plot point, they defined a list of drama manager actions that can influence a plot point occurrence.

4.4.1.7 Plot Points, Preconditions and Effects, and Graph Representation

In the *Interactive Drama Architecture* system, Magerko (2005) defined the decision constraint function using a directed graph where nodes are character-focused plot points, and edges connect the plot points in a partial order [55]. Plot points are defined with a set of preconditions that must be true to execute the plot point, actions executed once the preconditions are fulfilled, and a timing constraint associated with the plot point.

Sharma et al. (2010) limited the options the manager has to choose between using a combination of three patterns: plot points, actions with preconditions and effects, and graph representation [116]. They used plot points as nodes of a directed graph representing how the story state can progress. They use plot points to focus on the narratively important player actions, so they are considered player-focused plot points.

4.4.1.8 Preconditions and Effects, and Graph Representation

In *PACE*, Poo Hernandez et al. (2015) used a narrative graph encoded as states and actions using the PDDL [67] for the generation of candidate narratives [89]. PDDL defines actions in terms of preconditions and effects on the state, and the preconditions are used to constrain the actions presented to the manager.

In Braunschweiler et al.’s (2018) system, the *story domain representation* component of the EM can be identified as the decision constraint function [16]. It includes the world state that represents an abstraction of the state with a collection of story states and arcs. Each story arc is annotated with preconditions used in the story graph to annotate the entry point of the graph. So, the combination of preconditions and graph representation restricts the manager’s options to build an effective storyline.

Ware et al. (2019) divided the implementation of the decision constraint function into two parts: story domain and story graph [143]. Objects in the form of logical constraints and actions with preconditions and effects form the story domain. The story graph is composed of nodes representing the world’s state with objects and edges that are the actions that change the state.

4.4.1.9 Other methods

Another approach to achieve the objective of the decision constraint function is the one by Harrison and Roberts (2014). They used the current player’s sequence of quest interactions to represent the game state by employing a set of vanity analytics [40]. Vanity analytics are represented by data that have a great deal of explanatory and predictive power but are difficult to manipulate (e.g., how many enemies the player killed). The manager reads these analytics and uses them to constrain the options (quests) that it can choose from.

In the *Directed Emergent Drama* architecture [6], the director uses schemas to direct the drama and limit the options that the manager can consider. A schema is a narrative episode that is used by a director and a set of actors to structure the drama so that it emerges into a fully developed drama. Each schema has a finite set of roles annotated as essential or non-essential. Each role is annotated with a finite set of characteristics that it supports.

4.4.2 Discussion

The purpose of the decision constraint function is to simplify the manager’s work by reducing the available alternatives for it to choose following different player histories. This functionality is a fundamental part of the GEM framework; for this reason, the analysis showed that all the papers analyzed had a way of limiting the available actions that the manager has to consider at any time. As summarized in Table 4.2, there are four categories of how different experience managers developed this functionality.

- **Plot points:** they represent important events of a story. The advantage of this technique is that it makes an abstract representation of the story where

the author has to define each event manually, allowing precise control of the story. This advantage comes at the cost of more work for the story's authors since they have to define each plot point manually.

- **Preconditions and effects:** the preconditions describe conditions that an action has to fulfill to be applied, and the effects describe how the world state will change after the action is executed. The advantage of using this representation is that authors have less work to do since they need to define only the general actions (e.g., move, talk, etc.), all the characters involved in the story, and an initial state of the world to have a story to work. Once the system has an overview of everything possible, it can generate how the story evolves. This comes at the cost of higher computation time, and the author has less control over what can happen.
- **Graph:** It represents how the story can evolve. Using a graph-based technique allows an easy representation of the history and a convenient way to limit the number of options an experience manager has to choose. The disadvantage of using graphs is that they can add complexity to the system when the graph increases the number of nodes and edges, and the developers need to implement techniques to limit its size or explore it efficiently.
- **Hand-made function:** the author hand-authors mapping between histories and sets of candidates for the manager to consider. Using this technique, the designer directly controls how the experience manager decides which actions to consider. This comes at the cost of work overhead for the authors since they have to map every possible history in the story to a set of manager operations.

By further analyzing Table 4.2, we can observe that some systems use hybrid techniques by combining different approaches. For example, two of the papers we analyzed ([16, 143]) use graph representations where each node in the graph represents actions with preconditions and effects. These hybrid techniques allow the designer to achieve a better result with a finer-grained ability to reduce the manager's options.

Preconditions and effects was the most common design pattern, with ten systems using them in various forms. The reason behind this broad use might be the capability of representing actions easily and predictably, with preconditions that check if the action can be applied in the state and the effects used to change the state after the action is applied.

4.5 Objective Function

As we described in section 4.3, an experience manager might use a function created by the game designer to evaluate different possible changes to the game mechanics. This objective function can predict the outcome of different decisions, and the manager can use it to make informed choices. In this section, I analyze the papers found in the literature to understand how the objective function is implemented.

4.5.1 Analysis

For the analysis of the objective function, I extend the classification used in my previous work [77] to include new types of objective functions found during this study. The classification is divided into two parts. The first part describes the *nature* of the data that the objective function considers when evaluating the options. The nature of the data is divided into two categories: *experience's structure* and *direct effects on player*. I classified the nature of the data *experience structure* when it involves information about the structure of the experience, such as scoring an action higher because it will lead to fulfilling a certain precondition of another action. The data nature is classified *direct effects on player* when it involves information about the direct effects of the actions on the player, such as scoring an action higher that will lead to better player enjoyment.

The second part of the classification describes the *source* of the data that the objective function considers when evaluating an option. The source of data is divided into four categories: *authored annotation*, *machine-learned*, *player model*, and *structural analysis*. The source of data is classified *authored annotation* when the designer manually annotates the data to be used by the objective function. The source of data is classified *machine-learned* when the objective function uses a machine learning algorithm to learn the data to be used by the objective function. The source of data is classified *player model* when the objective function uses a model of the player to estimate the quality of the different options. The source of data is classified *structural analysis* when the objective function uses the structure of the experience to evaluate the option.

Table 4.3 shows an outline of all the systems that will be analyzed in this section, arranged by the design pattern that they employed in the implementation of the objective function. The analysis in the remainder of the section is structured by grouping the papers using data with the same nature and dividing them by their source of data.

4.5.1.1 Experience Structure

During the analysis, I identified that nine of the papers analyzed used the structure of the experience to evaluate the manager's options.

Authored Annotation. The objective function in Arinbjarnar and Kudenko's (2010) system allows the director to choose the best schema to allow the actors to show their characteristics [6]. The director uses the "dramatic arc" [48] technique to develop an interesting drama and choose the best action.

Machine-learned. The objective function in Lee et al.'s (2014) system is the part of the dynamic Bayesian network that receives the observation function, changes the marginal probabilities of connected nodes, and performs inferences about the most probable intervention decision [50]. The intervention decision encodes when the director should intervene at a given time.

Papers	Data Nature		Source of the Data			
	Experience's Structure	Direct Effects on Player	Authored Annotations	Machine-learned Source	Player Model	Structural Analysis
Arinbjarnar and Kudenko (2010) [6]	✓		✓			
Lee et al. (2014) [50]	✓			✓		
Corradini et al. (2009) [20] Sullivan et al. (2009) [120] Farrell et al. (2019) [28]	✓					✓
Cheong and Young (2008) [18] Ware and Young (2016) [142] Braunschweiler et al. (2018) [16]	✓		✓			✓
Sharma et al. (2010) [116]	✓				✓	✓
Tomaszewski (2011) [137]		✓	✓			
Peirce et al. (2008) [85]		✓			✓	
Harrison and Roberts (2014) [40] Yu and Riedl (2015) [154]		✓		✓	✓	
Thue et al. (2007) [131]		✓	✓		✓	
Mehta et al. (2007) [68]	✓	✓	✓			
Magerko (2005) [55] Poo Hernandez et al. (2015) [89]	✓	✓	✓		✓	
Nelson and Mateas (2005) [78] Nelson et al. (2006) [82]	✓	✓	✓			✓
Ramirez and Bulitko (2015) [94]	✓	✓	✓		✓	✓

Table 4.3: Overview of all the papers cited during the analysis of the objective function divided by categories.

Structural Analysis. The EM in Corradini et al.’s (2009) system rates each candidate action with a function that evaluates the player satisfaction using a combination of player model and predefined story aesthetic rules [20]. These aesthetic rules derive from Weyhrauch’s [147] work and are thought flow, activity flow, and manipulation. These metrics are calculated based on the analysis of the story structure. They monitor that the story does not alternate too much among different sub-plots, that the actions do not alternate too much among different locations, and that the player does not feel manipulated.

Sullivan et al. (2009) used an author-specified evaluation function that rate four features: motivation, thought flow, manipulation, and story density [120]. Thought flow and manipulation were the same as Corradini et al. (2009), whereas motivation measures whether events that happen in the world are motivated by previous events and story density measures the frequency and intensity with which a group of plot points occurs.

Farrell et al. (2019) use as an objective function a function that calculates the salience of past events based on the presence of shared indices with the event currently being perceived [28]. To do so, they use an equation derived from five indices of the Event-Indexing Situation Model [157]: protagonist, time, space, causality, and intentionality.

Authored Annotation and Structural Analysis. Cheong and Young (2008) developed an objective function using three heuristic functions: one to compute the suspense level of a given partial plan, and the other two together to compute the potential suspense of action in a plan [18]. The first heuristic derives from the notion articulated by Gerrig and Bernardo [35], calculating the inverse of the number of planned solutions for the protagonists’ goal. The other heuristics compute the potential suspense for an action by counting the number of its effects that both negate the player’s goal and align with the protagonist’s goal but also reveal new information that was not previously known by the audience. This is an example of an objective function developed as a heuristic function using rules derived from structured analysis (the first heuristic) and authored annotations (the second heuristic).

In the *Best Laid Plans* [142], an objective function is used to ensure that the stories the narrative planner generates meet the audience’s expectations. To do so, they use Riedl and Young’s (2010) model of agent intentionality and Ware’s (2011) model of conflict. In Riedl and Young’s (2010) model, each action is annotated with a list of characters that need to give permission to execute the action (authored annotations). Meanwhile, in Ware’s (2011) model, the planner marks actions that the character wanted to execute but it could not due to conflicts with other characters (structural analysis).

In Braunschweiler et al.’s (2018) story graph-based system, the objective function can be found in the component called “trajectory planner” [16]. The trajectory planner aims to score possible paths through the story graph according to how well they fit trajectories defined by the author. To do so, the heuristic function they implemented to rate each node during the search process bases its score on three values: engagement value for contained event instances, engagement value for par-

ticipants, and the story arc.

Structural Analysis and Player Model. Sharma et al.’s (2010) system uses an objective function to maximize player interest and narrative coherence in the story arc selection [116]. To do so, they use a combination of author-specified story guidelines such as thought flow, activity flow, and manipulation [147] to track the narrative coherence (structural analysis), and a player interest model to estimate the player’s interest (player model).

4.5.1.2 Direct Effects on Player

During the analysis, I identified that five of the papers analyzed used the direct effects on players to evaluate the manager’s options.

Authored Annotation. Tomaszewski’s (2011) system uses an objective function that is based on an author-specified import value that each action has encoded in the definition which indicates how narratively exciting the action is [137]. So, the manager uses this value to rate each action.

Player Model. The experience manager in Peirce et al.’s (2008) system rates each of the possible candidate actions based on a set of rules derived from abstracted pedagogical rules that examine a player model and rate each possible adaptation [85].

Machine-learned and Player Model. Harrison and Roberts (2014) developed a objective function by calculating each player’s retention probability based on the quest they chose [40]. To calculate this probability, they use an equation that depends on the class of the player (that describes if the player is about to quit the game), the last n events, and the turn. They use this calculation to adapt the quest proposed to the player by NPCs. The AI will assign the quest with the highest probability of retention to the closest NPC to the player.

The personalized drama manager [154] worked on computing the expected value of each of several potential futures, each of which began with a set of customized actions that the PDM would present to the player [77]. To perform this computation, it combined models of the player’s preferences and policy to assess the quality and probability of each trajectory that could be reached.

Authored Annotation and Player Model. Thue et al. (2007) used an objective function to estimate how much fun the player will derive from their experience in the game [131, 127]. This estimation is based on potential courses of action identified by the designer and augmented with a player model.

4.5.1.3 Experience Structure and Direct Effects on Player

Six of the systems analyzed used the combination of the experience’s structure and direct effects on the player to evaluate the options.

Authored Annotation. In *Façade* [60], before execution, the designer provides an estimate of how much tension each beat will add to the story once executed, and they also provide a desired curve of tension versus time [68]. *Façade*'s manager, during execution, chooses between beats to minimize the distance between the given curve and an estimate of the amount of tension that is currently in the story.

Authored Annotation and Player Model. The experience manager in Magerko's (2005) system has two ways to rate the possible actions. One rating uses the predictions of the player model [55]. When the system has a reasonable hypothesis of the player's objective, it rates the future character's action higher to guide the player toward their objective. The second way to rate actions is by understanding which action execution can fulfill the precondition of the plot point that will need to be executed next.

Poo Hernandez et al. (2015) used an objective function to understand how desirable certain narrative goals are to the player [89]. They use a combination between a player model and author-provided mapping between play-style inclination and goal desirability.

Authored Annotation and Structural Analysis. Nelson and Mateas's (2005) system rates the candidate actions using an evaluation function that encodes the author's criteria for evaluating stories [78]. These criteria are: location flow, thought flow, motivation, plot mixing, plot homing, choice, and manipulativity. These criteria derive from a combination of authored annotations (thought flow, motivation, and manipulativity) and structural analysis of the story structure (location flow, plot mixing, plot homing, and choice). The author can build the evaluation function weighting from 0.0 to 1.0 each of these seven features (authored annotation). Nelson et al. (2006) also used the same technique, but they annotated each plot point with additional information used by the experience manager when choosing what to do next.

Authored Annotation, Player Model, and Structural Analysis. Ramirez and Bultko (2015) analyzed partial potential futures with a heuristic function [94]. The planning heuristic they used can be conceptually seen as a ranking over possible accommodations based on the product of the player's estimated agency (calculated with a mix of authored annotations and player model) and the candidate's proximity to the exemplar narrative (structural analysis).

4.5.2 Discussion

The manager uses an objective function to choose the best option to pick between available candidates. Evaluating an option is generally system-dependent because each experience manager focuses on a particular aspect of an experience. Despite this, my analysis found that designers used common techniques to understand which option is the most suitable when the system has to make a decision. From a first analysis of table 4.3, I can infer that with objective functions, there are many examples of the use of mixed techniques to achieve a better result in the evaluation.

When considering all the methods together, two distinctions can be drawn. The first concerns the nature of the data that the objective function considers: Does it concern the experience's structure, direct effects on the player, or both? Structural assessments offer a more convenient way to perform evaluations since running user studies with real players might not be required [81]. Unfortunately, the results of such evaluations suffer from difficulties with generalization – especially when having an effect on players is the manager's ultimate goal.

The second distinction concerns the source of the data that the objective function receives: Does it come from authored annotations, machine-learned sources, player models, structural analysis, or a combination? The most common design pattern used among the systems analyzed in this work is the authored annotations, with 13 uses. Authored annotations have similar benefits and limitations as explicit decision constraints; they are simple to specify and offer absolute control, but their associated authorial leverage⁴ is low. We can deduce that since the evaluation of a specific option is highly related to the objective of the experience manager, the authors must define rules to allow this process to happen. To do so, they can make the rules themselves or search the literature for established methods. However, finding methods to evaluate certain aspects of the experience is not always possible, so the authors need to create new methods themselves. Another reason that authored annotations are popular might be that evaluating a particular aspect of the experience can be the central aim of the research that the authors are trying to disseminate.

Using a machine-learned source of data offers the potential advantage of transferring learned information between experiences or games. However, the costs include less predictable manager behaviour and more complicated troubleshooting.

A player model is a source of data often used during the objective function evaluation process. The player model is a representation of the player's preferences and behaviour. The advantage of using a player model is that it can be used to evaluate the options in a more personalized way.

Another common pattern identified during the analysis is the use of structural analysis to evaluate the options. The advantage of structural analysis is that it can be used to evaluate the options, referring to data that can be extrapolated from the experience (e.g., occurrence of a certain action or plot point). It can leverage the knowledge of what happened and the current state of the experience to evaluate the options in a more informed way.

Objective functions are often created with a specific notion of quality in mind. However, even if a manager uses a particular technique to pursue its defined quality, that same method (perhaps with some adjustment) might help pursue other notions of quality as well.

⁴Authorial leverage refers to the amount of control an author has over the content of their work and how it is presented to readers.

4.6 Rollout Function

The rollout function is the part of the EM that aims to “roll out” possible futures of the player’s experience. Rolling out the future can be done for only one step after the current state of the environment, or for multiple steps if the system needs more complex predictions.

4.6.1 Analysis

During the analysis, I found different design patterns that the developers of experience managers used to implement a rollout function. I categorize them as follows: *tied with player model*, *planner*, *graph-based*, *search-based*, and *author-defined*. In Table 4.4, we can see a summary of the design patterns in the rollout function used by the experience managers analyzed in this work.

Paper	Design Pattern				
	Player Model	Planner	Graph Traversal	Search-based	Author-Defined
Thue et al. (2007) [131] Peirce et al. (2008) [85] Wang et al. (2017) [140]	✓				
Cheong and Young (2008) [18] Harrison and Roberts (2014) [40]	✓	✓			
Ware and Young (2016) [142] De Lima et al. (2018) [23] Farrell et al. (2019) [28] Ramirez and Bulitko (2015) [94]		✓			
Poo Hernandez et al. (2015) [89]		✓	✓		
Magerko (2005) [55] Ware et al. (2019) [143] Yu and Riedl (2015) [154]			✓		
Sharma et al. (2010) [116] Corradini et al. (2009) [20] Sullivan et al. (2009) [120] Braunschweiler et al. (2018) [16]			✓	✓	
Nelson and Mateas (2005) [78]				✓	
Nelson et al. (2006) [82] Arinbjarnar and Kudenko (2010) [6]					✓

Table 4.4: Overview of all the papers cited during the analysis of the rollout function divided by categories.

In this analysis, when a rollout function is tied with the player model, the system uses the player model to generate the potential future likely to happen based on how the player is expected to behave. This approach saves time during the computation of the subsequent actions because the system has to simulate only a smaller set of steps but at the cost of missing the evaluation of potential futures.

The second category is the rollout function that uses planners to achieve its objective. A planner uses an algorithm to transform an initial state into a desired goal state by applying available actions. Transforming an initial state into a goal

state generates middle states and the actions applied to generate middle states can be considered the set of future actions.

Then, some systems use graph traversal as a rollout function. As described in section 4.4, graphs are used to represent possible states that the world can evolve in the future. From any node of a directed graph, the system can check all the edges exiting the source node to find possible future states that the game can evolve from that part of the graph. When a system uses graph traversal techniques, it is common to generate potential futures only when needed because producing and working with a complete story graph is computationally expensive. Designers can apply many different techniques when generating potential futures “on-demand”. This analysis found that the most common are planning and search-based techniques. Planning was discussed in the previous paragraph, and we can apply the same principles to graphs. Search-based techniques use search algorithms to generate potential nodes between a starting state and a target or final state.

The last category of rollout function I found is the author-defined function. This is used when the author manually defines rules the system must follow when looking for potential future actions. An example is when the author decides to assume that when generating potential futures, every time a player has the opportunity to do something in the game, they will remain idle. Using these rules can simplify the computational work that the system needs to do when rolling out futures.

Going forward, this section will delve deeper into the papers outlined in Table 4.4, comprehensively examining each category and giving specific examples of the design pattern usage within each paper.

4.6.1.1 Player Model

A rollout function tied with a player model has been used in Wang et al.’s (2017) system, and it is developed as a module that predicts the player’s following action each time step [140]. This module is based on player data and a deep reinforcement learning-based narrative planner architecture. This can be considered a mix of rollout function and estimated player policy because it uses the player model to predict the player’s next action.

Another example of a rollout function tied with the player model is in Thue et al.’s (2007) system [131]. It starts from the current decision point and stops after it finds the next decision point or an ending of the game. The author identifies each decision point with weights that are used to rollout only the actions that might interest the player.

The experience manager developed by Peirce et al. (2008) uses a rollout function that can go ahead in time for one action. They examine a reader model and determine the best-suited adaptation outcome for the player.

4.6.1.2 Player Model and Planner

Cheong and Young’s (2008) drama manager uses the rollout function to validate the skeleton building phase of the system [18]. The system tests whether the skeleton is coherent from the reader’s perspective using an algorithm that uses the player model to generate complete plans to achieve the protagonist’s goals consistent with

the skeleton candidate. By generating the plans, the system looks into possible future states. It is a rollout function tied with the player model that uses planning to generate potential futures.

A similar approach is the rollout function in Harrison and Roberts’s (2014) system that uses the transition matrix derived from the player model to generate a sequence of actions that begins with the player accepting a quest and ends with them completing that quest [40]. The subsystem that generates future actions that can happen in the future is a rollout function that uses planning and player models to work.

4.6.1.3 Planner

PAST [94] used two types of planning-based rollout functions to generate potential futures. First, a domain defined a set of possible plans, including *partial* plans (i.e., those that did not reach a game-ending), which PAST explored as part of its search through potential possible futures [77]. Second, when the player performed an unplanned action, PAST engaged the planner to compute a set of accommodations satisfying the intermediate and final goals. By computing the accommodations set, PAST is rolling out potential futures.

In Ware and Young’s (2016) system, a planner is used to plan possible future actions, given some initial state, a goal, and a set of possible actions [142]. Planning the next actions to apply in the state to achieve a goal can be considered executing a rollout function.

The system described in De Lima et al.’s (2018) work uses a planning algorithm to search in the space of world states and actions for a sequence of actions that leads the player from the current state of the world to one of the quest’s goals [23]. By searching the space of the world states to achieve the quest’s goals, the planner is acting as a rollout function.

Farrell et al. (2019) use a planner to generate possible futures. Given a world in some initial state, a goal, and a set of possible actions, the planner finds a sequence of events which achieves the goal [28]. Their model is STRIPS-based (as described in section 4.4). This planner needs space and time to be described within each domain operator, to make the prediction coherent with the story.

4.6.1.4 Planner and Graph Traversal

In *PACE*, Poo Hernandez et al. (2015) the authors use a graph and a planning technique to rollout the possible futures that can happen during the experience [89]. As described in section 4.4, the graph is built using PDDL, and they use the Fast Downward planner [41] to generate narrative candidates.

4.6.1.5 Graph Traversal

Ware et al.’s (2019) use a full story graph to identify every possible node (that represents state) and every allowable directed edge (that represent actions that are applied to change the state) that a story can have [143]. In this case, when

generating the graph, they are rolling out all the possible futures that can happen from the initial state of the game.

Magerko (2005) uses a graph traversal as rollout function, in a graph with nodes identified as plot points and direct edges connecting them [55]. However, the link between the nodes does not represent a path in a story graph for the player to follow. They describe an explicit partial ordering of the plot content. Therefore, instead of being a normal graph traversal technique, this representation describes a space of possible topological orderings.

Yu and Riedl (2015) traverse a *multi-option branching story graph* when rolling out the future [154]. In fact, with a story graph, they can rollout the future by following the edges connected to the current state of the story. A peculiarity of this graph is that multiple options could point to the same plot point.

4.6.1.6 Search-Based

In Nelson and Mateas’s (2005) system, once plot points, DM actions, and an evaluation function are specified, the rollout function is a search through abstract plot-point space to find the optimal DM action for any given situation [78]. To perform the search, the authors first tried a complete search over all possible future combinations of DM actions, concluding that this search is too big to perform. Then, they tried the use of SAS and SAS+ algorithms.

4.6.1.7 Graph Traversal and Search-Based

The drama manager in Sharma et al.’s (2010) system uses search-based techniques with graphs to look ahead for a possible combination of player actions and predict the effects that different DM actions might produce. [116] The algorithm that it uses in the search is the *expectimax* algorithm [69].

A search-based technique for a rollout function was also used in Corradini et al.’s (2009) system [20]. The drama manager also uses the *expectimax* search algorithm to foresee the effects of different DM actions given all the possible actions a player can execute.

In Sullivan et al.’s (2009) system, the drama manager’s job was to choose actions (or no action) after the occurrence of every plot point to maximize the future good of the complete story [120]. This process was developed using the tree search algorithm *expectimax* to back up story evaluations from complete sequences of stories. The authors used a player model to optimize the search result to predict future player actions at the plot point level (more details in section 4.7).

Braunschweiler et al. (2018) used a graph traversal system that employed a search-based technique to traverse the graph and rollout the future [16]. The *trajectory planner* uses an exhaustive search to traverse the graph, representing each possible path as a series of values that can be fitted to a author defined trajectory.

4.6.1.8 Author defined

Nelson et al. (2006) used an author-defined rollout function that assumed an equal

probability of each action (where the ordering constraints are satisfied) happening in the future [82].

In Arinbjarnar and Kudenko's (2010) system, the rollout function relies on the process of choosing the optimal sentence to choose when the agent needs to decide on which speech action to take [6]. They use the relevance reasoning technique for Bayesian networks to find a set of sentences that satisfy the goal and are contextual to the input. When the Bayesian network checks if a certain sentence can satisfy the goal, it checks a potential future.

4.6.2 Discussion

An experience manager uses a rollout function when generating potential futures to find the best option aligned with the manager's objective. Generating potential futures to understand the best move in a game or similar application is a problem well-studied in AI research. For this reason, some of the techniques we found in our analysis were derived from AI research.

As described in the previous section and table 4.4, we found five categories of rollout functions in our analysis.

- **Player model:** When developing a rollout function, a player model-based approach uses a player model to generate potential futures that are most likely to occur based on the player's expected behaviour. This approach requires the system to simulate a smaller set of futures, saving time spent computing subsequent actions. However, this approach may fail to value futures that do not match the player model's expectations.
- **Planner:** The planner-based approach uses a planning algorithm to transform an initial state into a desired goal state by applying available actions. This approach generates intermediate-states that can be used as potential futures, providing a tool for the developers that can be classified as a rollout function. The advantage of using a planner is that they are quite flexible and scalable since it can be used to handle a wide variety of problem domains, can be easily scaled up to handle large state spaces, and can be easily adapted to different environments and use cases. The disadvantage is that this approach can be computationally expensive and may require more resources.
- **Graph Traversal:** The graph traversal approach uses a graph data structure to represent and store the states and transitions of the interactive system. This approach can provide an efficient way to generate potential futures by traversing the graph. However, a graph can be challenging to maintain, especially when the number of nodes and edges becomes excessively large.
- **Search-based:** The search-based approach uses search algorithms to generate potential futures by exploring the state space of the interactive system. The advantages of this approach are that it can be applied to a wide variety of problems and can be easily adapted to different environments and use cases, it can handle large state spaces and complex or dynamic environments, and it can provide a solution when searching for potential futures if one exists. This

approach may have some drawbacks, such as high computational complexity and requiring more resources than other approaches. Additionally, it may not always provide the best solution, especially when the state space is too large or complex, resulting in missing potential futures.

- **Author-defined:** The author-defined approach relies on the experience manager’s developer to define the rules for generating potential futures. The advantage is that the designer has complete control over how the future is generated, potentially allowing the system to generate only states needed for the task. The disadvantage is that author-defined systems are usually tailored to a particular system, so it is difficult to generalize and apply the same approach to other experiences.

From table 4.4, we can see that it is common to use multiple methods together to improve the rollout function’s generation of potential futures. For example, using a player model in combination with a planner is a solution that can reduce the work that the planner has to do when generating potential futures because it can focus only on the generation of actions that the user is likely to do.

Between the design patterns found in this analysis, I have not identified a single most common technique used to achieve the objectives of the rollout function block. Player model, planner, and graph-based are the most used design patterns within the five categories we found, with six, seven, and eight uses. The rollout function is highly dependent on the design of the other parts of the system. For example, suppose the system does not have an estimated player policy. In that case, it cannot have a rollout function tied to the player model, even if this design pattern has the characteristics the designer is looking for.

4.7 Estimated player policy

The estimated player policy is the part of an experience manager used to predict how the player might behave during a simulation of the potential future. When rolling out the future, the experience manager might need to know what the player might do in certain situations. This will avoid unnecessary computation in actions that are not likely to happen and tailor the experience to the type of user.

4.7.1 Analysis

In this analysis, three design patterns of estimated player policy were identified, and they can be categorized in *rule-based*, *history-based*, and *planning-based* methods. Table 4.5 summarizes the analysis of the estimated player policy, dividing each of the papers into the category of the design pattern they use.

A rule-based estimated player policy is a way to describe the player’s behaviour based on a set of predefined rules and assumptions about the player’s behaviour and knowledge. These rules simulate the player’s behaviour and predict their actions within the interactive drama or game. The author of the experience manager system typically designs the rules that make up the rule-based estimated player

Paper	Design Pattern		
	Rule-Based	History-Based	Planning-Based
Nelson et al. (2006) [82]	✓		
Sharma et al. (2010) [116] Harrison and Roberts (2014) [40] Yu and Riedl (2015) [154] Wang et al. (2017) [140]		✓	
Magerko (2005) [55] Thue et al. (2007) [131] Poo Hernandez et al. (2015) [89] Braunschweiler et al. (2018) [16] Ramirez and Bulitko (2015) [94] De Lima et al. (2018) [23]	✓	✓	
Cheong and Young (2008) [18]			✓
Sullivan et al. (2009) [120]		✓	✓

Table 4.5: Overview of all the papers cited during the analysis of the estimated player policy divided by categories.

policy. For example, an author can assume that the player will be idle every time the rollout function encounters a player input.

A history-based estimated player policy predicts the behaviour of the current player by analyzing the history of their actions and (in some cases) using data gathered from previous players' experiences. This data is used to build a player model for the current player and encode their game-playing characteristics, such as interests and preferences.

A planning-based estimated player policy uses planning algorithms to predict the player's behaviour. The planner generates a plan composed of the player's actions to satisfy the player's goals while considering the game's objectives and current state. The generated plan is then used to predict the player's behaviour by assuming that the player will follow the plan.

4.7.1.1 Rule-based

In their system, Nelson et al. (2006) use a relatively simple estimated player policy [82]. They assume that the player has no particular knowledge of the story or the author's goals and is acting randomly to explore the story. With this method, all the possible plot points are equally likely to occur at any given point in the story.

4.7.1.2 History-based

The *Player Modeling Module* in Sharma et al.’s (2010) system is used as player policy, and it constantly builds and maintains a player model for the current player [116] using Case-Based Reasoning [1]. It encodes the game-playing characteristics of the current player and maps them to plot points to predict an associated interestingness value. Once enough data is stored, the player model predicts the player’s interests in the future plot points to understand the most likely plot point to occur next.

In Harrison and Roberts’s (2014) system, the player model is used to model session-level retention in the game analytic space [40]. To model the session-level retention, their approach is two-fold (as described in section 4.5). First, they calculate the probability that a player quits the game using an equation with one free parameter. The free parameter is calculated using a player model formed during a user study, where they deployed the game online and were asked to play until completion. Then, they used the data from the user study to choose the best parameter. In this case, the estimated player policy is composed by the player model combined with the objective function to calculate the probability that the player continues to play the game.

To estimate the player policy, Yu and Riedl (2015) use machine learning classification algorithms to predict which story branch the player will choose at any point in the story [154]. The model that they used is “prefix-based collaborative filtering” [152]. It predicts which action the player would likely choose at each potential state, with the objective of calculating the probability that the player would reach an ending of the story. When a new player enters the game, the drama manager collects data on the player’s preferences. Once the data is enough, they use three algorithms (Logit regression, Probit regression, and probabilistic Support Vector Machine) to train the branch transition probability model.

The estimated player policy in Wang et al.’s (2017) system uses a deep learning algorithm to simulate the player’s behaviour during the interaction with the interactive narrative planners [140]. To do so, they use a bipartite statistical model to predict and generate player actions and outcomes. The data where they trained the algorithm was generated from two human subject studies collecting data logs and questionnaires from the players. The player action simulator and outcome simulator are formulated as a classification problem. The algorithm had to choose between possible actions based on the previously collected data.

4.7.1.3 Rule and History Based

Magerko’s (2005) system uses a predictive player model to compare hypothesized player behaviour against a well-defined story space [55]. This player model is a rule-based simulation implemented within the director of the interactive drama system [56]. It tracks the player’s actions and hypothesizes what knowledge the player is acquiring. The director periodically uses the player model to simulate which action the player will choose in the world state and uses this information to predict whether the player’s actions will keep the plot from progressing.

PaSSAGE [131] has the objective of automatically learning the preferred playstyle

of the current player and uses this knowledge to adapt the content of an interactive story dynamically. During gameplay, *PaSSAGE* learns a player model expressed as weights for five styles of play (fighters, power gamers, tacticians, storytellers, and method actors). Before run-time, the designer identifies potential action courses and adds weight deltas, updating the model based on the player's actions. *PaSSAGE* approximates the probability that the player might choose an action by assigning a probability of 1 to the action that maximizes the inner product between the current player model vector and the authors annotations.

The player model used in *PAST* [94] and *PACE* [89] is the same as described in *PaSSAGE* [131].

In the *trajectory planner* of Braunschweiler et al.'s (2018) system, they estimated the player policy when calculating the optimal path that maximises the user engagement at any point of the story [16]. When the system chooses the optimal path, it predicts the actions that the user is more likely to choose in the story graph based on the engagement values of the story arc. The system observes the user according to their involvement with the smart objects (characters and objects in the story world) or events that occur. The involvement is captured in the form of engagement values calculated as a combination of the user's proximity to the smart objects and how often the user interacts with them. The user model then stores these values as an average of the measurements. The trajectory planner then compares these values with an author-defined trajectory to predict the optimal path for the user. The estimated player policy is a combination of history based (the engagement values are calculated based on the user's history) and rule based (the optimal path is compared with an author-defined trajectory).

The player policy in De Lima et al.'s (2018) system is composed of three components: observation, behaviours, and predictive function [23]. The observation component extracts information from the game data to enable the prediction how the player can behave. The behaviours component defines the model's output as the possible behaviours the model will predict. It uses a widely accepted theory about human personality: the five-factor model [37]. The model's output is represented by the five factors disposed on five behavioural axes, each within the interval of $[-1, 1]$. Then, they associated each of the five factors with a specific player behaviour. They implemented a single hidden layer Artificial Neural Network trained by an incremental backpropagation learning algorithm to associate the information extracted from the game data with the five factors model. Then they use the result of the prediction of the five factors to calculate the probability of the player choosing a specific action in the game.

4.7.1.4 Planning-based

In Cheong and Young's (2008) system, the estimated player policy is a planner that is used to generate complete plans to achieve the protagonist's goals [18]. In this case, the authors use the *Crossbow* planner (based on the *Longbow* planning system [149]) to keep track of all the constraints that derive from the game's history and the player's past actions, and generate a plan that fulfills the protagonist's objectives.

4.7.1.5 History and Planning Based

The player policy is a fundamental part of the system developed by Sullivan et al. (2009). The drama manager that performs the look-ahead search described in section 4.6, requires a player model to predict the probabilities of each player-focused plot point based on future player actions [120]. The authors developed three models to find the best player model for their system: the uniform player model, the Manhattan distance player model, and the world knowledge user model. The uniform player model assigns that each plot point has an equal probability of happening. When hints are active, it adds weights to the plot point that has been hinted out. The Manhattan distance player model uses the knowledge of the world known by the drama manager to calculate the probabilities of each plot point happening. It uses the Manhattan distance between the coordinates of each plot points in the game world multiplied by 1.5 to offset that the players cannot always move on a path equal to the Manhattan distance. Since players are much more likely to encounter plot points that are closer to them, the Manhattan distance player model assigns a higher probability to the plot points that are closer to the player. Finally, the world knowledge user model uses an internal representation of the world layout with the location of the plot points. Using this map, they traverse one hundred random walks, noting the first plot point encountered. Afterwards, they tally their numbers and normalize them to reach the final probability distribution used as the player model.

4.7.2 Discussion

The estimated player policy is used to simulate what the player might do in the future to adjust the experience to the player's preferences and reduce the number of states the system has to evaluate when generating potential futures. The estimated player policy can be implemented in different ways, depending on the type of information the system has about the player and the technique that the designers implemented to simulate the player's behaviour. The estimated player policy can be implemented using three main methods:

- **Rule-based:** an estimated player policy uses a rule-based method when it relies on assumptions about the player's behaviour and knowledge, which can be encoded into a rule-based simulation. These assumptions are made by the author of the experience, and they can be based on well-known theories about human behaviour (e.g., the five-factor model [37]). The rule-based method can be as simple as assuming that the player is going to choose a random action every time has to make a choice [82], or more complex based on the needs of the designer.
- **History-based:** an estimated player policy uses a history-based method when it relies on data from past actions of the current player or previous players' experiences to predict the player's behaviour. A history-based approach allows the experience manager to tailor the experience to the individual player by considering their past behaviour, interests, and preferences. With this approach, the EM can also adapt to the player's changing behaviour

and interests as the player plays the game. To predict the player’s behaviour based on the history, the EM can use various algorithms, such as case-based reasoning [116], machine learning [154], or deep learning [140]. However, the history-based approach also has some limitations, such as the need to collect data to make predictions, and it may not work for players whose behaviour differs from the majority of the players.

- **Planning-based:** an estimated player policy uses a planning-based method when it uses a planner to generate a plan to achieve the protagonist’s goals and uses that plan to make predictions of the player’s behaviour. A planning-based method can handle uncertainty and changing conditions by planning for different scenarios and adapting to the player’s actions. It can also make predictions that generalize well to new players or situations since it is based on the game objectives and dynamics. However, a planning-based approach can be computationally expensive because solving a planning problems requires a lot of computation.

From Table 4.5, we can see that is common for the research to use a combination of these techniques to simulate the player’s behaviour. In fact, the most common approach in this analysis is to combine a rule-based method with a history-based method. I can speculate that this is because combining the rule-based method and the history-based method allows the EM to make predictions based on the designer’s assumptions about the players’ past behaviour.

4.8 Discussion

In the past sections, we have seen that there are many different ways to build an experience manager. Each experience manager was developed to solve a specific problem, but the techniques implemented to achieve the objectives of each GEM building block were very similar. In fact, I was able to identify common ways with which each building block was developed, as I discussed at the end of each section. To bring this analysis one step further, I considered whether there were any commonalities between the different papers across multiple GEM building blocks. In this section, I discuss what we can learn by having a more general overview of all the systems and how each building block is built. To facilitate this analysis, I created a table where there are short descriptions of all the components of each papers (Table 4.7). To help clarify Table 4.7, I created a legend on Table 4.6 with the name of each of the components found in the analysis in the previous sections and the corresponding abbreviation used in Table 4.7. Between these abbreviations, I included the keyword “Not Found”, to mark cases where I could not find a description of the corresponding building block in the text of the paper or any related work made by the authors.

From the first look into Table 4.7, we can note something odd about two papers: Cheong and Young (2008) [18] and Wang et al. (2017) [140]. In the case of Cheong and Young (2008) [18], the paper does not describe how the decision constraint function was developed. As Thue described when presenting the GEM framework

Decision Constraint Function		Objective Function		Rollout Function		Estimated Player Policy	
Name	Abb.	Name	Abb.	Name	Abb.	Name	Abb.
Player Focused Plot Points	PFPP	Experience's Structure	ES	Player Model	PM	Rule-Based	RB
Character Focused Plot Points	CFPP	Direct Effects on Player	DEP	Planner	PL	History-Based	HB
Preconditions and Effects	PE	Authored Annotations	AA	Graph Traversal	GT	Planning-Based	PB
Graph Representation	GR	Machine-Learned Source	MLS	Search-Based	SB		
Hand-Made Function	HMF	Player Model	PM	Author-Defined	AD		
Other	Other	Structural Analysis	SA				

Table 4.6: Table legend for abbreviations of table 4.7.

[127], an experience manager may be considered as such only if the system has a decision constraint function or an objective function developed. This system has some way of limiting the options that the manager has to consider since, in the text, there are some cues to this aspect, but I could not find any details on its development. However, Wang et al.'s (2017) [140] paper does not provide any details of both the decision constraint and the objective functions. After further analyzing this issue, I concluded that this experience manager has the decision constraint function, but, as for the other case, it is not explicitly stated the method with which it was implemented it.

If we focus on the bigger picture, we can see that most of the papers I analyzed implemented different techniques. As seen in the previous sections, I could identify groups of techniques developed to solve similar problems within each GEM block. However, if we expand the analysis to the other blocks, we cannot find any experience manager that uses all the same techniques as any other one. This is interesting because it shows that there is no single solution to the problem of developing an experience manager, and there is an influence between researchers and their ideas. However, in the context of the problem that I am trying to solve, this shows that there cannot be a single solution that can fit all the existing experience managers and environments.

Another consideration is that it is common to use different design patterns to achieve the result of a building block. However, when combining more than one method, the system becomes more complex, and the developer must maintain different subsystems. Nevertheless, choosing the proper subsystems when developing an experience manager is a balance between fine-tuned controls, the complexity of the system, and run-time performance.

In the context of this dissertation, this analysis aims to identify the most important aspects to consider when designing a solution for the problem of separation and

Paper	Decision Constraint	Objective Function	Rollout Function	Estimated Player Policy
Nelson and Mateas (2005) [78]	PFPP	ES, DEP, AA, SA	SB	Not Found
Nelson et al. (2006) [82]	PFPP, GR	ES, DEP, AA, SA	AD	RB
Sharma et al. (2010) [116]	CFPP, PE, GR	ES, PM, SA	GT, SB	HB
Corradini et al. (2009) [20]	CFPP, PE	ES, SA	GT, SB	Not Found
Sullivan et al. (2009) [120]	PFPP, GR	ES, SA	GT, SB	HB, PB
Lee et al. (2014) [50]	CFPP	ES, MLS	Not Found	Not Found
Magerko (2005) [55]	CFPP, PE, GR	ES, DEP, AA, PM	GT	HB, RB
Braunschweiler et al. (2018) [16]	PE, GR	ES, AA, SA	GT, SB	HB, RB
Ware et al. (2019) [143]	PE, GR	Not Found	GT	Not Found
Poo Hernandez et al. (2015) [89]	PE, GR	ES, DEP, AA, PM	PL, GT	HB, RB
Endrass et al. (2014) [27]	GR	Not Found	Not Found	Not Found
Yu and Riedl (2015) [154]	GR	DEP, MLS, PM	GT	HB
Mehta et al. (2007) [68]	PE	ES, DEP, AA	Not Found	Not Found
Ware and Young (2016) [142]	PE	ES, AA, SA	PL	Not Found
De Lima et al. (2018) [23]	PE	Not Found	PL	HB, RB
Farrell et al. (2019) [28]	PE	ES, SA	PL	Not Found
Tomaszewski (2011) [137]	PE	DEP, AA	Not Found	Not Found
Ramirez and Bulitko (2015) [94]	HMF	ES, DEP, AA, PM, SA	PL	HB, RB
Thue et al. (2007) [131]	HMF	DEP, AA, PM	PM	HB, RB
Peirce et al. (2008) [85]	HMF	DEP, PM	PM	Not Found
Harrison and Roberts (2014) [40]	Other	DEP, MLS, PM	PM, PL	HB
Arinbjarnar and Kudenko (2010) [6]	Other	ES, AA	AD	Not Found
Cheong and Young (2008) [18]	Not Found	ES, AA, SA	PM, PL	PB
Wang et al. (2017) [140]	Not Found	Not Found	PM	HB

Table 4.7: Summary of all the components of each paper.

interchangeability between experience managers and environments. The relationship between these two components is critical for delivering experiences optimized for a certain metric, as experience managers rely on the environment to deliver the experience to the user. One crucial aspect of this relationship is the decision constraint function, which we can imagine as a boundary between the experience manager and the environment. In essence, this function defines the rules and constraints that limit the decisions the experience manager can make based on the current state of the environment. There are several types of decision constraints that experience managers can use, but one of the most common is preconditions and effects (PE), as we can see in Table 4.7. This approach involves representing the environment as a set of preconditions and effects, which describe the necessary conditions for a particular event to occur and the resulting consequences of that event. When an experience manager uses preconditions and effects, it implies that it needs a way to keep track of the abstract state of the environment and a representation that uses preconditions and effects to decide how the experience should

unfold.

4.9 Takeaways

The objective that I wanted to achieve with my analysis in this chapter was to answer two sets of questions. The first set of questions is stated at this chapter's beginning.

With which of Thue and Bultko's perspectives were the managers that I analyzed developed?

This first question was answered in Section 4.2.1. The most common perspective used to develop experience managers is the joint perspective. When analyzing the experience managers and environment developed using the disjoint perspective, we saw that achieving separation between the two systems is possible, and we have examples in the literature. In such cases, this separation was achieved by designing the environment to be used by multiple experience managers [142, 117, 156]. However, the highly-specific set of instructions that the EM needs to use to work with those environments prevents the development of experience managers that can be used across multiple environments. To achieve partial-interchangeability, more than interchangeability across multiple experience managers is needed; we also need a way to have interchangeability across multiple environments. To overcome this challenge, one potential solution is to use a design pattern that employs a third component that acts as a layer of separation between the environment and the EM. This third component would be distinct from the environment and the EM, reducing the likelihood of the design pattern being dependent on the specifics of any environment or experience manager. This would allow the third component to facilitate interoperability between different environments and EMs, enabling greater flexibility in using these systems. However, it is important to note that trade-offs may be associated with using a third component to achieve separation and partial-interchangeability. For example, adding an additional layer in the communication between two components can increase latency, potentially leading to performance issues. As such, I will need to evaluate the performance of this design pattern which employs a third component to ensure that it is appropriate for this specific use case. This evaluation can be done by implementing a real-world example of an experience manager and environment that uses this design pattern.

What are the main techniques that researchers have used to develop experience managers?

There are many techniques that can be used to develop experience managers, and they vary based on the objectives the researcher wants to achieve. With this analysis, I established that if we divide the purpose of each technique based on the GEM framework's building blocks [127], it is possible to show common trends of techniques used to develop experience managers. In most cases, experience managers work with abstract representations of the game environment (e.g., world

state, actions with preconditions and effects, plot points, and graphs) that are used to make decisions on how to adapt the game to the player. The adaptation varies based on the metrics the experience manager tries to optimize. To optimize these metrics, the experience manager might use techniques to generate potential futures to evaluate the impact of the experience manager's decisions. When generating potential futures, it might also try to predict the player's next actions to personalize the experience for the player further.

Through my analysis, I identified a crucial point of connection between an environment and an experience manager. Specifically, this connection is based on the GEM decision constraint function, which plays a vital role in how experience managers operate. In most cases, the environment is responsible for sharing updates with the experience manager regarding what is happening during the experience. This communication is made through a specific representation, which the decision constraint function uses to filter the available transition functions for the manager's policy to select the desired adaptation. In my solution, the messages exchanged between the experience manager and the environment must contain all the data required for the decision constraint function to work properly.

The second set of questions that I wanted to answer is stated in Section 1.3. These questions are more general and are related to achieving a separation between the experience manager and the game environment, toward achieving partial-interchangeability. The number at the beginning of each question corresponds to the number of the question in Section 1.3.

1. What are the key obstacles that must be overcome to accomplish the separation between experience managers and environments?

The main challenge is to design an architecture that allows the experience manager and environment to be developed independently and be used interchangeably. I have discussed this challenge in the first question answered in this section. As mentioned previously, I aim to use a design pattern employing a third component to separate the environment and the experience manager.

Designing a way for various experience managers and environments to communicate is another challenging task that requires careful planning. The challenge lies in the fact that there are numerous combinations of experience managers and environments, each with its unique set of requirements. Thus, designing a communication system that can accommodate every possible combination is a difficult task. However, with a well-defined set of constraints, it is possible to limit the number of available experience managers and environments while still creating a communication system that can support the most common designs. This approach makes the design process tractable. The literature review presented in this chapter provides valuable insights into the most frequently used EM's techniques. In the following question, I will use this information to identify the constraints that I must consider during development.

The last important aspect to consider when designing a communication system for experience managers and environments is the need for flexibility. In this case, the challenge is to design a communication system with a high degree of flexibility

that does not make obsolete any existing experience managers and environments developed with a previous communication design.

1.a. What are the constraints that need to be taken into account when developing the communication framework between the experience manager and the environment?

The most important constraint is that having a solution that fits all experience managers and environments cannot be achieved with the limited resources available. In the analysis, we have seen that the range of possible applications of experience managers is wide, and the requirements of each experience manager and environment are different. If I want to accommodate all the possible combinations of experience managers and environments, I will need to design a solution that is too complex and cannot be implemented in a reasonable amount of time. However, from this analysis, I identified a common relationship between experience managers and environments that could be a starting point: the manager needs an abstract representation of the environment to make decisions on adapting the game. As a result, the first constraint will be to limit the set of available managers to the ones that use an abstract state representation of the environment to make decisions. Based on the analysis presented in this chapter, this constraint does not seem to be too restrictive, as most experience managers use an abstract state representation of the environment to make decisions.

In the analysis, we have seen several ways to represent the abstract state of the environment. Supporting all these representations in the first version of the solution would be unfeasible, considering the limited amount of resources available. As a result, the second constraint will be to limit the set of available representations to those that use a world state of true facts and actions with preconditions and effects to represent the abstract state of the environment. An example of this type of representation is Planning Domain Definition Language (PDDL) that, as I describe in Section 5.3.1, is divided into two files (domain and problem), where the domain hosts the definition of all the actions available using preconditions and effects, and the problem file hosts the initial state of the world and the goal state as a set of relations that indicates true facts in the world. The combination of the first and second constraints gives me the initial requirements for designing a communication protocol that allows an experience manager and an environment to be interchangeable. During the communication, they will share updates on the world state and execute actions with preconditions and effects that affect the abstract and physical state of the environment.

Based on my analysis, one important factor to consider when examining the work of an experience manager is the timeframe in which decisions are made. Sometimes, the experience manager has pre-defined the decisions regarding how the experience will unfold. In these cases, the experience manager follows through with the pre-planned decisions during the execution of the game, ensuring that users have a consistent experience. However, there are other systems where the experience manager must make decisions while the experience executes. This happens with systems where the user's actions influence the course of the experience,

and the experience manager must adapt to the changing circumstances in real time. With this work, I am interested in the second type of experience managers, where the manager needs to make decisions while the experience is executing. As a result, the third constraint will be to design the communication to support the exchange of information between the experience manager and the environment in real time.

1.b. What are the characteristics that an experience manager needs to have to be able to work in different environments?

An experience manager capable of working in diverse environments requires a set of characteristics. One of the primary traits is the ability to adapt to the abstract state of the environment. This implies that the experience manager should be able to understand and operate with different domains and initial states that can vary between environments. Since each environment has its unique types of objects, predicates that describe the state of the world, and actions that the manager can perform, the experience manager must quickly adapt to these aspects and customize their approach to suit the environment. For example, an experience manager developed in combination with a game environment will always have the same environment, thus characters will always be represented by the same names and predicates. However, if the experience manager is developed with this design pattern, it will be able to work with different environments, thus the experience manager must be able to adapt to the different names and predicates used by the environment.

Additionally, the experience manager must possess functionalities that allow them to interpret the data the environment shares. In this case, the experience manager must have a good understanding of the communication protocol of the environment. They should be able to extract, process and analyze relevant data from the environment, including player actions, player progress, and game statistics. This will enable the experience manager to make informed decisions based on the collected data, ensuring that it can manage the player's experience based on the metrics it is trying to optimize.

Another essential trait for an experience manager is the ability to work while the game runs. It must make real-time decisions to adjust the environment as needed.

1.c. What are the characteristics that an environment needs to have to be able to work with different experience managers?

An environment designed to work with different experience managers must have specific characteristics that enable it to adapt and evolve based on the direction given by the experience manager through the game. The environment should provide an explicit declaration of the actions that can be executed, including their preconditions and effects, to help experience managers make informed decisions about the game's direction. Additionally, the environment must offer managers the information they need to make decisions on adapting the game to the player, such as player preferences, achievements, and challenges faced. This information can help managers tailor the game to better suit each player based on the metrics

the manager tries to optimize.

The environment must also accept instructions that enable the experience manager to change the game's content during a player's experience. For example, the environment should allow experience managers to move non-player characters (NPCs) or create new items to enhance the player's experience. This ability to make changes on-the-fly can help experience managers respond to players' actions and adapt the game to the player.

The environment must keep track of the state of the game world using an abstract representation to facilitate communication between the environment and experience managers. This representation should capture all relevant aspects of the game, such as the location and state of NPCs, items, and other game objects. By maintaining an abstract state representation, the environment can quickly and efficiently communicate changes to experience managers, reducing the time and effort required to update the game.

Finally, the environment should handle the game's core dynamics without the manager's intervention. This includes managing the game's physics engine, allowing interactions with basic items (e.g., opening a chest), and other core features that make the game world feel alive and immersive. By allowing the environment to handle these tasks automatically, experience managers can focus on high-level decisions regarding the game's direction, story, and overall player experience.

Summary

In this chapter, I analyzed a subset of experience managers from the literature based on two methodologies: Thue and Bulitko's (2018) joint and disjoint perspective, and Thue's (2015) GEM framework. From the first analysis, I found that most of the experience managers analyzed are developed using a joint perspective, where the experience manager and the game environment are developed without a clear distinction between their codebase. With the second analysis, I could not find any single, most common technique used to develop experience managers. However, with these two analyses, I gained enough data to identify the main challenges and constraints I need to consider when designing a system that allows the interchangeability of experience managers and environments.

Chapter 5

Proposed Approach

In an effort to address the issue of interchangeability between EMs and environments, this chapter presents a framework that researchers can use to develop EMs and environments that are practically decoupled from each other, and that can work interchangeably. The framework is designed based on the intuition that to fully decouple EMs and environments from each other, a middle layer is needed for defining clear boundaries and structured external communication. This intuition is supported by the results of the previous chapter (Section 4.9), where the analysis of the literature showed that to fully-decouple EMs and environments, a layer of separation is needed to remove the dependencies of the separation from both systems.

Creating a framework that can work with experience managers and environments is a complex task that requires careful planning and consideration of several key factors. One of the most important aspects to consider is the architecture and design of the framework. The framework should be designed to allow easy integration with existing systems and be flexible enough to accommodate the creation of new systems that may be developed in the future.

It is also important to consider the maintainability and extensibility of the framework. As the needs of researchers may change over time, it is crucial to ensure that the framework can evolve and adapt to these changes. This may include adding new features and functionality or integrating with new technologies as they become available. This will help ensure the framework remains relevant and valuable over the long term.

5.1 General Overview

The objective of this framework is twofold: (i) creating a middle layer between EMs and environments to ensure clear boundaries and (ii) regulating the communication between them. I designed the middle layer as an interface where an experience manager communicates with the environment via this layer. In the rest of this dissertation, I refer to this middle layer as the “communication platform”, “platform”, or with the name of the project: “EM-Glue”. Every communication between the

two systems (experience manager and environment) is done through the platform, which can see both sides and routes the messages between them.

During the communication, experience managers and environments exchange messages to update each other on the events happening in the experience and how they should respond to them to achieve the manager’s desired outcome. The environment sends messages to the experience manager and these messages contain an abstract representation of the events happening in the experience. The experience manager receives these messages and uses them to decide which action to let the environment perform and sends it to the environment. The environment receives this message and executes the action if it is possible to do so. Figure 5.1 shows a high-level overview of how EM-Glue is designed and an example of communication between an EM and an environment. EM-Glue is open source and available on GitHub [75]. This makes it easier for developers and researchers to access the platform and contribute to its development.

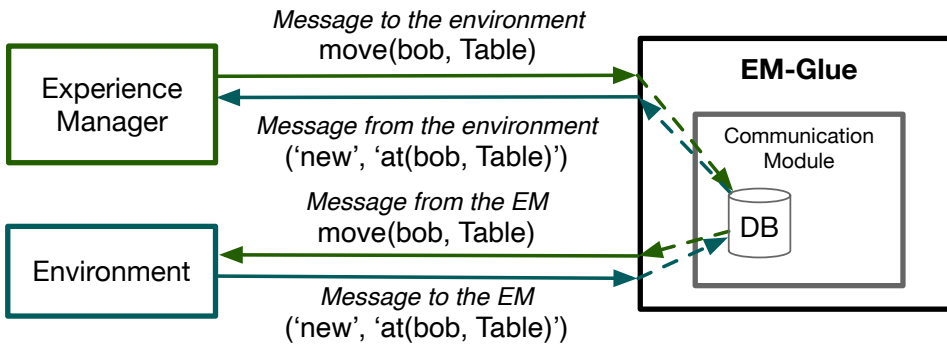


Figure 5.1: An overview of our EM-Glue’s design. The Experience Manager and the Environment are external modules. The Environment sends messages to update the EM with what is happening in the environment and the EM sends the action that it wants to apply.

Figure 5.1 shows the design of EM-Glue, which considers three kinds of components: the platform itself, experience manager, and environment. The external modules that are connected to the platform are experience managers and environments.

5.1.1 Environment

An environment’s three objectives are to handle the experience’s audio/visual components, manage user interaction, and monitor the experience activity using a high-level state representation. The graphical component of the experience is what the user sees and can vary in complexity, ranging from a text-based interface to a 3D environment. The environment is responsible for handling user interaction through an interface connected to the graphical component. A keyboard, mouse, or other input device can be used for this interaction. In addition, the environment

keeps track of what is happening in the world where the user is interacting using a high level state representation. This state representation is important because it provides information about the events happening in the experience and how the abstract state of the environment changes in response to these events. This representation is abstract because of the constraints defined in Section 4.9. The messages exchanged between the environment and the experience manager are based on this state representation and provide information about the user experience and the environment's state to the manager.

5.1.2 Experience Manager

An experience manager is responsible for handling the experience at a more abstract level compared to what the environment does. In the previous chapter's literature review, it was found that experience managers tend to focus on an abstract view of the environment. This is because attempting to handle all the details of the environment can be overwhelming, and they prefer to keep things within a manageable scope. By using a high-level state representation, the manager can easily understand what is happening in the environment without having to deal with the intricacies of how it is implemented. Let us imagine a situation where the experience manager wants to move an NPC from one room of the environment to another room. If the manager needed to handle low level interaction, it would need to issue all the commands related to having the NPC to move. This would involve telling the environment to gain control of the NPC, calculating the path that the NPC needs to take to reach the other room, applying a movement to the NPC in the direction of the path, controlling the animations of the NPC, and so on. With a higher level of abstraction, the experience manager would only need to issue the command to move the NPC to the other room. The environment would then execute the needed commands and move the NPC. So, using an abstract representation simplifies the manager task of defining the overall flow and structure of the experience. Additionally, it can handle tasks like tracking the progress of the player and adapting the experience based on the player's actions.

5.1.3 EM-Glue

EM-Glue is the middle layer that connects the external modules and facilitates communication between them. It includes the software that transfers messages from one end of each connection to the other and maintains a protocol that allows a structured and ordered transmission. To transfer messages between experience managers and environments, there is the need to use a technology that allows the communication between different systems regardless of the technology that they are developed with. Another important aspect to consider is that this technology should be able to handle the amount of data that is exchanged between the external modules, in a fast and reliable way. As I discussed in Section 4.9, a missed message during the communication between the external modules can cause the experience to break and the user to lose the flow of the experience. A more in-depth description of the design and development of the communication module can be found in Section 5.2.

Meanwhile, a standardized protocol for the communication is needed to allow the external modules to understand each other, and to be able to exchange messages in a way that regulates the kind of information that is exchanged and when it needs to be exchanged. In fact, during the set up of the experience, the external components need to exchange the necessary data to initialize the environment and the experience manager. Once this information is shared, then they can start exchanging messages to update each other. The environment updates the manager with changes to the abstract state indicating what is happening in the environment. The manager sends to the environment the instructions that the environment needs to execute. While managing this conversation, the communication module also stores a history of all of the messages that are exchanged between the external modules. This history can be useful for comparing different EMs, because one can use it to reconstruct the sequence of events that occurred in any player's experience. For example, if a player killed an important character for a story, from the history of messages one could check how the EM responded to that event. A detailed description of the communication protocol can be found in Section 5.3.

5.2 Communication Module Infrastructure

The first step in developing the software structure of the communication module was to survey the space of possible technologies for communication. During this survey, my focus was on prioritizing three aspects: easy inclusion of new external modules, the ability to handle the amount of data exchanged between the external modules, and that the technology would need to be easy to understand and use.

5.2.1 Communication Technology

I first analyzed the communication method used by *Camelot* [117], which is the “standard” input/output. Analysing this as first method was promising because *Camelot* is a system that was designed to separate the experience manager from the environment, and I thought that it would be a good idea to use the same method that they used. However, I found that this technique does not work well with concurrent requests. For example, if a program needs to write a message at the same time that it is waiting for a message to arrive, it becomes stuck in a deadlock situation. This is a problem I faced during the development of the *Camelot Wrapper*, as I discuss in Section 6.1.

I also analyzed sockets, the fundamental technology that enables network communication. Sockets have been used in the past in similar systems, as we have seen in Section 3.2. The benefits of using sockets are multiple: portability across different operating systems, low overhead that makes them suitable for high-performance applications, support for full-duplex communication so that two applications can send and receive data simultaneously, and interoperability since the communication between processes can be written in different programming languages. However, sockets also have multiple disadvantages to consider: complexity since implementing sockets requires a good understanding of network programming, reliability since communication can suffer from dropped packets and data corruption and requires

additional mechanisms to ensure reliability, and maintenance since bugs in the communication can be hard to diagnose and fix. Another important downside of using sockets is that the learning curve is steep. One requirement was to use a technology that is easy to understand and use so that others can develop external modules using our platform with minimal added complexity to their system. Given these considerations, I decided not to use sockets directly, but a modern evolution of sockets that have become commonly used: RESTful APIs [123].

REST (Representational State Transfer) is a set of architectural principles that define how data should be transferred over the internet [30]. In REST, a resource is a piece of information that can be accessed and manipulated using standard HTTP methods, such as GET, POST, PUT, and DELETE. A resource is identified by a unique identifier, usually a URI (Uniform Resource Identifier), and can be considered a single endpoint representing the resource. RESTful APIs are a type of web service that adhere to the REST architectural style, using HTTP methods to perform operations on resources. One of the most used Python libraries to implement RESTful APIs is FastAPI [96]. It is straightforward to implement in a codebase and allows a high-performance exchange of messages. Communicating with a web API only requires sending an HTTP request, which can typically be done with a few lines of code. There are two disadvantages of using RESTful APIs. First, RESTful APIs can be inefficient in some cases because they require multiple HTTP requests to complete a single operation, which can add unnecessary overhead and affect performance. Connected to this disadvantage, the second drawback is that the server cannot directly communicate with the client, which means that the client needs to make a request to the server and then wait for a response. However, these disadvantages can be eased with simple solutions that I will discuss in Section 5.2.2.

5.2.2 Message Exchange Using RESTful APIs

Figure 5.2 shows the exchange of messages used as an example in Figure 5.1, but converted to use the RESTful APIs technology. EM-Glue hosts a server where the process of the API operates, and it is constantly waiting for requests to activate the available functionalities. When one of the external modules (e.g., an EM) needs to send a message to another external module (e.g., an environment), it makes an HTTP POST request to the platform using a URL dedicated to receiving messages from that module (e.g., `/add_em_message`). The module attaches the message's content to the HTTP request's body. The platform receives the request on the specific URL, ensures that the content of the message is formatted correctly, and stores the message in a database. Then, it replies to the request with a success message (HTTP code 200). For an external receiving module to read an incoming message, it must keep polling the URL for incoming messages with GET requests (e.g., `/get_messages_for_env`). When an incoming message is available, the platform sends it to respond to the most recent requests. The same process applies to the other direction of communication, where the environment sends messages to the EM. The environment sends a POST request to the platform using a URL dedicated to receiving messages from the environment (e.g., `/add_env_message`)

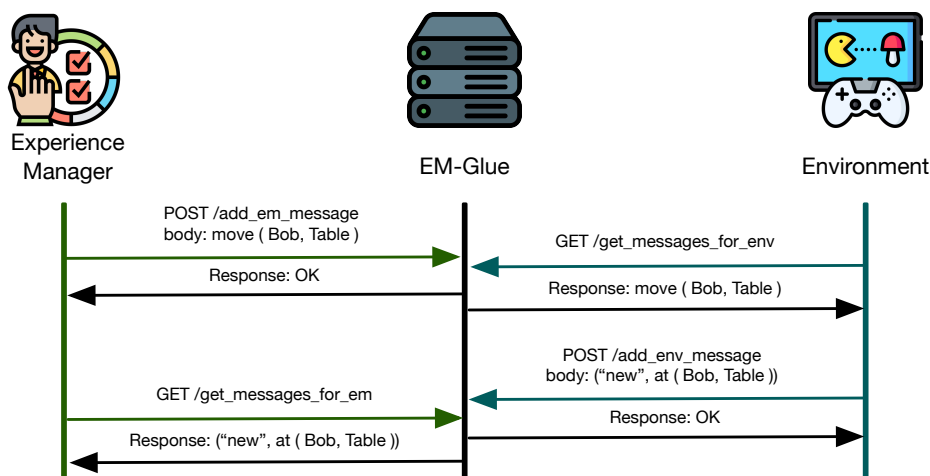


Figure 5.2: An example of a communication between an experience manager and an environment via EM-Glue using the RESTful APIs. This example uses the same messages from Figure 5.1. The icons used are licensed from flaticon.com.

with the attached content of the message. Upon receiving a request at a designated URL, EM-Glue verifies the formatting of the message’s contents and then saves it within a database. Since the experience manager is polling the URL (e.g., `/get_messages_for_em`) for incoming messages, it will receive the message as a response to the most recent GET request.

Errors have a different path than regular messages, because if an error occurs in the environment, it might be of such gravity that the experience breaks. If an error occurs, the EM might need to take immediate action to solve it. For this reason, the platform provides a mechanism to send errors to the EM as soon as they occur. The principle used is the same as the one used for regular messages, but the URL used differs. The environment sends a POST request to the platform using a URL dedicated to receiving errors from the environment (e.g., `/add_error_message`) with the content of the error message. Upon receiving a request at a designated URL, EM-Glue verifies the formatting of the message’s contents and then saves it within a database. Since the experience manager is polling the URL (e.g., `/get_error_messages`) for incoming errors, it will receive the error as a response to the most recent GET request. In the case of Camelot and the Camelot wrapper (which I discuss in Chapter 6.1), it attempts to avoid errors by validating every message before sending it to the environment.

One drawback of RESTful APIs is that the server cannot directly send a message to a client; it can only respond to a request that comes from a client. We have seen this limitation in Figure 5.2, where EM-Glue cannot send a message to the external modules without them first sending a request to the platform. The communication described previously compensated for this limitation by having the external modules poll the platform for incoming messages. Resolving this problem is relatively straightforward, and there are two ways that it can be solved. One

way is to develop a service in the client that periodically sends GET requests to the URL for incoming messages (e.g., every 0.2 seconds). This is the solution that I have implemented in the current design of the platform, since it requires minimum work, and the complexity added to the code is relatively low (a GET request to EM-Glue). Another possible solution would be to allow the client to send a GET request and then have it wait until there is an update available. In the current platform version, I decided not to support this solution because it would involve using multi-threading in the client to obtain a smooth operation. In fact, if the platform waits to give an answer to the GET request, the client will be blocked until the answer is available. The client might want to do other things while waiting for the answer, and multi-threading would allow it to do so. The goal is to keep the engineering of new managers as simple as possible, and multi-threading would add complexity to that work. In a later version of the platform, I plan to support both options so that developers can choose to use what they prefer.

5.3 Communication Protocol

The second step towards effective communication between the experience manager and the environment is to define a protocol that allows the two modules to communicate in a structured way. A communication protocol is essential for facilitating seamless communication between external modules. This protocol ensures that experience managers and environments during the communication process can understand each other, set up an initial experience exchanging all necessary data, and exchange messages in a controlled and regulated manner. The communication protocol provides a framework for the modules to follow, regulating the turns in which they can communicate, ensuring that the messages are transmitted in an orderly and organized manner and in a common language that both external modules can understand. My discussion of the communication protocol is divided into three parts. First, in Section 5.3.1, I describe the language used during the communication to exchange information about the experience and represent the abstract state of the environment. Then, in Section 5.3.2, I describe the handshake protocol used to set up the initial experience. Finally, in Section 5.3.3, I describe the communication protocol used to exchange messages during the experience.

5.3.1 Communication Language

To achieve an effective communication, there is the need to use a common communication language. For instance, if two people need to interact where one speaks Italian, and the other speaks German, they must determine some common language to do so. They might discover at that point that they are both capable of understanding and speaking English, so their common language to interact would become English. Similarly, a specific language is needed to allow experience managers to communicate with environments. This language should effectively communicate a high-level overview of what is happening in the environment, so the EM can interpret a message and understand the current state to act upon it. In addition, the EM also needs to have a set of possible actions to choose from, described with

conditions that need to be met for correct execution and the effects each action has on the state once executed. The literature offers many different languages that can be used to represent these aspects of the experience (as described in Section 3.3), each solving problems of representation that other languages have.

For EM-Glue, after careful selection, I decided to use the Planning Domain Definition Language (PDDL) [67] as the language to represent the environment’s state and the actions that the EM can perform. PDDL allows for the specification of the initial state, goal state, and the set of operators that can be applied to transition from one state to another. The PDDL specification consists of a domain file and a problem file. The *PDDL domain* file defines the essential elements of the problem domain and the actions that are available to the agent. It specifies the types of objects in the world, the predicates that describe the state of the world, and the operators (or actions) that the agent can perform. These operators are defined in preconditions (conditions that must be true for the action to be executed) and effects (changes to the world state that result from executing the action). The domain file serves as a basis for all problems in the same domain.

The *PDDL problem* file, on the other hand, specifies a particular instance of the problem to be solved. It defines the initial state of the world, the goal state, and the objects and predicates relevant to the particular problem instance. The problem file uses the definitions from the domain file and specifies the details of a specific problem.

As briefly described in Section 3.3, PDDL was designed to describe problems in automated planning, which is notably different from this context. In automated planning, the objective is to find a sequence of actions that will lead from the initial state to the goal state. In this context, PDDL is used to describe the current state of the environment, the actions that the EM can perform, and the effects of each action on the state. The goal state is not specified, and the EM is not required to find a sequence of actions to reach it. In fact, in the current version of EM-Glue the PDDL goal is not used. Nevertheless, PDDL is well-known across the field and has been used with EMs in the past [90, 135, 142, 25, 91]. Moreover, learning PDDL is supported by many online resources, which helps intending to create a platform that is easy to use and accessible to the community. In Section 7.3, I discuss the limitations of using PDDL in this setting. However, the benefits of using PDDL outweigh the drawbacks, especially since the initial design of EM-Glue is a proof of concept to demonstrate that it is possible to achieve interchangeability between EMs and environments. Moreover, I made the design of the platform’s software and communication protocol independent of the language that the EMs and environments use, and it would be straightforward to exchange PDDL with another language.

5.3.2 Handshake protocol

EM-Glue’s handshake protocol is a standardized set of messages that connect an EM and an environment through the platform. It was inspired by the TCP/IP three-way handshake protocol [29], a TCP/IP network process to establish a communication channel between two devices. The three-way handshake consists of

three steps. First, the initiating device (typically a client) sends a packet with the SYN (synchronize) flag set to the target device (typically a server), requesting the establishment of a communication session. Second, the target device receives the SYN packet and sends back a packet with both the SYN and ACK (acknowledge) flags set, indicating that it is ready to communicate and acknowledging receipt of the SYN packet. Finally, the initiating device receives the SYN-ACK packet and sends back a packet with the ACK flag set, completing the three-way handshake and confirming that both devices are ready to start communicating. The three-way handshake protocol is used to establish a reliable connection between two devices, and it was a good starting point for the design of EM-Glue’s handshake protocol. However, EM-Glue’s handshake protocol is more complex than the TCP/IP three-way handshake protocol for two reasons. First, it involves three devices (experience manager, EM-Glue, and environment) instead of two. Second, it is not limited to establishing a connection but also exchanging information to set up the experience. Figure 5.3 shows a diagram of the steps needed to initialize the communication between an EM and an environment successfully.

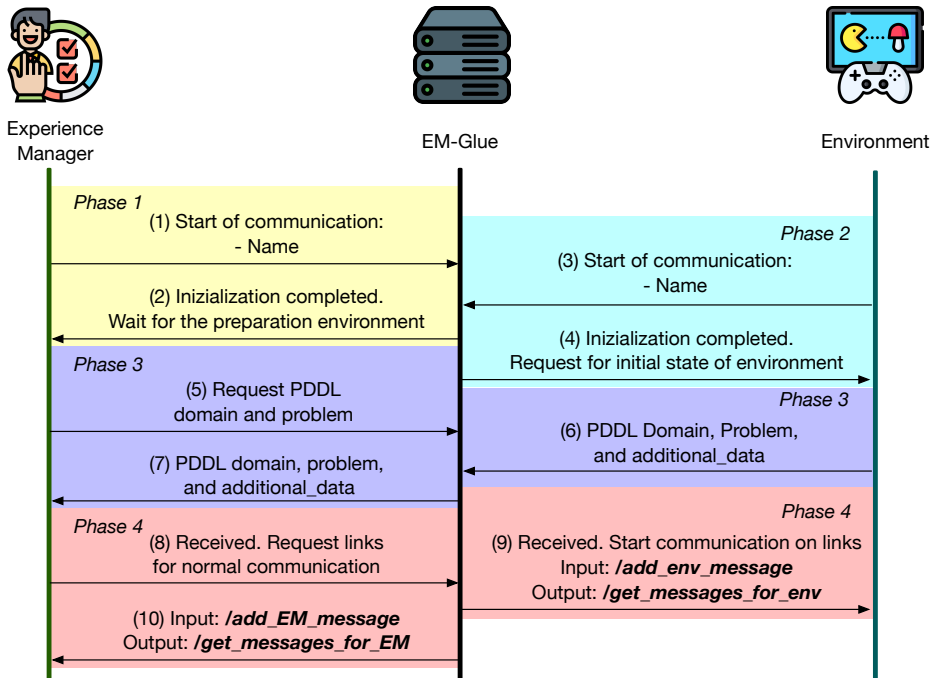


Figure 5.3: A summary of EM-Glue’s handshake protocol steps. (#) indicates the order of the protocol steps, each involving one message. Each coloured area represents a phase of the protocol.

This handshake protocol involves the EM, EM-Glue, and environment simultaneously. It is divided into four phases, and each is used to handle a different part of the initial communication. I describe this protocol assuming that EM-Glue is up

and running and that the EM and environment are ready to be initialized. Once the platform is ready, it launches the EM process automatically. EM-Glue knows which EM and environment to use based on a JSON file [87] that holds the paths for their executables. It starts with *phase 1* where the EM creates a request to the initialization link (*/inizzialization_em*) with the name of the EM as a parameter (step 1). After receiving the initial message from the EM, EM-Glue adds the received data to the database (DB). EM-Glue reads the message, replies to this request with the confirmation that the initialization is completed and is waiting for the environment to be ready (step 2). In the meantime, EM-Glue starts the environment and waits for a request to arrive from the environment. Once the environment started, it begins *phase 2* by sending a new request on the environment's initialization link (*/inizzialization_env*) with the parameter representing its name (step 3). EM-Glue reads this message and replies to the environment with a message saying that the initialization is completed and that it is requesting the domain and initial state of the environment (step 4). Phases 1 and 2 are the steps that ensure that the EM and environment are up and running, they can read and write messages, and they are ready to start the communication with the information needed to initialize the experience.

Once phase 2 finishes, the EM prepares a request on the link (*/inizzialization_em*) to begin *phase 3* and sends a message to request the PDDL domain and problem (step 5). In the meantime, the environment prepares a request to send on the URL (*/inizzialization_env*) to reply to EM-Glue's request for the initial state of the environment by providing the PDDL domain and problem that contain such information (step 6). Once the platform receives the message from the environment, it keeps the environment on hold (by waiting to send back the reply) and replies to the request of the EM with the PDDL domain and problem received from the environment (step 7). It is important to keep the environment waiting until the platform confirms the successful transmission of the PDDL data because initialization should halt if there is an error. When the EM receives the data, it starts *phase 4* by creating a new request on the URL (*/inizzialization_em*) to confirm that it correctly received the PDDL data and asks for the URLs for regular communication (step 8). The platform then confirms the correct receipt of the data to the environment and sends the URLs for regular communication (step 9). Finally, the platform replies to the EM by sending the regular communication URLs to it (step 10).

I designed this handshake protocol to account for the drawback of RESTful APIs that I discussed earlier in Section 5.2. Specifically, the external modules must always make the first request to allow EM-Glue to respond with the needed data. I also chose to let the environment wait for the reply to the request between steps 6 and 9, even if, in Section 5.2, I mentioned that I would not support this type of behaviour. This was an informed decision with two motivations behind it. First, in general, I did not support the waiting because it requires multi-threading in the client to handle situations where the server is not ready to reply and the client needs to do computation in the meantime. However, this is not the case because, between steps 6 and 9, the environment is waiting for a confirmation from the manager saying the data was correctly received. This step is crucial to ensure

that the data exchanged between the EM and the environment is correct. This implies that the environment must wait for the reply to the request to start doing any other preparation work, thus not requiring any multi-threading to handle the wait since the environment needs to wait in any case. Secondly, I did not want to add complexity to the handshake protocol by adding two new steps, even if the added complexity would be minor. The two steps would not contain any new information since, in the first message, the platform would need to reply with an acknowledgment that it received the message, and it must wait for the reply of the EM. Then, in the second message, the environment would need to make a meaningless request to the platform just to be able to receive the reply from EM-Glue.

In addition, another aspect that I considered during the design process of this handshake protocol is that I wanted a design that could be easily adapted to languages other than PDDL in the future. This allows the platform to be updated later with a different language for representing experience management tasks, which helps to future-proof the protocol. This consideration affected my design decisions by not including any reference to the PDDL language in the protocol. In fact, the PDDL specification is exchanged in *phase 4* between the EM and the environment, but the platform does not check which language is exchanged and it could be easily replaced with a different language in the future.

Another aspect I considered during the protocol development is that, in practice, the order of phases 1 and 2 could be interchangeable. These initial stages confirm the readiness of both the EM and the environment, and ensure they can exchange messages and have the necessary information to begin the communication and the experience. However, the EM should go first to allow, in the future, the EM or EM-Glue to choose which environment to use when more than one environment is available.

5.3.3 Regular Communication

The last step towards achieving successful communication between the EM and the environment is establishing a protocol that controls the regular communication while a player's experience is running. This protocol is used to exchange messages between the EM and the environment during the experience. Regular communication does not work in turns like the handshake protocol. Instead, the EM and the environment can send messages to each other at any time. During regular communication between an EM and an environment, two types of messages are exchanged that describe the different kinds of information that the EM and the environment need to exchange.

The first type of message travels from the experience manager to the environment. Each of these messages represents an action the experience manager wants to perform in the environment. This message is composed of the name of the action and the entities that are involved in the action. This information must correspond to the description of an action from the domain. For example, in Listing 5.1, `openfurniture` is the name of the action, and `bob` and `alchemyshop.chest` are the entities involved in the action.

```
1  openfurniture(bob, alchemyshop.chest)
```

Listing 5.1: A call of an action using PDDL.

Between the actions that the experience manager can call, there are some that have special abilities of influencing the environment. There are two types of these actions. First, the actions whose name starts with `instantiate_` that are used to instantiate a new object in the environment. Secondly, the actions whose name starts with `start_` which are used to start the execution of either a conversation (e.g., `start_conversation`) or a scene (e.g., `start_scene`). Both the `instantiate` and `start` actions are defined as any other action in the PDDL domain, but they have a special meaning for both the environment and the manager.

The second type of message travels from an environment to an experience manager. This message describes all the updates to the world state of the environment. It is composed of a tuple of two elements. The first element is a string that indicates how to handle the second element of the tuple, and the second element can be a relation, a new entity, or an update to the player model. An example of such a tuple is shown in Listing 5.2.

```
1  ('new', at(bob, alchemyshop.chest)')
```

Listing 5.2: A message that communicates an update of the environment state.

The first element can have one of four values: `new` when the second element is a new relation that the world state did not have before, `changed_value` when the second element is a pre-existing relation whose value is changing (e.g., from true to false or vice-versa)¹, `update_player_model` when there is the need to update the player model after a choice that the player made in the environment, and `new_entity`. The `new_entity` value is used when the environment responds to an EM action that instantiates a new entity in the game; the second element is the entity's name. This message tells the EM that the `new_entity` action was executed successfully. Player actions are not directly reported; the environment only communicates the relations that change while playing the game. However, in the future, I plan to add a message type to report player behaviour explicitly, as I discuss in Section 7.4.

5.4 Implementation

In this section, I will present the software architecture of EM-Glue with the implementation details of the different components. Software architecture refers to the high-level design of a software system that defines its components, their relationships, and their interactions. It plays a crucial role in the development process by providing a blueprint for the system's structure and behaviour.

Figure 5.4 shows the architecture diagram of EM-Glue with the scripts and libraries I used to develop the platform. The software architecture is divided into three main components: the SQLite database, the API server, and EM-Glue manager script. I developed EM-Glue using Python 3.9.1 and libraries for the database

¹A relation that is not explicitly recorded is assumed to be false.

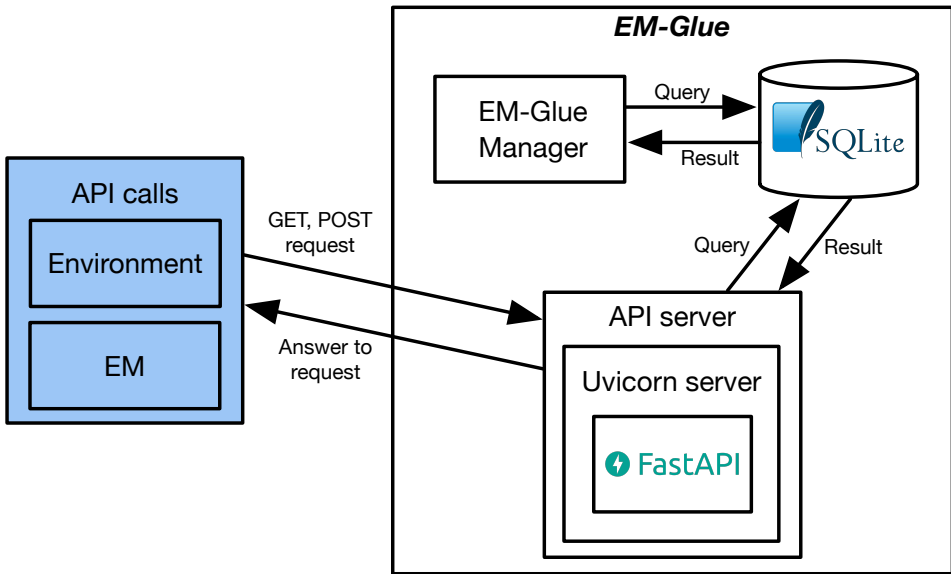


Figure 5.4: Architecture diagram of the EM-Glue platform.

management and API server that I will describe in the following sections. All the code is open source and available on GitHub [75].

5.4.1 SQLite Database

In EM-Glue, a database is used to store the messages that are shared between the EM and the environment. There are many different options when it comes to choosing a database for a project. In this case, I needed a database that could be embedded in the platform, that could be used by both the API server and the EM-Glue manager script, and that is lightweight and easy to use. The database will handle a relatively small amount of data, with a maximum of three connections open simultaneously. These requirements resulted in my choosing SQLite as the database for EM-Glue. A SQLite database is a file that contains a structured set of data that is stored and managed by the SQLite library [43]. It is a lightweight, serverless, and self-contained database that can be embedded in applications or used as a standalone database.

To implement a SQLite database in Python, there are two options: the `sqlite3` module and the SQLAlchemy library. `sqlite3` is the native library where I would have needed to manually handle the database connection and the queries. This would have required much boilerplate code that would have made the code harder to read and maintain. Instead, I chose to use SQLAlchemy, an Object-Relational Mapper (ORM) that provides a high-level interface to the SQLite database [13]. ORMs are libraries that map the database tables to Python classes and provide an interface to query the database using Python objects. This allowed me to write the database queries in a more readable way and focus on the application's logic

instead of the database connection and queries.

All the scripts that handle the database and queries are stored in the `sql_app` folder of the EM-Glue repository. The first script is the `database.py` file, which contains the database declaration and path to the database file. The second script is the `models.py` file, which contains the database models mapped to the database tables. Figure 5.5 shows an image representing the database schema. The database is

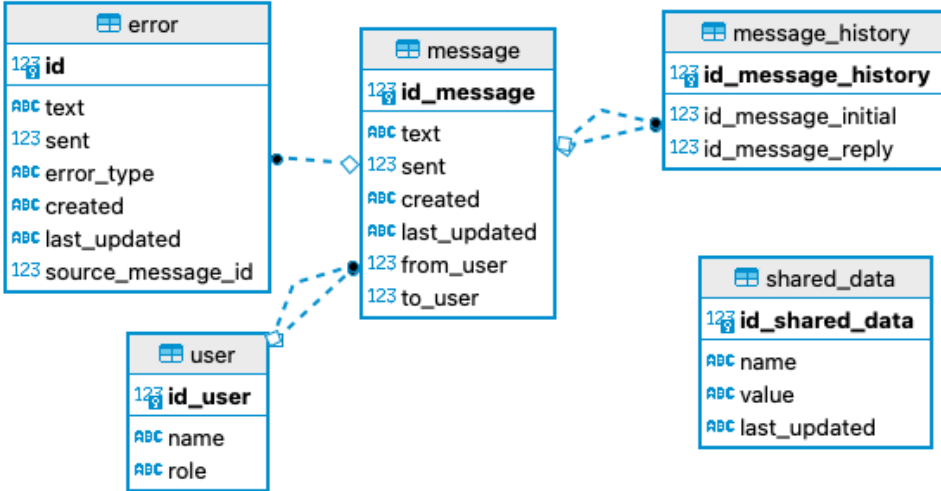


Figure 5.5: Database schema.

composed of five tables: `User`, `Message`, `Error`, `MessageHistory`, and `SharedData`. The `User` table contains the information about the users that are connected to the API server. It has a one-to-many relationship with the `Message` table. The `Message` table contains the messages sent by the EM and the environment. It has a one-to-many relationship with the `MessageHistory` table, which stores the history of the messages. Then, the `Error` table contains the errors generated by the experience manager or the environment. Finally, the `SharedData` table contains the data shared between the EM and the environment (e.g., the current phase of the handshake protocol). Each table is represented as a Python class in the `models.py` file. Listing 5.3 shows an example of the `Message` class. As we can see, each of the class's attributes is mapped to a column in the database table. Also, the python class defines the relationships between the tables using the `relationship` function.

```

1 class Message(Base):
2     __tablename__ = "message"
3     id_message = Column(Integer, primary_key=True, index=True)
4     text = Column(String)
5     sent = Column(Boolean, default=False)
6     created = Column(DateTime, default = datetime.datetime.now)
7     last_updated = Column(DateTime, default = datetime.datetime.now, onupdate=datetime.datetime.now)
8     from_user = Column(Integer, ForeignKey('user.id_user'))
9     to_user = Column(Integer, ForeignKey('user.id_user'))
10    from_user_rel = relationship("User", foreign_keys=[from_user], uselist=False)
11    to_user_rel = relationship("User", foreign_keys=[to_user], uselist=False)

```

Listing 5.3: Class `Message` of the `models.py` script. I used classes like this to create the tables of the SQLite database.

The third script of the `sql_app` folder is the `schemas.py` file, which contains the schemas for the database models. The schemas represent a valid data shape that can validate the data sent to the API server. Each validation schema is made using the Pydantic library [19]. Pydantic models define the expected structure and types of data input into the API calls and the database queries. The Pydantic library uses the model to validate and parse data, ensuring that the input data is in the expected format and contains the expected data types. For example, the `MessageCreate` class in Listing 5.4 defines the expected data structure to be sent to the API call to create a new message. This data structure contains the system (platform, experience manager, or environment) that sent the message and the system that will need to receive the message as required attributes and, as an optional attribute, the message id that caused this message to be sent. It extends the `MessageBase` class, which defines that a message must always have a text attribute.

```
1 class MessageBase(BaseModel):
2     text: str
3     class MessageCreate(MessageBase):
4         from_user: int
5         to_user: int
6         old_message_id: Optional[int] = None
```

Listing 5.4: Classes `MessageBase` and `MessageCreate` of the `schemas.py` script. These are used to define the valid data shape accepted by the API server.

The fourth script of the `sql_app` folder is the `crud.py` file, which contains the functions used to query the database. CRUD comes from the first letter of each of the operations that the queries in the class can do: Create, Read, Update, and Delete. All the functions in this class have a rule that needs to be followed when named to keep them consistent and easy to understand. The first part of the function name is the name of the operation that the function performs, the second part is the name of the table that the function is querying, and the last part is the parameters that are needed. As we can see from Listing 5.5, the function `get_user_with_ID` is used to query the `User` table, and it needs the database connection and the id of the user that we want to query. Using SQLAlchemy, the query is written by applying functions to the database object. In this example, the function `query` is used to start a query operation on the table that is given as a parameter (like the `FROM` statement in SQL), the function `filter` is used to filter the results of the query using the parameters given (like the `WHERE` statement in SQL), and the function `first` is used to return only the first result of the query (like the `LIMIT 1` statement in SQL).

```
1 def get_user_with_ID(db: Session, id_user: int):
2     return db.query(models.User).filter(models.User.id_user == id_user).first()
```

Listing 5.5: Query read example of the `crud.py` script.

Another example is the function `create_user` in Listing 5.6, which is used to create a new user in the database. This function needs the database connection and data to create the user. To create a new user in the database, the function creates a new instance of the `User` class and passes the data given as a parameter to the class’s constructor. Then, the function adds the new user to the database and commits the changes. Finally, the function refreshes the new user’s data to get the parameters that are automatically generated (e.g., the primary key) and returns it.

```
1 def create_user(db: Session, item: schemas.UserCreate):
2     if item.role not in ["EM", "ENV", "PLATFORM"]:
3         raise InvalidRoleException("Invalid role, must be EM or ENV")
4     db_item = models.User(**item.dict())
5     db.add(db_item)
6     db.commit()
7     db.refresh(db_item)
8     return db_item
```

Listing 5.6: Query create example of the `crud.py` script.

The last script of the `sql_app` folder is the `exceptions.py` file, which contains the custom exceptions used in the API. The custom exceptions are used to return a more specific error message to the developer that is using the API. For example, the `InvalidRoleException` in Listing 5.6 returns an error message when the role given as a parameter is invalid.

5.4.2 API Server

The API server contains all the software needed to run the APIs for communication with the external systems (experience managers and environments). As we can see from the EM-Glue architecture in Figure 5.4, the API server comprises multiple services that run within it. The first service is the Uvicorn server [138]. Uvicorn is a fast ASGI (Asynchronous Server Gateway Interface) server designed to run Python web applications. ASGI is a standard interface between web servers and Python frameworks, allowing asynchronous communication between them. This means the server can handle multiple requests simultaneously without waiting for a response before processing the subsequent request. Uvicorn is built on top of a high-performance networking library (called “uvloop”) and supports the HTTP protocol.

The Uvicorn server is used to host the FastAPI application [96] FastAPI is a modern, lightweight, high-performance web framework for building APIs with Python 3.7+. FastAPI extensively uses Python type annotations to provide automatic data validation and self-documenting code. It also generates OpenAPI documentation automatically, making it easy to keep API documentation up-to-date and accessible. All these characteristics made me choose FastAPI as the framework to build the API server.

In FastAPI, to create an API endpoint, we need to create a function decorated with the URL we want to use. In Python, a decorator is a function that takes another function as input and extends its behaviour without changing its source code. The syntax of a decorator is the `@` symbol followed by the name of the

decorator function, and it is placed before the definition of the function that will be decorated. For example, in Listing 5.7, the function `is_online` is decorated with the `@app.head("/")`. This decorator means that the function `is_online` of the FastAPI application `app` will be called when a `HEAD` request is made to the main URL (e.g., `localhost/`) of the API server.

```
1 @app.head("/")
2 def is_online():
    return "online"
```

Listing 5.7: Example of an API endpoint using FastAPI.

The FastAPI application that I developed to run the communication in EM-Glue comprises three types of API calls: general, handshake protocol, and normal communication.

5.4.2.1 General API Calls

The general API calls perform general operations on the API server. Three general API calls are always available regardless of the state of the experience.

- `HEAD` request on the main URL. This API call is used to check if the API server is running.
- `GET` request on `/protocol_phase`. This API call is used to get the current phase of the handshake protocol. The result of a call on this API endpoint is either: `PHASE_1`, `PHASE_2`, `PHASE_3`, `PHASE_4`, or `DONE`.
- `GET` request on `/get_protocol_messages`. This API call is used to get the messages sent during the handshake protocol. This allows experience managers and environments to get the expected messages during the handshake protocol.

The messages sent during the handshake protocol are in a JSON file loaded when the API server starts. This JSON file is called `messages.json` and is located in the `json_data` folder. Listing 5.8 shows the content of this JSON file. It is a dictionary where the keys are the phases of the handshake protocol, and the values are the messages that are sent during that phase of the handshake protocol (as they are in Figure 5.3)

```
1 {
2   "PHASE_1" :
3     {
4       "message_1" : "start of communication. name:",
5       "message_2" : "inicialization completed. wait preparation environment."
6     },
7   "PHASE_2" :
8     {
9       "message_3" : "start of communication. name:",
10      "message_4" : "inicialization completed. Request for initial state of environment."
11     },
12  "PHASE_3" :
13    {
14      "message_5" : "request domain and problem",
15      "message_6" : "domain and problem",
16      "message_7" : "domain and problem"
```



```

18     };
19     "PHASE_4" :
20     {
21         "message_8" : "received domain and problem. request links for communication",
22         "message_9" : "received. start communication on links",
23         "message_10" : "links"
24     }
25 }

```

Listing 5.8: Messages that are exchanged during the handshake protocol in JSON format.

5.4.2.2 Handshake Protocol API Calls

The second type of API calls are the handshake protocol API calls. These API calls are used during the execution of the handshake protocol described in Section 5.3.2. Two API calls are used during the handshake protocol.

- GET request on `/inizialization_em`. The experience manager uses this API call to exchange all the necessary messages for the handshake protocol.
- GET request on `/inizialization_env`. The environment uses this API call to exchange all the messages necessary for the handshake protocol.

Listing 5.9 shows the code of the `/inizialization_em` API endpoint. I included only one of the two API endpoints because they are very similar.

```

1  @app.get("/inizialization_em", response_model=schemas.Inizialization)
2  def inizialization_em(text : str, db: Session = Depends(get_db)):
3      protocol_phase = crud.get_shared_data_with_name(db=db, name="protocol_phase")
4      return_message = ""
5      if protocol_phase.value == "PHASE_1":
6          return_message = communication_protocol_phases.phase1(db=db, text=text, ←
7          ←communication_phase_messages=communication_phase_messages)
8          elif protocol_phase.value == "PHASE_2":
9              raise HTTPException(status_code=404)
10         elif protocol_phase.value == "PHASE_3":
11             return_message = communication_protocol_phases.phase3_EM(db=db, text=text, ←
12             ←communication_phase_messages=communication_phase_messages)
13             elif protocol_phase.value == "PHASE_4":
14                 return_message = communication_protocol_phases.phase4_EM(db=db, text=text, ←
15                 ←communication_phase_messages=communication_phase_messages)
16             else:
17                 raise HTTPException(status_code=404)
18             return return_message

```

Listing 5.9: The script that governs the handshake protocol for the experience manager.

When a GET request is made to the `/inizialization_em` API endpoint, the function `inizialization_em` is called. This request must contain a parameter called `text` that contains the message that needs to be sent during the corresponding phase of the handshake protocol. The function first gets the current phase of the handshake protocol from the database. Then, based on the current phase, it calls the function that is responsible for handling the corresponding phase. If the communication protocol is not being executed, then the function returns a 404 error.

All the functions responsible for handling the messages during the various phases of the handshake protocol are located in the file `communication_protocol_phases.py`. The functions for `PHASE_1` and `PHASE_2` are similar and change only in the part used to manage the role of the experience manager or the environment. Let us inspect the function responsible for handling `PHASE_1` in Listing 5.10.

```

1 def phase1(db: Session, text: str, communication_phase_messages: dict):
2     if text.startswith(communication_phase_messages["PHASE_1"]["message_1"]):
3         name = text[len(communication_phase_messages["PHASE_1"]["message_1"]):]
4     else:
5         raise HTTPException(status_code=400)
6     try:
7         res = crud.create_user(db=db, item=schemas.UserCreate(name=name, role="EM"))
8     except InvalidRoleException as e:
9         raise HTTPException(status_code=400, detail= str(e))
10    platform = crud.get_user_with_role(db, role="PLATFORM")
11    try:
12        crud.create_message(db=db, item=schemas.MessageCreate(text=text, from_user=res.id_user, ↵
13        ↵to_user=platform.id_user))
14    except (InvalidMessageIDException, InvalidUserException) as e:
15        raise HTTPException(status_code=400, detail= str(e))
16    return _wait_and_return_message_for("EM", db)

```

Listing 5.10: Function that handles the communication during `PHASE_1` of the handshake protocol.

First, it checks if the message that is received is the expected message for the current phase of the handshake protocol, and it extracts the name of the EM from the message. It creates a user in the database with the role of EM with the previously extracted name. Then, it creates a message that contains the received message, the ID of the system that sent the message (EM), and the system that needs to receive the message (EM-Glue). It then inserts this message into the database. Finally, it calls the private function `_wait_and_return_message_for` that waits for the message that needs to be received by the EM to end the current phase and returns it.

In the case of `PHASE_3` and `PHASE_4`, the handling of the messages differs based on the system's role that is sending the message. As we can see from Figure 5.3, the experience manager executes these two phases using two API calls while the environment uses only one call to the API. Listing 5.11 shows the code of the function that is responsible for handling the message that is received from the environment during `PHASE_3` and sends the message to the environment for `PHASE_4`.

```

1 def phase3_4_ENV(db: Session, item: schemas.Inizialization, communication_phase_messages: dict):
2     if not item.text.lower().startswith(communication_phase_messages["PHASE_3"]["message_6"]):
3         raise HTTPException(status_code=400)
4     platform = crud.get_user_with_role(db, role="PLATFORM")
5     env_user_id = crud.get_user_with_role(db, role="ENV").id_user
6     message_text = item.text + "###" + item.domain + "###" + item.problem + "###"
7     if item.additional_data is not None:
8         message_text += item.additional_data
9     try:
10        crud.create_message(db=db, item=schemas.MessageCreate(text=message_text, from_user = ↵
11        ↵env_user_id, to_user = platform.id_user))
12    except (InvalidMessageIDException, InvalidUserException) as e:
13        raise HTTPException(status_code=400, detail= str(e))
14    text = _wait_and_return_message_for("ENV", db)["text"]
15    text_parts = text.split("###")

```

```
return {"text" : text_parts[0], "add_message_url" : text_parts[1], "get_message_url" : text_parts[2]}
```

Listing 5.11: Function that handles the communication during PHASE_3 and PHASE_4 of the environment's handshake protocol.

It starts by checking if the received message is the expected message for the current phase of the handshake protocol. Then, it creates a message that contains all the data necessary to complete PHASE_3 (PDDL domain, problem, and additional data), the ID of the system that sent the message (environment), and the system that needs to receive the message (EM-Glue). The environment sends the necessary data as a JSON object. However, it is converted to a single string by concatenating the fields with the separator `###` to be added to the database. Finally, it waits for the message necessary for PHASE_4 containing the links for normal communication. It formats them as the expected JSON object and returns it.

The two functions responsible for handling the messages received and sent from and to the experience manager during PHASE_3 and PHASE_4 are similar to the previous one. They follow the same steps: (i) check the message that is received, (ii) create the message for EM-Glue, and (iii) wait for the message that needs to be sent back to the experience manager.

5.4.2.3 Normal Communication API Calls

The last API call type is used during the normal communication between the experience manager and the environment. Six API calls are used for this purpose.

- GET request on `/get_messages_for_env`. The environment uses this API call to get the messages sent by the experience manager or EM-Glue. It returns a list of messages that could be empty if no messages are available.
- POST request on `/add_env_message`. The environment uses this API call to send a message to the experience manager or EM-Glue. It requires that the message is sent as a JSON object with the field `text` that contains the message and `to_user_role` containing the role of the receiver (either PLATFORM or EM). It returns the message that was sent.
- GET request on `/get_messages_for_EM`. The experience manager uses this API call to get the messages sent by the environment or EM-Glue. It returns a list of messages that could be empty if no messages are available.
- POST request on `/add_EM_message`. The experience manager uses this API call to send a message to the environment or EM-Glue. It requires that the message is sent as a JSON object with the field `text` that contains the message and `to_user_role` containing the role of the receiver (either PLATFORM, or ENV). It returns the message that was sent.
- GET request on `/get_error_messages`. The experience manager uses this API call to get the error messages the environment sends or EM-Glue. It returns a list of messages that could be empty if no messages are available.

- POST request on `/add_error_message`. The environment uses this API call to send an error message to the experience manager or EM-Glue.

These endpoints are configurable and can be changed by the user by editing the configuration file called `parameters.json`. Listing 5.12 shows the code of the function responsible for handling the POST request on the API endpoints that handle the messages the experience manager sends. All the other functions used to create new messages are similar to the one presented.

```

1 @app.post(parameters["url"]["in_em"], response_model=schemas.Message)
2 def add_experience_manager_message(item: schemas.MessageReceive, db: Session = Depends(get_db)):
3     if not _is_communication_enabled(db):
4         raise HTTPException(status_code=404)
5     env_id = crud.get_user_with_role(db, "EM").id_user
6     if _user_role_check(item.to_user_role):
7         to_id = crud.get_user_with_role(db, item.to_user_role).id_user
8     message = schemas.MessageCreate(text = item.text, from_user = env_id, to_user = to_id, ←
9         ↪old_message_id = item.old_message_id)
10    try:
11        res = crud.create_message(db=db, item=message)
12    except (InvalidMessageIDException, InvalidUserException) as e:
13        raise HTTPException(status_code=400, detail= str(e))
14    return res

```

Listing 5.12: API call that is used to add a new message from the experience manager during the normal communication.

The decorator of this function differs from the previous ones because the API endpoint is a variable configurable by the user. It starts by checking if the communication is enabled to understand if the initialization process has finished. Then, it gets all the data from the database necessary to insert a new message correctly. Finally, it creates the message and returns it.

The other functions are used to get the messages sent to the environment or the experience manager. Listing 5.13 shows the code of the function responsible for handling the GET request on the API endpoints that retrieves all the messages the experience manager needs to receive. All the other functions used to get the messages are similar to the one presented.

```

1 @app.get(parameters["url"]["out_em"], response_model=List[schemas.Message])
2 def get_messages_for_EM_not_sent(db: Session = Depends(get_db)):
3     if not _is_communication_enabled(db):
4         raise HTTPException(status_code=404)
5     res = crud.get_messages_not_sent_for_EM(db=db)
6     res = crud.update_sent_before_sending(query_result = res, db=db)
7     return res

```

Listing 5.13: API call that is used to get the new messages available to the experience manager during the normal communication.

As in the previous function, the decorator of this function contains as API endpoint specification a variable that is configurable by the user. It starts by checking if the communication is enabled to understand if the initialization process has finished. Then, it gets all the messages not sent where the recipient is the experience manager and updates the database to mark them as sent. Finally, it returns the messages as a list.

5.4.3 EM-Glue Manager

The EM-Glue Manager is a Python script used to manage the EM-Glue operations during the life cycle of an experience. It is responsible for starting the experience manager and the environment, directing the handshake process, and running the normal communication between the two systems.

Before starting an experience using EM-Glue, the user needs to configure the experience manager and environment that they want to use by editing the configuration file called `parameters.json`. In the configuration file, the user needs to specify the shell commands that EM-Glue needs to use to start the experience manager and the environment. With this information, the EM-Glue Manager can automatically start the experience manager and the environment after the API server and database are up and running, as the handshake protocol requires. The EM-Glue Manager uses the `subprocess` module of Python to start a new external processes: one for the experience manager and one for the environment.

The other primary responsibility of the EM-Glue Manager is to direct the initialization process. We have seen in the previous section that, during the initialization process, when the experience manager or the environment sends a new message, the message's recipient is the platform. In the EM-Glue Manager, there is a set of functions responsible for handling the messages sent by the experience manager or the environment to EM-Glue during the handshake protocol. Listing 5.14 shows the function that handles the handshake protocol.

```

1 def _communication_setup(self):
2     handshake_running = True
3     while self._is_platform_online() and handshake_running:
4         with SessionLocal() as db:
5             message = crud.get_first_message_not_sent_for_platform(db)
6             if message is None:
7                 time.sleep(0.1)
8                 continue
9             crud.update_sent_before_sending(query_result=[message], db = db)
10            protocol_phase = self.get_protocol_phase()
11            text = str(message.text)
12            if protocol_phase == "PHASE_1":
13                self._communication_protocol_phase_1(text)
14            elif protocol_phase == "PHASE_2":
15                self._communication_protocol_phase_2(text)
16            elif protocol_phase == "PHASE_3":
17                self._communication_protocol_phase_3(text)
18            elif protocol_phase == "PHASE_4":
19                self._communication_protocol_phase_4(text)
20            handshake_running = False

```

Listing 5.14: EM-Glue Manager function that handles the handshake protocol.

The function comprises a `while` loop that runs while the platform is online and the handshake phase is running. Inside the loop, the function gets the most recent message whose recipient is the platform and was not already handled and updates the database to mark it as sent. Then, it gets the current phase of the handshake protocol and the message's text. Finally, it calls the function responsible for handling the message based on the current phase of the handshake protocol.

The functions that handle the various phases of the protocol are similar to each other. The main objective is to check if the message received is valid and, if it is, create the following message that the experience manager or the environment needs

to receive and change the protocol phase to the next, whenever it is time to change. For example, I include in Listing 5.15 and 5.16 the functions that handle PHASE_3 and PHASE_4 since they are the most interesting of the four. They simultaneously handle the messages from the experience manager and the environment.

```

1 def _communication_protocol_phase_3(self, text : str):
2     if text == self.communication_phase_messages["PHASE_3"]["message_5"]:
3         self.phase3_part1_received = True
4     elif text.startswith(self.communication_phase_messages["PHASE_3"]["message_6"]):
5         self.phase3_part2_received = True
6         self.pddl_text = str(text)
7     if self.phase3_part1_received and self.phase3_part2_received:
8         with SessionLocal() as db:
9             crud.create_message(db = db, item = schemas.MessageCreate(text = self.pddl_text, ↵
10             ↵from_user=self._plt_id, to_user=self._em_id))
11         self._change_protocol_phase("PHASE_4")

```

Listing 5.15: EM-Glue Manager function that handles the PHASE_3 of the handshake protocol.

The function that handles PHASE_3 starts by checking if the message received is one of the two possible messages that the experience manager or the environment can send during this phase. If it is, it updates a class variable of the EM-Glue Manager to mark that the message has been received. Then, it checks if both messages have been received. When both messages have been received, it creates the message that the experience manager needs to receive and changes the protocol phase to the next one.

```

1 def _communication_protocol_phase_4(self, text):
2     if text == self.communication_phase_messages["PHASE_4"]["message_8"]:
3         with SessionLocal() as db:
4             t_ENV = self.communication_phase_messages["PHASE_4"]["message_9"] + "###" + self.↵
5             ↵communication_urls["in_env"] + "###" + self.communication_urls["out_env"]
6             crud.create_message(db = db, item = schemas.MessageCreate(text = t_ENV, from_user=self.↵
7             ↵_plt_id, to_user=self._env_id))
8             t_EM = self.communication_phase_messages["PHASE_4"]["message_10"] + "###" + self.↵
9             ↵communication_urls["in_em"] + "###" + self.communication_urls["out_em"]
10            crud.create_message(db = db, item = schemas.MessageCreate(text = t_EM, from_user=self.↵
11            ↵_plt_id, to_user=self._em_id))
12        self._change_protocol_phase("DONE")

```

Listing 5.16: EM-Glue Manager function that handles the PHASE_4 of the handshake protocol.

The function that handles PHASE_4 starts by checking if the message received is the one that it is waiting for from the experience manager. If it is, it creates the two messages that the experience manager and environment need to receive containing the API endpoints to use during normal communication. It then changes the protocol phase to DONE. Similarly to other messages, the message containing the API endpoints is composed of the concatenation of the URLs with the separator ###.

After the handshake protocol is completed, the EM-Glue Manager has a loop that keeps EM-Glue alive.

Summary

In this chapter, I have presented my proposed approach to the problem of separating and allowing partial-interchangeably between experience managers and environments. This approach is based on a design pattern where the experience manager and the environment are decoupled of each other and communicate through a intermediary layer that I called EM-Glue. EM-Glue is responsible for handling the communication between the experience manager and the environment and for providing the experience manager with the necessary information to interact with the environment. This communication is based on two protocols: a handshake protocol that is used during the setup of the experience to share data between the environment and the experience manager and a normal communication protocol that is used while the experience is running to exchange data between the experience manager and the environment.

Chapter 6

Case Study

In this chapter, I present a case study that demonstrates the use of EM-Glue to decouple experience managers from environments. The case study consists of four parts. First, in Section 6.1, I describe the *Camelot Wrapper*, software built to extend the *Camelot* environment and connect it to the platform. Second, in Section 6.2, I describe *PaSSAGE*, an existing experience manager adapted to be used with the platform. Third, in Section 6.3, I describe a random experience manager that I developed to test the ability of the platform of supporting interchangeability. Finally, in Section 6.4, I present the evaluation of this case study by showing the equivalence between the implementation of PaSSAGE in its joint version and the disjoint version presented in Section 6.2, and by highlighting the differences between PaSSAGE and the random experience manager. This section also shows how the experience managers and environment can communicate successfully using EM-Glue.

6.1 Camelot Wrapper

To test if EM-Glue can be successfully used to decouple experience managers and environments, I needed to develop an environment compatible with the platform. I had two options: either develop a new environment from scratch or use an existing one. Developing a new environment from scratch would have been the safest option since it would have allowed me to control the entire development process but at the cost of time and effort. Instead, I decided to use an existing environment to show the capability of EM-Glue to work with an environment that was not initially designed to be used with it. Another benefit of this approach is that I could document and assess the process of adapting an external system to the platform. This approach is beneficial because it allows other researchers to use a similar process to adapt their environments to EM-Glue.

The first step was to identify an environment; ideally one designed for use in a similar application that could be modified to work with EM-Glue. As seen in Section 3.4, some environments could potentially be used for this purpose. I considered them based on the following criteria:

- it must have an explicit declaration of the actions that can be executed (with preconditions and effects),
- it must share information about the state of each player’s experience within the environment,
- it must accept instructions that can change the environment’s content or progression during a player’s experience (e.g., move an NPC or create a new item),
- it must keep track of the state of the environment’s world using an abstract state representation.

To the best of my knowledge, Camelot¹ [111, 117] is the only visualization engine with two of these four features: it shares information about what is happening in the environment and accepts instructions to change the environment’s content or progression. Another benefit of Camelot is that it was designed for applications where the experience manager is decoupled from the environment. The purpose of its design was to enable it to be employed like EM-Glue, to be used by multiple experience managers. However, I could not use Camelot as-is for four reasons.

First, Camelot is designed to *not* keep track of the state of the environment’s world [117]. EM-Glue needs the state of the environment to be shared with the EM using a common language (PDDL). This is a core feature that EM-Glue uses to enable the decoupling of the experience manager from the environment.

The second reason is that Camelot does not provide an explicit declaration of the actions that can be executed. They provide a list of actions Camelot supports but do not provide a formal description of the actions using preconditions and effects. This problem is connected to the first one because Camelot does not have a state where these conditions could be checked.

Third, as introduced in Section 3.4, Camelot requires a highly specific set of instructions to work (Camelot instructions). This is a problem because in EM-Glue I decided to use PDDL to exchange information between environments and experience managers to increase its generality. Camelot instructions are designed to be used with Camelot, and there is no direct way to translate them into PDDL.

Finally, Camelot requires a connected EM to control almost everything that happens in a player’s experience. This includes responding to most player inputs (e.g., interacting with NPCs or objects) as well as moving and animating all NPCs. The only dynamic enabled directly by Camelot is the fine-grained player movement (e.g., moving within a room). As a result, most of the Camelot environment’s dynamics must be implemented outside of Camelot itself. Comparing two EMs in a single Camelot environment becomes difficult, as both EMs would need to implement the same dynamics consistently. It would also be difficult to test those EMs in a different environment that implemented more of its dynamics (e.g., a commercial role-playing game).

¹Throughout the dissertation, I have used italic font when referring to external systems. However, since Camelot is an integral part of this chapter, from now on, I will not use italic font when referring to it.

To overcome these four significant issues that prevented me from using Camelot directly with EM-Glue, I developed a solution that wraps the Camelot environment and acts as a middle layer between EM-Glue and Camelot. The *Camelot Wrapper* is designed to overcome the four problems identified above as follows:

- It translates from PDDL instructions to Camelot instructions. This translation is done using a set of rules defined in a configuration file that maps PDDL actions to Camelot instructions.
- It handles more of the environment's dynamics and low-level interactions than Camelot does on its own. This includes the movement of NPCs and (basic) interaction with objects (e.g., opening a chest without the manager's intervention).
- It keeps track of the current state of Camelot's world. This is done by interpreting the Camelot instructions that Camelot shares to communicate what is happening in the environment.
- It provides an explicit declaration of the actions that can be executed in Camelot. This is done using the PDDL domain file to define the actions that Camelot can execute.

The Camelot Wrapper is open source and available on GitHub [74]. The source code of the wrapper is written in Python 3.9.1. Figure 6.1 describes the design of the Camelot Wrapper, which spans different software components. These components are not a one-to-one representation of the scripts that form the Camelot Wrapper but a high-level description of the main components that make up the wrapper. This figure also shows the flow of information between the components.

I divided the presentation of these components into three categories. First, Sections 6.1.1 to 6.1.3 describe the components that facilitate external communication. With external communication, I refer to the software components of the Camelot Wrapper that handle the communication with EM-Glue and Camelot. When dealing with I/O operations, there is a problem of long waits when standing by for new messages. The reason behind this problem is that I/O operations are blocking, which means that the program will wait for the operation to finish before continuing. This is a problem because it means that the program will be stuck waiting for a message to arrive and unable to process other tasks. To solve this problem, a possible solution is to use threads. A thread is a unique execution route within a program. It is a lightweight process that the operating system can schedule and run concurrently. Threads are produced and managed by the operating system, sharing memory and resources with the programs that created them. This allows several threads to cooperate and operate effectively within a single application. Given the capability of concurrency, all of the components that deal with communication run on separate threads. Separating these components into separate threads allows the application to operate efficiently while waiting for input. This choice adds complexity to the wrapper's code because it requires handling concurrency and queues to exchange messages, but the final result benefits from a smoother operation since it solves the blocking problem.

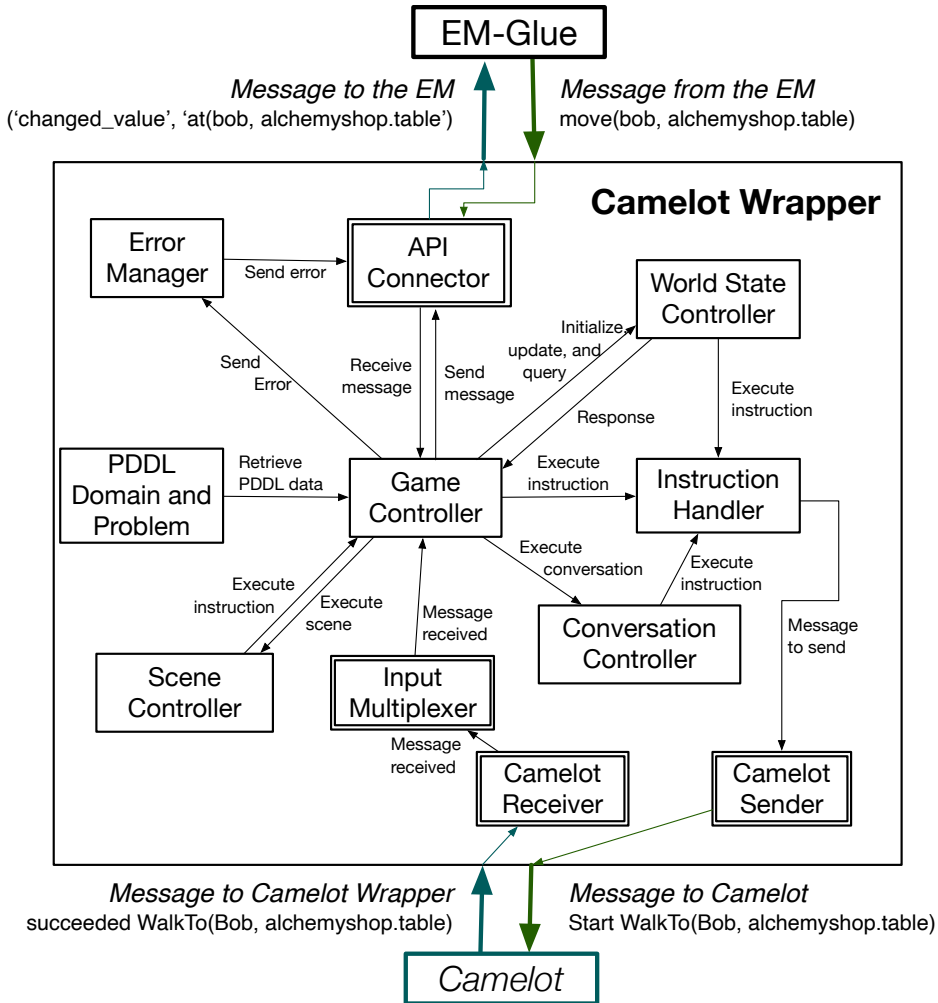


Figure 6.1: A schematic diagram of the Camelot Wrapper's main components. Double boxes indicate that the component is executed in a separate thread. The arrows indicate the flow of information between components.

Secondly, Sections 6.1.4 to 6.1.8 describe the components that handle the translation and the internal logic of the wrapper. The objective of this category of components is to translate the messages sent by the EMs to Camelot and vice-versa. At the beginning of this chapter, I mentioned that one of the main objectives of the Camelot wrapper is to translate between the PDDL language and Camelot instructions. Translating between these two formalisms is challenging, as the two languages have different structures and conventions. The Camelot wrapper addresses this challenge by providing a comprehensive set of tools to automate and streamline the translation process while ensuring the accuracy and completeness

of the translation.

Lastly, Sections 6.1.9 to 6.1.11 describe the components that handle the game in a Camelot environment. These components of the Camelot wrapper need to handle two types of dynamics: low level and high level. A low level dynamic is a sequence of actions that can be executed by the player without the need of an external intervention. For example, opening doors and moving between locations. Without an external intervention, once a player is setup in a room, they can only move within that room. This is the default behaviour in a Camelot environment.

An high level dynamic refers to a sequence of actions that involve complex aspects of a game environment and require external intervention to be properly executed. In the case of the Camelot wrapper, high level dynamics correspond to managing gameplay elements such as conversations and scenes that are specific to the environment but not automatically handled by Camelot's low level interactions. During the development of the first experience manager, I realized that the Camelot wrapper needed to handle these high level dynamics. Given that they required additional processing beyond what Camelot provided, they needed to be handled by the wrapper automatically.

6.1.1 API Connector

The API connector is the component that handles the communication with EM-Glue using the APIs that are available in the platform. This component handles both incoming and outgoing communications. For incoming messages, it uses a service that makes HTTP requests on the link provided with the communication protocol every 250 milliseconds to check for new messages. As described in Section 5.2, this is the service needed to receive messages from the platform with the lowest delay possible. For outgoing messages, it makes an HTTP request on the link for sending environment messages to share with the experience manager the changes to the world state.

6.1.1.1 Implementation

The API connector is implemented using a Python class called `PlatformIOCommunication`. This class is a **singleton** to ensure that only one class instance is created. This is necessary because the API connector is used in multiple parts of the scripts of the Camelot Wrapper, and all of these parts must use the same instance of the class. The class hosts a queue to store the incoming messages from the platform. Two methods are used to communicate with EM-Glue: the service for receiving messages and the method for sending messages. The service for receiving messages is a thread that is started when the class is instantiated. Listing 6.1 shows the code for the thread that handles the incoming messages.

```

1 def __receive_message_thread(self, message_queue: queue.Queue):
2     while self._is_platform_online():
3         message = self.receive_message()
4         if message != "":
5             message_queue.put(message)

```

Listing 6.1: Thread that handles the incoming messages from EM-Glue.

This thread is a `daemon` thread, meaning it will be terminated when the main thread is terminated. This is necessary because the thread is waiting for messages to arrive, and if it is not terminated when the main thread is terminated, it will block the application from terminating. It comprises a `while` loop that runs as long as the platform is online. It checks for new messages using the `receive_message` method described in Listing 6.2. If a message is received, it is added to the queue.

```

1 def receive_message(self) -> str:
2     if self._is_platform_online():
3         response = requests.get(self.base_link + self.receive_message_link)
4         if response.status_code == 200:
5             if response.json() == []:
6                 return None
7             else:
8                 return response.json()
9     return None

```

Listing 6.2: Function used to send a message to EM-Glue.

The `receive_message` method performs a GET request to the link for receiving messages. If the request is successful and the content is not empty, it returns the message.

The method for sending messages is more complex, as it needs to perform differently based on whether it follows the handshake or normal communication protocols. The `send_message` method is described in Listing 6.3.

```

1 def send_message(self, message, initialization = False):
2     if self._is_platform_online():
3         if initialization:
4             if type(message) == str:
5                 data = {'text': message}
6                 response = requests.post(self.base_link + self.initial_message_link, json = data)
7             elif type(message) == dict:
8                 response = requests.post(self.base_link + self.initial_message_link, json = message)
9             else:
10                return None
11        else:
12            message_preparation = {'text':message,'to_user_role' : 'EM'}
13            response = requests.post(self.base_link + self.send_message_link, json = ↵
↵message_preparation)
14        if response.status_code == 200:
15            return response.json()
16        else:
17            return None

```

Listing 6.3: The `send_message` method that is used to send messages to EM-Glue.

The `send_message` method starts by checking if the platform is online. If it is, it checks if it follows the handshake or normal communication protocols. In the case of the handshake protocol, it checks if the message that is about to be sent is a string or a dictionary to format it correctly and send it to the platform. When the handshake protocol has finished, the message is prepared using the rules of the normal communication protocol and sent to the platform. In both cases, the method checks if the request was successful and returns the response.

6.1.2 Camelot Receiver and Sender Threads

As mentioned previously, the Camelot Wrapper uses threads to handle all external communications, including communication with Camelot. Camelot communicates with EMs (or, in this case, this wrapper) via standard I/O. The standard input (or output) is an input stream where data is sent to and read by a program. This stream of data is identified as a file descriptor. A file descriptor is a number that uniquely identifies an open file in a computer's operating system. It describes a data resource and how that resource may be accessed. A problem with using this type of communication is that writing and reading operations are mutually exclusive. This means, for example, that if the wrapper is waiting for a message from Camelot to arrive, it cannot send any messages *to* Camelot, nor vice-versa. If this happens, the wrapper will be stuck waiting for a message that will never arrive, causing a deadlock. In the context of this wrapper, if a deadlock happens, it causes a catastrophic failure of the communication between the wrapper and Camelot that will finally result in an experience-breaking error.

I came across this type of deadlock situation during the early phases of the wrapper development process. The first solution I tried to implement was to interrupt the read operation after a certain time. This would require operating system-level calls that not all operating systems support (e.g., Microsoft Windows does not allow it). Since Camelot can be used in Windows and MacOS, I decided to look for a different solution to keep the wrapper cross-platform.

The problem I needed to solve is also known as a race condition, a common problem in concurrent programming. The condition occurs when one thread tries to modify a shared resource while another thread is modifying that resource, and the solution requires the threads to be synchronized. Thread synchronization can be achieved in multiple ways, but in this case, I decided to implement a solution that uses thread locks and thread events. A thread lock allows only one thread to access the shared resource at a time. When a thread acquires the lock, all other threads that try to acquire it will be blocked until it is released. The basic idea behind a thread lock is that a thread must acquire the lock before accessing a shared resource and then release it when it is done. Meanwhile, a thread event allows one or more threads to wait for a particular event to occur before proceeding with their execution. In a typical use case, a thread will hold off on executing until another thread signals the occurrence of an event.

In my solution, the thread lock controls access to the read and write operation on the standard input and output streams to Camelot. Meanwhile, the thread event stops the receiver thread from constantly acquiring the lock on the standard input stream and waiting for the sender thread to send a message if available.

6.1.2.1 Implementation

The Camelot receiver and sender threads are implemented in a singleton Python class called `CamelotIOCommunication`. To handle the multi-threading aspect of the communication with Camelot, I used the `threading` module of Python. When an object of this class is first created, there is the execution of the initialization function that creates the thread lock and thread event that will be used to synchronize the

threads. Two possible states for a lock are *locked* or *unlocked*. A lock object has two basic methods, `acquire()` and `release()`. When the state is unlocked, `acquire()` changes the state to locked and returns immediately. When the state is locked, `acquire()` blocks until a call to `release()` in another thread changes it to unlocked, then the `acquire()` call resets it to locked and returns. Meanwhile, an event object manages a flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

Moreover, the initialization function creates the queues that will be used to store the messages before sending them to Camelot and the messages that are received from Camelot. Queues are commonly used in multi-threaded programs to provide a way for threads to communicate in a thread-safe manner. A queue is a data structure that allows elements to be added to one end and removed from the other end. In this case, the queues share data between the main thread, the Camelot receiver, and the sender threads. Once all the initialization of the objects needed for the threads to work is completed, the last step is to start the daemon thread that will send messages to Camelot. Listing 6.4 shows the code for the Camelot sender thread.

```

1 def __camelot_sender_thread(self, queue: queue.Queue, is_running: bool, lock: threading.Lock, ↔
  ↔event_obj: threading.Event):
2     while(is_running):
3         event_obj.clear()
4         message = queue.get()
5         if self.__started == False:
6             self.__started = True
7             self.__start_receiver_thread(lock, event_obj)
8         if message == "kill":
9             is_running = False
10            break
11        if message != "%PASS%":
12            self.__standard_IO_operations(message, 0, lock)
13        event_obj.set()

```

Listing 6.4: Camelot Sender thread implementation.

This thread runs in a loop until the `is_running` variable is set to `False`. Every time there is a new execution of the loop, the thread will `clear` the event object and get the following message from the queue. The idea behind this event is to give the Camelot sender thread enough time to wait for an available message from the queue and send it to Camelot. When a message is ready to be sent to Camelot, two checks need to be done. First, the thread checks if the Camelot receiver thread has been started. If it has not been started, then the thread starts it. This is done because I need to ensure that the Camelot receiver thread is started only after the Camelot sender thread has been started and is ready to send a message. This is to prevent the Camelot receiver from acquiring the lock on the input stream before the Camelot sender can send the first message to Camelot and start the experience. Second, the thread checks if the message is a `kill` message. If so, the thread sets the `is_running` variable to `False` and breaks the loop. This stops the Camelot sender thread when the Camelot Wrapper is closed. Finally, the thread checks if the message is a `%PASS%` message. If it is not, the thread sends the message to Camelot using the `standard_IO_operations` function (shown later in this section).

Otherwise, the `%PASS%` message indicates that the Camelot Wrapper does not need to send a message before it receives another one from Camelot. In any case, the thread will `set` the event object to indicate that the message has been sent to Camelot, and now the receiver can continue its operation.

The Camelot receiver thread is the daemon thread used to receive messages from Camelot. Listing 6.5 shows the code for the Camelot receiver thread.

```

1 def __camelot_receiver_thread(self, queue: queue.Queue, is_running: bool, lock: threading.Lock, ←
    ↪event_obj: threading.Event):
2     timeout = 1.0
3     while(is_running):
4         event_obj.wait(timeout=timeout)
5         message = self.__standard_IO_operations(None, 1, lock)
6         if message == None:
7             time.sleep(0.1)
8             continue
9         queue.put(message)
10        if message == "input Quit":
11            is_running = False

```

Listing 6.5: Camelot Receiver thread implementation.

This thread runs in a loop until the `is_running` variable is set to `False`. Every time there is a new execution of the loop, the thread will `wait` for the event object to be `set` by the Camelot sender thread. When the event is `set`, the thread will read the message from Camelot using the `standard_IO_operations` function. If the message is `None`, the thread will sleep for 0.1 seconds and continue with a new iteration of the loop. Otherwise, the thread will put the message in the queue and check if the message is a `input Quit` message. If so, the thread sets the `is_running` variable to `False` to stop the loop.

The `standard_IO_operations` function is used to send and receive messages from Camelot. It is used by both the Camelot sender and receiver threads, and it uses the lock to ensure that only one thread can access the input and output streams at a time.

```

1 def __standard_IO_operations(self, message: str, mode: int, lock: threading.Lock) -> str:
2     lock.acquire()
3     return_message = None
4     if mode == 0:
5         if message == None:
6             return None
7         print(message)
8         return_message = "OK"
9     elif mode == 1:
10        return_message = input().strip()
11        lock.release()
12    return return_message

```

Listing 6.6: Function that handles the I/O operations with Camelot.

The first step of the function is to `acquire` the lock to ensure that only one thread can access the input and output streams at a time. Then, the function checks the `mode` parameter. If the mode is 0, the function will send the message to Camelot using the `print` function. Otherwise, if the mode is 1, the function will read a message from Camelot using the `input` function. After the message is sent or received, the function will `release` the lock and return the message.

6.1.3 Input Multiplexer Thread

The objective of the input multiplexer is to get the messages that the receiver thread receives from Camelot and to sort them into specific queues to handle each type of message differently. An example of a message that Camelot uses to share updates is shown in Listing 6.7.

```
1 input arrived bob position alchemyshop.Door
```

Listing 6.7: An example of a message that comes from Camelot.

In this example, Camelot is communicating that an `input` happened in the environment where the action `arrived` was executed by the character `bob` in the `position` that corresponds to the `alchemyshop.Door`. The input multiplexer thread can sort the messages into five categories based on the following criteria:

- A *success message* is one that starts with the `succeeded` string. Camelot sends this message when an action previously sent by the platform is executed with success.
- A *location message* is one that starts with the `input` string and continues with one of the following sub-strings: `arrived`, `started walking`, `stopped walking`, or `exited`. Camelot sends these messages when a character moves in the environment.
- An *input message* is one that starts with `input` and is not a location message. An input message is sent by Camelot when the user clicks on an entity that can accept inputs (e.g., a chest that can open).
- An *error message* starts with one of the following words: `error`, `failed`, or `exception`. This type of message is generated when an error occurs in Camelot.
- Another type of message is when a Camelot message starts with anything that is not the things listed above. Those are messages that the wrapper does not support, and they are stored in a queue to help debug the platform.

For each category, the game controller deals with the message differently. If there is an error message, the error manager takes charge of the message.

6.1.3.1 Implementation

The input multiplexer is implemented as a daemon thread. It comprises five queues that are used to store the messages received from Camelot once they are sorted. The sorting operation is done in the `input_messages_management` function presented in Listing 6.8.

```
1 def _input_messages_management(self):
2     while self.__thread_running:
3         message = self.camelot_IO_communication.get_message()
4         if message == "input Quit":
5             self.__thread_running = False
6         if message.startswith("succeeded"):
```

```

    self.__success_queue.put(message)
8  elif message.startswith("input"):
    if message.startswith(shared_variables.location_message_prefix):
10     self.__location_queue.put(message)
    else:
12     self.__input_queue.put(message)
    elif message.startswith("started"):
14     self.camelot_IO_communication.print_action("%PASS%")
    elif message.startswith("error") or message.startswith("failed") or message.startswith("↔
↪exception"):
16     self.__error_queue.put(message)
    else:
18     self.__other_queue.put(message)

```

Listing 6.8: Function that sorts the messages that come from Camelot.

It comprises a loop that runs until the `thread_running` variable is set to `False`. Every time there is a new iteration of the loop, the thread will get a message from the Camelot receiver thread. The `get_message` function is a blocking function that will wait until a new message is available in the queue. Once the message is received, the next step is to sort the message into one of the five queues based on the criteria presented previously. However, there is a particular case not listed in the criteria when the message is `input Quit`. In this case, it is the sign that Camelot was asked to finish its execution, and the thread will set the `thread_running` variable to `False` to stop the loop.

6.1.4 PDDL Domain and Problem

The PDDL domain and problem files are the sources of information for the initialization and translation processes. In section 5.3.1, I described the original scope of the PDDL domain and problem files when they were used in automated planning. In this case, the PDDL domain and problem files are used to give an abstract description of the Camelot environment with all the actions that are possible to perform in the environment, the available entities, and the initial state of the environment.

The domain file provides a high-level description of Camelot, which captures the underlying structure of the environment, independent of any specific initial state in that Camelot will be initialized. The domain file defines the types of objects that exist in the world, the actions that can be performed on those objects, and the preconditions and effects of those actions. The PDDL domain comprises three main parts used to declare the entities, the actions, and the predicates available in the environment.

- **Entity types:** PDDL types restrict the objects that can be used as parameters in actions and predicates. By defining types and subtypes, we can create a hierarchy of objects and specify constraints on which objects can be used in a particular context. For example, if we define a type `vehicle` and two subtypes `car` and `plane`, we can create actions and predicates that are specific to each subtype. So, we can create a predicate `canCarryPeople` that applies to the type `vehicle` and, consequently, to the subtypes `car` and `plane`. However, we can also create a predicate `canFly` that only applies to type `plane`. Listing 6.9 shows an example of a PDDL entity declaration.

```

1  (:types
2     vehicle — object
   car plane — vehicle
4  )

```

Listing 6.9: Example of a PDDL entity declaration

- **Predicates:** PDDL predicates describe properties or relations between objects in the PDDL domain and problem. Predicates are used to define the initial state of the experience, and the conditions and effects of actions that can be applied to objects in the problem. They consist of a name followed by a list of one or more parameters type in parentheses. For example, we could take the definition of the two predicates `canCarryPeople` and `canFly` from the previous example and define them as follows in Listing 6.10.

```

1  (:predicates
2     (canCarryPeople ?v — vehicle)
   (canFly ?p — plane)
4  )

```

Listing 6.10: Example of a PDDL predicate declaration.

- **Actions:** PDDL actions are named procedures that can be executed to transform one state of the environment into another. In this wrapper, the action section of the domain is used to specify all the action that Camelot supports in PDDL form. Actions are defined by specifying their name, parameters, preconditions, and effects. The parameters of an action are defined using a list of variables in parentheses, which represent the objects on which the action will operate. The preconditions are a set of predicates combined using logical statements (e.g., `and`, `or`, `not`) that must be true in order for the action to be applicable. The effects are a set of predicates describing the changes the action will make to the state. For example, we can define an action `fly` that takes a plane `p` as a parameter and as preconditions the predicate `canFly p` being true and `isFlying p` being false. The effect of the action is to change the predicate `isFlying p` to true into the state. Listing 6.11 shows this action declaration.

```

1  (:action fly
2     :parameters (?p — plane)
   :precondition (and (canFly ?p) (not (isFlying ?p)))
4     :effect (and (isFlying ?p))
   )

```

Listing 6.11: Example of a PDDL action declaration.

Once the domain is defined, the problem file provides specific details about the Camelot environment, including the objects that exist in the world and the initial state of the world. Essentially, the problem file “solidifies” the domain file by specifying the specific instantiation of the problem. Similarly to the domain file, the problem file comprises three main parts: the objects, the initial state, and the goal state.

- **Objects:** The object section of the problem file is where we can define the types of objects in the world. The objects section typically contains a list of object type definitions, where each type definition includes the name of the type and a list of objects that belong to that type. Following the example defined before, we can define the objects that will be in the environment `ferrari` of type `car`, `boeing747` and `airbusA380` of type `plane`. Listing 6.12 shows this object declaration.

```

1  (:objects
2     ferrari – car
   boeing747 – plane
4     airbusA380 – plane
   )

```

Listing 6.12: Example of a PDDL object declaration.

- **Initial State:** The initial state section of the problem file specifies the initial state of the world and specifically which predicates are true at the beginning of the problem. The problem has the “closed world” assumption applied, meaning that anything not specified as true is considered false. For example, if we want to define the initial state of the problem to be that the plane `boeing747` and `airbusA380` can fly, we can define the predicate `canFly` to be true for the object `boeing747` and `airbusA380`. Listing 6.13 shows this initial state declaration. Differently from the standard PDDL specification, the initial state of the environment may be incomplete, in a sense, since I allow actions to create new entities as described in Section 5.3.3.

```

1  (:init
2     (canFly boeing747)
   (canFly airbusA380)
4  )

```

Listing 6.13: Example of a PDDL initial state declaration.

- **Goal State:** The goal section of a problem file specifies the desired state of the world that the planner should aim to achieve. It defines the conditions that must be satisfied for the planning problem to be successfully solved. In the context of the Camelot wrapper, the goal section does not serve any purpose at this time, but the PDDL standard requires it. As a result, this part of the file can be left empty.

6.1.5 PDDL Data Framework

The PDDL domain and problem file are parsed by the PDDL parser component to be used by the Camelot wrapper. During the parsing process, the PDDL parser transforms the PDDL domain and problem into a PDDL data framework I created as a Python library. This library is used by the Camelot wrapper to initialize and keep track of the world state and to translate PDDL actions into Camelot instructions. Like EM-Glue and the Camelot wrapper, this library is also open source and available on GitHub [72].

6.1.5.1 Implementation

The PDDL data framework is a Python library that I developed to simplify the process of using and maintaining the PDDL specification in the Camelot wrapper using classes and data objects provided by Python. Figure 6.2 shows a high-level diagram of the PDDL data framework. All the boxes in the diagram represent

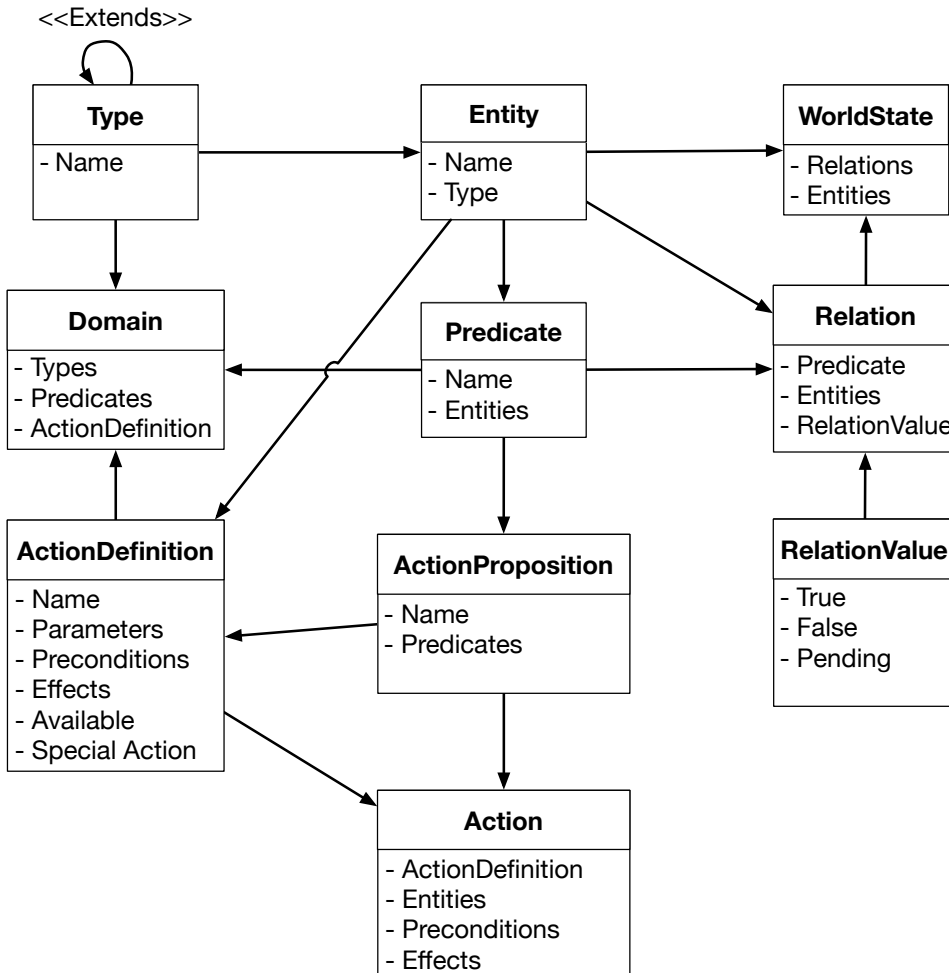


Figure 6.2: A diagram showing the PDDL data framework used to transform the PDDL specification into a Python framework within the Camelot Wrapper.

classes that are part of the PDDL data framework.

- **Type:** The `type` class represents a PDDL type. It comprises two parameters: the `name` of the type and an `extend` parameter that specifies the type that this type extends. For example, the type `car` extends the type `vehicle`.

- **Entity:** The `entity` class represents a PDDL entity. It is composed of two parameters: the `name` of the entity and the `type` of the entity. For example, the entity `ferrari` is of type `car`.
- **Predicate:** The `predicate` class is used to represent a PDDL predicate. It is composed of two parameters: the `name` of the predicate and a list of `arguments` that the predicate takes. For example, the predicate `canFly` takes an argument `plane`.
- **ActionProposition:** The `actionProposition` class is used to represent the proposition that forms a precondition or effect in a PDDL action. It is composed of two parameters: the `name` of the conjunction that is used in the action proposition (e.g., `and`, `or`) or the negation (e.g., `not`), and a list of `arguments` that are the predicates that form the precondition or effect. For example, in the action `fly`, the action proposition that might form the precondition is `(and (canFly ?p) (not (isFlying ?p)))` and it is represented as a series of `actionProposition` objects.
- **ActionDefinition:** The `actionDefinition` class is used to represent the definition of a PDDL action. It is composed of six parameters: the `name` of the action, a list of `parameters` that are the entities that are used in the action, two objects of type `actionProposition` that form the preconditions and effects of the action, an `available` parameter that is used to specify if the action is available in the environment, and a `special_action` parameter that specifies if an action has special effects (e.g., instantiate a new entity). The `available` parameter is needed because an action could be specified in the PDDL domain file but not be available in the environment. This happens when the Camelot instructions that are needed to execute the action are not specified (more details in Section 6.1.6.2.3).
- **Domain:** The `domain` class represents the PDDL domain. It is composed of four parameters: the `name` of the domain, a list of `types` that are the types that are defined in the domain, a list of `predicate` objects that are the predicates that are defined in the domain, and a list of `actionDefinition` objects that are the actions that are defined in the domain. After parsing a PDDL domain file, the `domain` object is created.
- **RelationValue:** The `relationValue` class represents the value of a PDDL relation. It is an `enum` object that can have four possible values: `TRUE`, `FALSE`, `PENDING_TRUE`, and `PENDING_FALSE`. However, the current version of this wrapper only uses the values `TRUE` and `FALSE`. The `PENDING_TRUE` and `PENDING_FALSE` values exist for future use when the wrapper is extended to support uncertainty when an action is executed.
- **Relation:** The `relation` class represents a PDDL relation. It is composed of three parameters: the `predicate` of the relation, a list of `entity` that are the entities that are used in the relation, and a `value` that is an object of the class `RelationValue`.

- **Action:** The `action` class represents a PDDL action. It is composed of four parameters: the `ActionDefinition` of the action, a list of `entity` that are the entities that are used in the action, and two `ActionProposition` objects that compose the preconditions and effects of the action.
- **WorldState:** The `worldstate` class represents the abstract state of the environment. It is composed of two parameters: a list of `entity` objects that are the entities available in the environment and a list of `relation` objects that are the relations that are true in the environment.

All these classes not only define the structure of the PDDL data framework but also provide methods to manipulate the data. These functions have various purposes, such as finding elements within objects, checking if an object is equal to another, checking if a relation is true in the world state, or checking if an action can be applied to the world state. Listing all the methods of these classes is out of the scope of this dissertation, but they are all documented in the library's source code.

The PDDL parser uses the PDDL data framework to create objects representing the PDDL domain and problem. A parser is a script that analyzes the structure of a text input and breaks it down into its parts according to a specific syntax. In this case, the PDDL parser reads the PDDL domain and problem files and parses their content into the PDDL data framework. The PDDL parser is implemented in Python, and I had to write the parser from scratch because there was no existing parser (at the time) for PDDL that I could use with Python. All the functions necessary for the parsing are implemented in the `PDDL_Parser` class.

The parsing process starts by reading the PDDL domain and problem files and storing their content in a string. The parser uses a regular expression "`[()|[\t()]+`" to divide the string into tokens based on the characters `(`, `)`, and `space`. Then, it checks all the tokens to find the relative keywords of the PDDL specification (e.g., `:domain`, `:action`, etc.) and executes the corresponding function to parse the content of that section of the PDDL specification. For example, if the parser finds the keyword `:action`, it executes the function `parse_action` that parses the content of the action section. The parser also checks if the PDDL specification is valid by checking if the keywords are in the correct order and if the content of each section is valid. Once each section of the PDDL specification is parsed, the parser creates the corresponding objects of the PDDL data framework.

6.1.6 Camelot World State

The Camelot world state component is a key part of the Camelot wrapper, as it handles the abstract world state representation of the Camelot environment. Essentially, the world state component is responsible for keeping track of everything happening within the Camelot world. The two primary purposes of the world state component are to initialize the world state and keep track of it throughout the game. To do so, it must handle the translation between Camelot instructions and PDDL. There are three types of translations: the translation of the initial state, messages to Camelot, and messages from Camelot.

6.1.6.1 Initialization

The first translation that the world state component performs is when it is setting up the Camelot game environment by translating the abstract world state into a format that Camelot can understand. This is done during initialization when the world state component uses the PDDL data framework to parse the PDDL domain and problem and create the necessary objects.

6.1.6.1.1 Problem

The problem that I aim to solve with this translation is to create a Camelot environment starting from a high-level description of the environment in the form of a PDDL problem. As I described in Section 6.1.4, the PDDL problem file contains two key components for the initialization of the Camelot environment: the objects and the initial state. As a result, there are two types of translation that needs to be performed: the translation of the PDDL objects in Camelot “create” instructions and the PDDL initial state into a format that Camelot can understand.

Firstly, the `objects` section is used to define all the entities that will be part of the Camelot environment. For example, it can declare two objects as `AlchemyShop - location` and `bob - player`, which will be used to instantiate an `AlchemyShop` location in the Camelot environment and a `bob` character that will be the player. I need to translate the PDDL objects declaration into Camelot instructions to perform this process in Camelot. Three Camelot instructions can be used to create the objects: `CreatePlace` is used to create a location, `CreateCharacter` is used to create a character, and `CreateItem` is used to create an item. As a result, the solution to this problem should be able to identify the object type and translate it into the corresponding Camelot instruction.

Secondly, the initial state section defines the relations between the entities that form the initial state of the environment. For example, it can declare the relation `in bob AlchemyShop`, which will be used to place the `bob` character in the `AlchemyShop` location. Similarly to the objects, the challenge is to translate the PDDL initial state into Camelot instructions. For example, the relation `adjacent AlchemyShop.Door City.GreenDoor`, that indicates that two doors are connected, needs to be translated in the Camelot instructions that will connect the two doors together by using an `EnableIcon` instruction. In this case, many different Camelot instructions can be used, and the connection between a predicate and a Camelot instruction is not straightforward since one predicate could correspond to multiple Camelot instructions. As a result, any solution to this problem needs to allow a designer to flexibly specify the connection between the PDDL predicate and the Camelot instructions so that send the corresponding Camelot instructions can be sent to Camelot whenever a predicate is processed by the controller.

6.1.6.1.2 Solution Design

The two problems described in the previous section each require a different approach to solve them. The first problem, the translation of the PDDL objects into Camelot instructions, can be solved by creating a connection between the PDDL

object type and the corresponding Camelot instruction. To do so, in the entity type section of the PDDL domain file, I created three main types of entities, each of which is associated with a Camelot instruction: `position` is associated with `CreatePlace`, `item` is associated with `CreateItem`, and `character` is associated with `CreateCharacter`. As a result, the solution to this problem is to identify the object type during the parsing, connect to the entity's name as described in the PDDL problem file, and translate it into the corresponding Camelot instruction following the Camelot syntax.

The second problem, the translation of the PDDL initial state into Camelot instructions, is more complex since there is no direct connection between the PDDL predicate and the Camelot instruction for the objects. This correlation cannot be automatically inferred from the PDDL domain file or the description of the Camelot instructions, and it needs to be manually defined. To solve this problem, I have created a JSON file that contains the list of the PDDL predicates currently supported by the Camelot wrapper and the corresponding Camelot instructions that need to be executed during the initialization. Listing 6.14 shows an example of an entry of such file for the `adjacent` predicate. Each key element of the JSON file is the name of the predicate. The `adjacent` predicate indicates that the two locations are adjacent to each other. The value corresponding to each key is divided into three other keys: `declaration`, `input`, and `response`. I describe the `input` and `response` keys later on in Section 6.1.6.3.2. The `declaration` key contains the Camelot instruction that should be executed when the predicate is encountered while parsing the PDDL problem's initial state. In this case, declaring two places adjacent means we need to create a connection between the two places. This means that we need to enable the click action (using the `EnableIcon` Camelot instruction) on a door, as this allows the player to declare the intention to move from one place to the other (line 3).

```

1  {"adjacent": {
2    "declaration": [{
3      "action_name": "EnableIcon",
4      "action_args": ["Exit", "Exit", "$param1$", "Exit from $param1$", "TRUE"]
5    }],
6    "input": {
7      "door": "input Exit $param1$",
8      "end": "input arrived $param2$ position $param1$"
9    },
10   "response": [{
11     "action_name": "WalkTo",
12     "action_args": ["$param2$", "$param1$"]
13   }, {
14     "action_name": "FadeOut",
15     "action_args": []
16   }, {
17     "action_name": "SetPosition",
18     "action_args": ["$param2$", "$param3$"]
19   }, {
20     "action_name": "SetCameraFocus",
21     "action_args": ["$param2$"]
22   }, {
23     "action_name": "FadeIn",
24     "action_args": []
25   }
26 }

```

Listing 6.14: Example of a JSON entry file that maps predicates to Camelot instructions.

6.1.6.1.3 Implementation

The solution to the first problem is based on a three step approach. First, the PDDL problem file is parsed to identify the objects that must be created in Camelot. Second, the Camelot instructions are created based on the object type. Third, the Camelot instructions are sent to Camelot to create the objects in the environment. I have already described this solution’s first and last steps in Sections 6.1.2 and 6.1.5, and so, focus on the second step of the solution here: creating the Camelot instructions based on the object type. We can identify three types of objects that Camelot can support: “locations”, “characters”, and “items”. For each of these types, a specific function reads that type of object and translates it into a “create” Camelot instruction. For example, I included Listing 6.15, which shows the function that creates the items in Camelot.

```

1 def _create_items_from_problem(self, problem):
2     list_item = problem.find_objects_with_type(shared_variables.supported_types['item'])
3     json_parsed = parse_json('items')
4     while list_item:
5         item = list_item.pop(0)
6         itm = ''
7         for i in json_parsed['items']:
8             if item.name.lower() == i.lower():
9                 itm = i
10                break
11            if itm == '':
12                raise Exception('item not found in camelot')
13            self._camelot_action.action('CreateItem', [item.name, itm], self._wait_for_actions)

```

Listing 6.15: Function that translates all the items listed in the PDDL problem into “create” Camelot instructions.

This function is called in the initialization of the world state component, and it is responsible for creating all the items listed in the PDDL problem. It first reads the list of objects whose type is `item` and parses a JSON file that contains the list of all items supported by Camelot. Then, it starts a `while` loop to iterate over all the items. For each item, it checks if Camelot supports the item. If the item is supported, the function creates the corresponding `CreateItem` Camelot instruction with the item’s name and the Camelot item type. Finally, it sends the instruction to Camelot via the Camelot action component.

The JSON file containing the list of all items supported by Camelot is not the only JSON file used by the world state component to validate the elements listed in the PDDL problem. In fact, I have defined a set composed of four JSON files that are used to validate the Camelot instructions before being sent to Camelot:

- `items.json`: it contains a list of all items supported by Camelot.
- `characterlist.json`: it contains a list of all characters supported by Camelot with related body type, haircut, and outfit.

- **places.json**: it contains a list of all the places (defined with the `position` type in PDDL) supported by Camelot. Each place comprises a list of inner locations (defined with the `location` type in PDDL) that indicates the sub-locations that form the place. For example, the `AlchemyShop` place is composed of eleven inner locations (e.g., `AlchemyShop.Door` and `AlchemyShop.Chest`).
- **Actionlist.json**: it contains a list of all actions supported by Camelot. Each action contains the Camelot instruction that should be executed and the arguments that the instruction requires.

The solution to the second problem is based on a JSON file which is shown in Listing 6.14. In practice, this file is called `pddl_predicates_to_camelot.json`. This file is used by the function whose pseudocode is shown in Figure 6.3, which transforms the predicates listed in the initial state of the PDDL problem into Camelot instructions. This function iterates over all the predicates listed in the

```

Data: is ← PDDL initial state
Result: is translated into Camelot Instructions
json ← parse(pddl_predicates_to_camelot.json);
for relation ∈ is do
  | if relation.predicate ∈ json then
  | | for i ∈ json[relation.predicate]['declaration'] do
  | | | iargs ← substitute(i['action_args'], relation.entities);
  | | | send_camelot_instruction(i, iargs);
  | | end
  | end
end

```

Figure 6.3: Algorithm that translates all the items listed in the PDDL problem initial state into Camelot instructions.

initial state of the PDDL problem, and for each predicate, it checks if it is listed in the JSON file with name `pddl_predicates_to_camelot.json`. If it is not listed, it will skip it. Otherwise, it handles the `declaration` part of the JSON entry, which contains the Camelot instructions that should be executed to create the item in the world. It does that by substituting the parameters of the Camelot instruction with the parameters of the PDDL predicate. We can see from Listing 6.14, that a parameter in the JSON file is represented by a string that starts with a dollar sign, followed by the param string with a number, and that ends with another dollar sign (e.g., `$param1$`). The `substitute` function replaces such parameters with the corresponding entity in the PDDL relation.

6.1.6.2 Messages to Camelot

The second translation is the one that happens when the EM or the wrapper needs to send a message to Camelot. During normal communication, the message shared from the wrapper to Camelot is a PDDL action that needs to be executed (as seen in Section 5.3.3).

6.1.6.2.1 Problem

The problem that I aim to solve with this type of translation is to translate the PDDL action into Camelot instructions. This problem is composed of two sub-problems. First, it needs to translate the PDDL messages into the action class of the PDDL data framework. These messages are composed of the name of the PDDL action that needs to be executed and the parameters that the action requires. It is necessary to translate this message into the PDDL data framework's action to check if the preconditions of that action are satisfied and to prepare for the updates to the world state that will happen after the action is executed. Listing 6.16 shows an example of such a message.

```
1 move-between-location(Bob, AlchemyShop, BlackSmith, AlchemyShop.Door, BlackSmith.Door)
```

Listing 6.16: Example of a PDDL message that needs to be executed.

This message says that the `move-between-location` action needs to be executed with the following parameters: `Bob`, `AlchemyShop`, `BlackSmith`, `AlchemyShop.Door`, and `BlackSmith.Door`. Let us look at the declaration of the PDDL action in Listing 6.17. We can see that the parameters of the action are: `?who` that indicates the entity of type `character` that performs the action, `?from` and `?to` indicate the entities of type `location` that specify the departure and arrival place, `?entryfrom` and `?entryto` indicate the entities of type `entrypoint` that specify the departure and arrival entry point.

```
1 (:action move-between-location
2  :parameters (?who - character ?from ?to - location ?entryfrom ?entryto - entrypoint)
3  :precondition (and (in ?who ?from)
4    (alive ?who)
5    (adjacent ?entryfrom ?entryto)
6    (at ?who ?entryfrom))
7  :effect (and (in ?who ?to)
8    (not (in ?who ?from))
9    (not (at ?who ?entryfrom))
10   (at ?who ?entryto)))
```

Listing 6.17: Declaration of the `move-between-location` PDDL action.

Thus, the first sub-problem I need to solve is associating the parameters of the PDDL message with the parameters of the PDDL action.

The second sub-problem is that I must translate the PDDL action represented with the PDDL data framework into Camelot instructions. For example, the `move-between-location` action needs to be translated into the Camelot instructions that will move the character from one location to another (e.g., the `Enter` Camelot action). The solution to this problem requires creating a mapping between each PDDL action and the Camelot instructions that will execute it.

6.1.6.2.2 Solution Design

The solution to the first sub-problem is to create an association between the parameters of the PDDL message and the parameters of the PDDL action. This association is made by ensuring that the message and the PDDL action declaration use the same order of parameters as one another. If we look at the example

in Listing 6.17, the first parameter of the action declaration is the character that needs to move, the second parameter the departure location, and so on. Comparing this to the message in Listing 6.16, we can see that the same order of parameters is followed. To check that each message was sent correctly, the wrapper checks if the parameters of the action are the same as the parameters of the message. It does this while transforming the message into a PDDL data framework action.

The solution to the second sub-problem requires creating a mapping between the PDDL action and the Camelot instruction(s) that will execute it. In Section 6.1.4, I discussed that all the Camelot instructions that the designer wants to allow the execution by the manager need to be listed in the PDDL action part of the domain. To add to this solution, I designed another solution that uses a JSON file containing a list of the PDDL actions that need a manual association created. I created this method for two reasons. First, I wanted to allow the creation of PDDL actions that are not Camelot's actions. This allows the experience designer to create actions that are not originally available in Camelot and that can be used to create more complex interactions. For example, the designer may want to create a PDDL action that picks up an object from the ground and gives it to the character. This action is not available in Camelot unless multiple Camelot actions are executed in sequence. However, by creating a PDDL action named `pick-up-and-give` and associating it with the Camelot instructions that will pick up the object and give it to the character, the designer can create this more complex sequence as a single action.

The second reason is that I wanted to allow some automation when executing an action. For example, when executed, the Camelot action `OpenFurniture` will open the corresponding furniture in the Camelot environment. However, if we previously enabled an icon showing a prompt to open the furniture, that icon will remain until an instruction to change or remove it is sent. This can be fixed by overriding the `OpenFurniture` action and adding a Camelot instruction that will change the text of the icon.

Listing 6.18 shows an example of a JSON entry for the action `move-between-location`. Similarly to the predicate file, each JSON entry of this file has the name of the PDDL action as its key. However, in this case, the value is a list containing the Camelot instructions that should be executed when the PDDL action is asked to be executed. These Camelot instructions are formatted as `action_name` and `action_args` where the `action_name` is the Camelot instruction that should be executed, and the `action_args` is the list of arguments that the Camelot instruction requires. These arguments need to have the same name as the arguments of the PDDL action to allow correct substitution during the translation process. More details of this process are provided later in this section.

```

1  {"move-between-location":{
2    "commands":[{
3      "action_name": "Enter",
4      "action_args": ["?who", "?entryto", "False"]
5    }]}

```

Listing 6.18: Example of an entry of the JSON file for converting PDDL actions into Camelot instructions.

6.1.6.2.3 Implementation

When the world state component receives a PDDL action message that needs to be applied, an algorithm is used to handle this message. The pseudocode of this algorithm is shown in Figure 6.4. First, it needs to translate this message into an

```

Data:  $m \leftarrow$  PDDL action message
Result:  $m$  applied to Camelot and world state
 $action \leftarrow$  create_action_from_incoming_message( $m$ );
if  $action$  is applicable then
    |  $i \leftarrow$  generate_instructions( $action$ );
    |  $result \leftarrow$  execute_instructions( $i$ );
end

```

Figure 6.4: Algorithm used to execute a PDDL action message.

action class of the PDDL data framework component. Then, it needs to check if the action is applicable by checking the preconditions of the action in the world state. The environment, at any time, has the most updated worldstate of the Camelot environment. As a result, it has the authority to stop a PDDL action from being executed if the preconditions are not met. If the action can be applied, the function creates the Camelot instructions needed for the execution in Camelot. The next step is to send these instructions to Camelot via the Camelot action component.

The implementation of this algorithm spans several functions. The `create_action_from_incoming_message` is the first function that is called, and the implementation is shown in Listing 6.19

```

1 def create_action_from_incoming_message(self, message):
2     message_parts = re.split(r"\(|\)|,", message)
3     action_name = message_parts[0]
4     action_definition = self.domain.find_action_with_name(action_name)
5     if action_definition is None:
6         return
7     parameters = {}
8     for i in range(len(action_definition.parameters)):
9         parameters[action_definition.parameters[i].name] = self.world_state.find_entity(name = ↔
10         ↔message_parts[i+1], type=action_definition.parameters[i].type)
11     return Action(action_definition, parameters=parameters)

```

Listing 6.19: Function that translates a PDDL action message into the action class of the PDDL data framework.

This function receives the PDDL action message and translates it into an action class of the PDDL data framework component. The first step is to split the message into parts using a regular expression matching the opening and closing parenthesis and the comma. Then, it needs to find the action definition in the PDDL domain. If the action definition is found, it must create a dictionary with the action's parameters. The parameters are created by matching the parameter's name with the entity's name in the world state. Finally, it creates the action class and returns it.

Once the action is created, the next step is to create the Camelot instructions that need to be executed in Camelot to apply the action. This is done by the `generate_instructions` function in Figure 6.4, which is shown in Listing 6.20 under a different name.

```

1 def generate_camelot_action_parameters_from_action(self, action: Action):
2     camelot_commands = []
3     if action.name not in self.json_actions_to_camelot.keys():
4         return None
5     command_data = self.json_actions_to_camelot.get(action.name).get("commands")
6     parameters = {k : v.name for (k,v) in action.parameters.items()}
7     if action.name.startswith("instantiate_"):
8         parameters['?name'] = parameters['?obj']
9         parameters['?obj'] = ''.join(i for i in parameters['?name'] if not i.isdigit())
10    for command in command_data:
11        command_dict = {"action_name": command["action_name"], "action_args": [], "wait": str2bool(↵
12            ↵command["wait"])}
13        for item in command["action_args"]:
14            command_dict["action_args"].append(replace_all(item, parameters))
15    camelot_commands.append(command_dict)
16    return camelot_commands

```

Listing 6.20: Function that translates an action into Camelot instructions.

This function receives an action and returns the Camelot instructions that must be executed to apply the action. The PDDL action declaration and the Camelot instructions are necessary to execute the translation between an action and the corresponding Camelot instructions. This translation uses the JSON file named `pddl_actions_to_camelot.json`. This function looks into this JSON file and creates the commands for executing the Camelot instructions with the correct parameters.

To map the correct parameters into the Camelot instructions, the section with the name `action_args` of the corresponding entry in the JSON file needs to be a list of strings that contain the name of the parameter as it appears in the definition of the action in the PDDL domain file. For example, let us consider the `move-between-location` action whose PDDL definition is shown in Listing 6.17 and the entry of the JSON file shown in Listing 6.18. The `action_args` section of the JSON file contains the string `?who` and `?entryto`. The same strings are used in the PDDL declaration of the action in the parameters section. Therefore, since it previously created the action with the correct parameters, the algorithm can replace the strings `?who` and `?entryto` with the actual names of the entities.

As another example, we can imagine adding another supported action by the Camelot wrapper. Suppose the game designer wants to have an action called `fight` (that is currently outside the available actions that Camelot allows). In that case, they can do so by including this action into the PDDL domain file and adding an entry into the corresponding JSON file with all the Camelot-specific instructions that should happen when this PDDL action is performed (e.g., ready a weapon and attack the targeted enemy).

The next step in the algorithm in Figure 6.4 is to execute the Camelot instructions generated by the previous function. This is done by asking the Camelot action component to execute the instructions. This process is presented in Section 6.1.7.

6.1.6.3 Messages from Camelot

The third translation is the one that occurs when Camelot shares updates of what is happening in the environment, and the Camelot wrapper needs to translate those updates into the world state. This process is important for the Camelot wrapper to

be able to keep track of the world state and to be able to give the EM the correct information about the state of the Camelot environment.

6.1.6.3.1 Problem

The problem I aim to solve with this translation is to interpret the messages Camelot sends to the Camelot wrapper and update the world state accordingly. Camelot shares updates about the state of the world using messages that are formatted using a particular methodology similar to a logging system. I discussed the types of these messages in Section 6.1.3, and we can see an example of a message in Listing 6.7. The Camelot world state component receives the messages related to the world's state and interprets them to update the world state. These messages are of two types: location and *success*. Also in this case, the problem of interpreting the messages and transforming them into world-state updates can be divided into two subproblems. The first sub-problem is related to the interpretation of the location messages. In fact, in Camelot, a player can move freely within the room that they are in. This means that every time the player moves, Camelot sends a message with the player's new location. The problem is to interpret these messages and update the position of the player in the world state accordingly.

The second sub-problem is related to the interpretation of the success messages. These messages are sent by Camelot when an action previously sent by the Camelot wrapper is successfully executed or when the Camelot wrapper previously enabled the player to interact with certain parts of the game, and the player performs that interaction. The problem is to interpret these messages to understand which action was executed and update the world state accordingly with the effects of that action. For example, suppose the Camelot wrapper sends a `OpenFurniture` action to Camelot and responds with a success message. In that case, the world state component will need to update the state of the furniture that was opened to be open.

6.1.6.3.2 Solution Design

The solution that handles this translation differs based on the cause that triggered the update. In the case of a location message, the objective of the Camelot world state component is to update the location of the entities that are in the message. Consider the example of a location message shown in Listing 6.21.

```
1 input arrived bob position alchemyshop.Door
```

Listing 6.21: An example of a message that comes from Camelot.

This message indicates that in Camelot, an `input` happened where the entity `bob` finished the action `arrived` at the position `Chest` within the location `alchemyshop`. This requires handling two predicates in the PDDL domain file: `in` and `at`. The `in` predicate is used to specify that an entity is inside a location. For example, the relation `in bob alchemyshop` means that `bob` is inside the location `alchemyshop`. The `at` predicate is used to specify that an entity is at a position within that location. For example, the relation `at bob alchemyshop.Chest` means that `bob` is

at the position `Chest` within the `alchemyshop` location. I need these two predicates because some actions may require two entities that are part of that action to be in the same location but not necessarily in the same position. Another location message that Camelot can send is the one shown in Listing 6.22.

```
1 input exited bob position Alchemyshop.Door
```

Listing 6.22: Example of another location message that Camelot shares.

This message indicates that in Camelot, an `input` happened where the entity `bob` finished the action `exited` at the `Alchemyshop.Door` position within the `alchemyshop` location.

The solution to this problem requires an algorithm that evaluates the three possible cases that can happen when a location message is received. First, when the entity changed position in the current location. This requires modifying the `at` relation in the current world state. Second, when the entity moved to a different location. This requires modifying the `in` and `at` relation in the current world state. Third, when the entity exited a location. This requires modifying the `at` relation in the current world state.

The second cause that can trigger an update is a success message. In this case, Camelot aims to communicate that an action was successfully executed. Let us consider the example of a success message shown in Listing 6.23.

```
1 succeeded WalkTo(bob, alchemyshop.Door)
```

Listing 6.23: Example of a success message that Camelot shares.

This message indicates that Camelot successfully executed (indicated with the string `succeeded`) the action `WalkTo` where the entity `bob` walked to the position `Door` within the `alchemyshop` location. This type of message can be received in two cases.

The first case is when the Camelot wrapper asks for the execution after receiving a message from the experience manager. In this case, as explained previously, the Camelot wrapper already knows which action must be applied to the world state since it previously sent that action for execution. Therefore, the Camelot world state component must only update the world state with the changes specified in the action that was executed.

The second case is when the player executes an action. This type of action happens when the Camelot wrapper previously enabled the player to interact with certain parts of the game. For example, the Camelot wrapper may enable the player to interact with the chests in a location of the game. In this case, when the player clicks on the chest to open it, the Camelot wrapper will receive a message from the game that indicates that the player has opened the chest successfully. I will explain how the handling of the input message works in Section 6.1.9. When the Camelot wrapper receives a success message where the player executed an action, the process to update the world state connects to how predicates are defined. Let us look at Listing 6.24, where the entry for the predicate `adjacent` is defined.

```
1 {"adjacent": {
2   "declaration": [{
```

```

4     "action_name" : "EnableIcon",
      "action_args": ["Exit", "Exit", "$param1$", "Exit from $param1$", "TRUE"]
5   }},
6   "input": {
      "door": "input Exit $param1$",
      "end": "input arrived $param2$ position $param1$"
7   },
8   "response": [{
10    "action_name" : "WalkTo",
11    "action_args": ["$param2$", "$param1$"]
12  }, {
13    "action_name" : "FadeOut",
14    "action_args": []
15  }, {
16    "action_name" : "SetPosition",
17    "action_args": ["$param2$", "$param3$"]
18  }, {
19    "action_name" : "SetCameraFocus",
20    "action_args": ["$param2$"]
21  }, {
22    "action_name" : "FadeIn",
23    "action_args": []
24  }
25 ]}],}

```

Listing 6.24: Example of an entry of the JSON file that maps predicates to Camelot instructions.

The `input` key contains the instruction that Camelot will send once the player clicks on the door. The `response` key contains the Camelot instructions that should be executed when the player clicks on the door. In this case, the list contains five Camelot instructions: `WalkTo`, `FadeOut`, `SetPosition`, `SetCameraFocus`, and `FadeIn`. These instructions are executed to move the player from the current location to the new location and animate this process. As a result, when the player clicks the door, the Camelot wrapper will receive the input message specified in the `input` key and execute the actions listed in the `response` key. For each of these actions, Camelot will send a success message that will be handled as explained previously.

6.1.6.3.3 Implementation

The solution to the problem of translating a location message into a PDDL state update uses the algorithm shown in Figure 6.5. The first step of the algorithm is to split the message into its parts using the space character as a separator. Then, the algorithm checks if the message is an input message. If it is, there is a different plan of action if the message is related to the arrival of an entity to a position or the exit of an entity from a position. In case of an exit, it finds the entities that are part of the message and the related `at` relation to those entities. Then, it removes those relations since the entity is no longer at that position. In the case of arrival, the entity must be in one position only, so we need to be sure that the entity has no other relations that say it is at a different position within the same location. To do so, the algorithm finds the entities that are part of the message and the relations that involve those entities. Then, it removes all the relations that involve those entities and use the `at` predicate. After that, the algorithm adds a new relation with the `at` predicate and the updated position. The algorithm also checks if the

```

Data:  $m$  = location message
Result: Updated world state
message_parts  $\leftarrow$  split  $m$  by ' ';
if message_parts[0] is 'input' then
  if message_parts[1] is "arrived" and message_parts[3] is "position"
  then
    entities  $\leftarrow$  find entities in message_parts;
    relations_at  $\leftarrow$  find at relations with entities;
    for relation  $\in$  relations_at do
      | remove relation from world state;
    end
    add new relation with updated position for entities;
    relations_in  $\leftarrow$  find in relations with entities;
    for relation  $\in$  relations_in do
      | if entities changed relation.location then
        | remove relation from world state;
        | add new relation with updated location for entities;
      | end
    end
  end
  if message_parts[1] is "exited" and message_parts[3] is "position"
  then
    entities  $\leftarrow$  find entities in message_parts;
    relation_at  $\leftarrow$  find at relations with entities;
    remove relation_at from world state;
  end
end

```

Figure 6.5: Algorithm used to update the world state using a location message.

entity changed location. If it did, the algorithm removes the relation that says the entity is **in** the old location and adds a new relation with the new location.

To handle the success messages, I have implemented the algorithm shown in Figure 6.6. The first step of the algorithm is to split the message into its parts using the space character as a separator. Then, the algorithm checks if the message is a success message and splits it into its parts. Then, it finds the action that is related to the message. If the action is in the list of actions previously sent to Camelot, it removes the action from the list and updates the world state with the effects of that action. In the other case, it means that the player executed the action. So, it finds the entities that are part of the message and populates the action with those entities. Then, it updates the world state with the effects of the action.

```

Data:  $m$  = success message
Result: Updated world state
message_parts  $\leftarrow$  split  $m$  by ' ';
if message_parts[0] is 'succeeded' then
    message_parts  $\leftarrow$  split  $m$  by (' ', ',', '(', ')');
    action  $\leftarrow$  find action with name message_parts[1];
    if action in previous sent actions then
        remove action from previous sent actions;
        update world state with action effects;
    else
        entities  $\leftarrow$  find entities in message_parts;
        PDDL_action  $\leftarrow$  populate action using entities;
        update world state with PDDL_action effects;
    end
end

```

Figure 6.6: Algorithm used to update the world state using a success message.

6.1.7 Camelot Action

The Camelot action component is responsible for creating and preprocessing Camelot instructions before sending them to Camelot. Whenever any component of the Camelot wrapper needs to send an instruction to Camelot, it passes through to the Camelot action component. The Camelot action component receives the name of the action that needs to be executed and the entities that are part of the action. Then, it creates the Camelot instruction performing all the necessary checks to avoid any error from Camelot. Once the Camelot instructions are sent to Camelot, the Camelot action component can wait for the result of the action if needed.

6.1.7.1 Implementation

The implementation of the Camelot action component is located in the `camelot_action.py` file. The file hosts the `CamelotAction` class with all the methods for creating the Camelot instructions. When a Camelot instruction needs to be created, the `action` method shown in Listing 6.25 is called.

```

1 def action(self, action_name, parameters = [], wait=True):
2     if not any(d['name'] == action_name for d in self.json_actionlist):
3         raise KeyError()
4     action_data = [d for d in self.json_actionlist if d['name'] == action_name][0]
5     if len(parameters) > 0:
6         self._check_action_parameters(action_data, parameters)
7     command = self._generate_camelot_string(action_name, parameters, action_data)
8     self.send_camelot_instruction('start ' + command)
9     if wait:
10        return self.check_for_success(command, action_name)
11    else:
12        return True

```

Listing 6.25: Function that creates and sends Camelot instructions.

This function receives three parameters: the name of the Camelot action that needs to be executed, the entities that are part of the action, and a boolean that indicates if the function should wait for the result of the action. The first step of the function is to check if the Camelot action is in the JSON dictionary of Camelot actions. Then, it gets the data of the declaration of the Camelot action from the dictionary to check if all the required parameters are present. If all the checks are successful, the function generates the Camelot instruction. Then, it sends the instruction to Camelot with the `start` prefix. Finally, if the function needs to wait for the result of the action, it calls the `check_for_success` function to wait for the result.

6.1.8 Error Manager

The error manager is the component that handles any errors from Camelot. When Camelot generates an error, the error manager performs a first analysis of the error and sends a report to the API connector to send it directly to the EM. The analysis is quite simple, and it only tries to understand which actions generate the error and which entities are part of the action. The Camelot wrapper tries to avoid any possible error in Camelot by performing all the necessary checks before sending an instruction to Camelot. If an error happens after the Camelot wrapper has sent an instruction to Camelot, it means that the error is not related to the Camelot wrapper but to Camelot itself.

6.1.9 Game Controller

The game controller hosts the main loop of the Camelot Wrapper, and its job is to control the game in the environment from start to finish. When an environment is started, the game controller starts all the other components of the Camelot Wrapper and handles the startup process. The startup process is composed of two phases: the initialization of the communication with EM-Glue and the initialization of the Camelot environment. The first phase of initialization is used to set up the communication with EM-Glue. To do so, the game controller needs to follow the handshake protocol described in Section 5.3.2 and send all the data needed by EM-Glue to start the game. Once phase two of the protocol is executed, then the game controller can start the second phase of the initialization by starting to set up the Camelot environment. This is done to create the updated Camelot world state with all the entities and relations that are part of the initial state of the game. During this phase, the Camelot world state component initializes the Camelot environment as we have seen in Section 6.1.6. Once the Camelot world state has finished the initialization and Camelot is ready to start the experience, then the game controller can start phases three and four of the handshake protocol. Once the handshake protocol is finished, the game controller can show the initial menu in Camelot and wait for the player to start the game.

Once the startup process is finished and the player has started the game, the game controller hosts the main loop of the Camelot Wrapper, where it executes the methods to handle different aspects of the game. The main loop executes the handlers for receiving messages from EM-Glue and Camelot, and it also executes

the handlers for the different aspects of the game such as the scene controller (Section 6.1.11).

6.1.9.1 Implementation

The code of the game controller is located in the file named `game_controller.py`. This file hosts the `GameController` class that is the main class of the Camelot Wrapper. During the initial phases of the execution, the game controller starts all the other components (described in the previous sections) of the Camelot Wrapper.

Once the initialization of all the other component finished, the communication with EM-Glue can start using the method `start_platform_communication` that is shown in Listing 6.26.

```

1 def start_platform_communication(self):
2     self._platform_communication.start()
3     message = self._platform_communication.communication_protocol_phase_messages['PHASE_2'] + '↔'
4     result = self._platform_communication.send_message(message, initialization=True)
5     if result['text'] == self._platform_communication.communication_protocol_phase_messages['↔'
6     return True
7     else:
8         raise Exception("Platform communication failed")

```

Listing 6.26: Function that starts the communication with EM-Glue.

The function starts the communication with EM-Glue by calling the `start` method of the `PlatformCommunication` class. Then, it prepares and sends the message that indicates that the Camelot Wrapper is ready to start the game. Once the message is sent, the function waits for the response from EM-Glue. If the response is correct, then the function returns `True` to indicate that the communication with EM-Glue is ready for the next phase.

Once this function finishes, the next step is to call the function that is shown in Listing 6.27.

```

1 def start_game(self, game_loop = True):
2     self._initialize()
3     initial_state = CamelotWorldState(self._domain, self._problem, wait_for_actions= game_loop)
4     initial_state.create_camelot_env_from_problem()
5     initial_state.check_domain_actions_available_to_use()
6     self._platform_communication_phase_3_4(initial_state.domain, initial_state.world_state)
7     self._player = initial_state.find_player(self._problem)
8     self._create_ingame_actions(game_loop)
9     self._camelot_action.action("ShowMenu", wait=game_loop)
10    while game_loop:
11        received = self.camelot_input_multiplex.get_input_message()
12        if received == 'input Selected Start':
13            self._camelot_action.action("HideMenu")
14            self._camelot_action.action('EnableInput')
15            self._main_game_controller(game_loop)

```

Listing 6.27: Function that starts the setup of the game in Camelot.

The function starts the setup of the game in Camelot by calling the `_initialize` function, which is used to initialize the other components that the Camelot Wrapper needs to start the game. Then, the function creates the initial state of the

game by creating an object of the `CamelotWorldState` class. The object is created by passing the domain and problem files of the game. Then, it calls the `create_camelot_env_from_problem` method of the `CamelotWorldState` class to create the initial state of the game in Camelot as described in Section 6.1.6. Once the initial state is created and sent to Camelot, then it continues the handshake protocol with EM-Glue by executing phase three and four of the protocol with the function shown in Listing 6.28. Once the handshake phase has finished, then it shows the menu in the Camelot environment and waits for the player to start the game. Once the player starts the game, then the function starts the main loop of the Camelot Wrapper by calling the `_main_game_controller` function.

```

1 def _platform_communication_phase_3_4(self, domain: Domain, world_state: WorldState):
2     message_text = {
3         "text" : self._platform_communication.communication_protocol_phase_messages['PHASE_3'] +
4             ↳message_6'],
5         "domain" : domain.to_PDDL(),
6         "problem" : world_state.to_PDDL(),
7         "additional_data" : self._scene_controller.get_scene_message()
8     }
9     result = self._platform_communication.send_message(message_text, initialization=True)
10    if result['text'] == self._platform_communication.communication_protocol_phase_messages['
11        ↳PHASE_4'] + 'message_9']:
12        self._platform_communication.send_message_link = result['add_message_url'].replace('/', '')
13        self._platform_communication.receive_message_link = result['get_message_url'].replace('/', '')
14        ↳)
15    return True
16    else:
17        raise Exception("Platform communication failed")

```

Listing 6.28: Function that executes phase three and four of the handshake protocol in the Camelot Wrapper.

To execute phases three and four of the handshake protocol, the function prepares the message containing the domain and problem updated after the Camelot initialization, and the additional data parameter with the details of the scenes that will be discussed in Section 6.1.11. Then, it sends the message to EM-Glue and waits for the response. If the response is correct, then the function saves the links for normal communication in the `PlatformIOCommunication` class and returns `True` to indicate that the handshake protocol is finished.

The `_main_game_controller` function hosts the main loop of the Camelot Wrapper. In this loop there are periodic calls to functions that handle the different aspects of the game such as inputs, success messages, location messages, incoming messages from EM-Glue, error messages, and the scene execution. I described most of these handling functions in the previous sections or I will discuss them in the next sections. For this reason, I will not discuss them in detail here. However, I want to discuss the input handling function that is shown in Listing 6.29. The peculiarity of this function is that, other than handling input messages, it also manages the menu that is shown in Camelot once the player presses the `Esc` button. It also hosts part of the conversation handling that will be discussed in Section 6.1.10.

```

1 def _input_handler(self) -> bool:
2     try:
3         received = str(self.camelot_input_multiplex.get_input_message(no_wait=True))
4         if received in self.input_dict.keys():

```

```

        for item in self.input_dict[received]:
            action_name = item['action_name']
            action_parameters = item['action_parameters']
            wait = item['wait']
            self._camelot_action.action(action_name, action_parameters, wait=wait)
    10 elif received == "input Key Pause":
        if not self._menu_showing:
    12     self._camelot_action.action("ShowMenu", [], True)
            self._camelot_action.action("SetTitle", ["Pause Menu"], True)
    14     self._menu_showing = True
        elif received in ("input Selected Resume", "input Close Menu"):
    16     self._camelot_action.action("HideMenu", [], True)
            if not self.conversation_active:
    18     self._camelot_action.action("EnableInput", [], True)
                self._menu_showing = False
    20     elif received in ("input Selected Quit", "input Quit"):
            self.camelot_input_multiplex.stop()
    22     elif received.startswith("input Selected"):
            selection = received.removeprefix("input Selected ")
    24     if selection.isdigit():
                self._conversation_controller.continue_conversation_with_choice(int(selection))
    26     elif selection == 'next':
                self._conversation_controller.continue_conversation()
    28     elif selection == 'end':
                self._conversation_controller.end_conversation()
    30     self.conversation_active = False
        except queue.Empty:
    32     return False
    return True

```

Listing 6.29: Function that handles the input messages from Camelot.

The function first attempts to retrieve an input message from Camelot. If no message is found, it returns `False` to indicate that no input message was received. If an input message is found, the function checks whether the message is included in the `input_dict` dictionary. This dictionary specifies the actions that should be taken when a particular input message is received, which is necessary for handling low-level interactions required to play the game (e.g., interacting with chests). If the message is found in the dictionary, the function performs the associated actions. If the message is not found in the dictionary, the function checks whether the message is a menu message. If it is, the function displays or hides the menu in Camelot. If it is not a menu message, the function checks whether the message is a conversation message. If it is, the function either continues the conversation or ends it based on the conversation's current state. I will cover additional details on how the conversation is controlled in Section 6.1.10.

6.1.10 Conversation Controller

Conversation is one of the high level dynamics that needs some automation to be controlled in Camelot. A conversation should be controlled by using a PDDL action. When the wrapper needs to execute this action, it should automatically set up and handle the conversation with the player, and report to the experience manager only the information that it needs. To achieve this, I needed to solve two problems. First, I needed to find a way to send the correct Camelot instructions to Camelot to set up and handle the conversation. Second, I needed to find a way to extract the information that the experience manager needs from the conversation.

To solve these two problems, I created a conversation controller that is responsible for controlling the conversation between the player and the characters in the game. It is responsible for displaying the conversation text and the choices that the player can make. It also handles the player's choice and sends the appropriate message to EM-Glue.

An environment might allow multiple conversations to happen during an experience. Each conversation is based on a *Yarn Spinner* [46] file that contains the text of the conversation and the choices that the player can make. *Yarn Spinner* is a tool that is used to create interactive conversations for a game using a simple, screenplay-like format. These conversations are saved in `.yarn` files that can be loaded into a game and run. In *Yarn Spinner*, a dialogue is organized into nodes, each of which contains a collection of headers and a body. At a minimum, every node has a title header, which is the node's name, and a body that contains the Yarn script for the dialogue.

When creating dialogue in *Yarn Spinner*, each line of text that is written in a node represents a distinct line of dialogue. When a node is run, it sends each line to the game in sequence. To allow the player to choose from multiple dialogue options, it is possible to use the `->` symbol to mark an option. This displays potential lines of dialogue to the player and allows them to select one. In Listing 6.30, an example of a *Yarn Spinner* node is shown.

```

1 title: Start
2 ----
3 Companion: Hi there! Which do you prefer, coffee or tea?
4 -> Player: I love coffee! It gives me the boost I need to start my day.
   Companion: Okay, let's go take some coffee.
6 -> Player: I prefer tea. It's soothing and helps me relax.
   Companion: Cool, we'll go take a cup of tea.
8 ====

```

Listing 6.30: Example of a *Yarn Spinner* node with options.

In *Yarn Spinner*, there are some special commands that can be used to control the flow of the conversation. For example, the `<<jump node name>>` command allows jumping the conversation to a different node. We can also use functions in a conversation. A function is a block of code that allows one to send data from a conversation back into the game. For example, in the conversations that I wrote for the experience that I created to replicate the *PaSSAGE* experience manager, I used a function to update the player model every time a meaningful choice was made. This function sends an `update_player_model` message to the experience manager via EM-Glue. I discuss this function in more detail in the implementation section. A *Yarn Spinner* file representing one conversation that I used in the Camelot wrapper can be found in Appendix A.

6.1.10.1 Implementation

The conversation controller is implemented in the class named: `ConversationController`. Each conversation is represented by an instance of the `Conversation` class. To start discussing about the implementation of the conversation controller, I will first describe the implementation of the `Conversation` class.

Conversation Class The `Conversation` class is responsible for loading the *Yarn Spinner* file that contains the conversation and for keeping track of the conversation while is running. I was able to work with *Yarn Spinner* in python using a library called `YarnRunner-Python` [146]. This library uses the compiled files that are generated by *Yarn Spinner* after a `.yarn` file is compiled. To provide the library with the compiled files, in the `__init__` method of the `Conversation` class, I compile the conversations that are located in the `narrative` folder in the Camelot wrapper. Listing 6.31 shows part of the implementation of the `__init__` method of the `Conversation` class.

```

1 def __init__(self, name : str, filename : str) -> None:
2     self.name = name
3     command = "ysc compile "+filename+" -o output -n "+ filename.replace(".yarn", ".yarnc") + " ←
4     ↪-t "+ filename.replace(".yarn", ".csv")
5     subprocess.run(shlex.split(command), stdout=subprocess.PIPE)
6     [...]

```

Listing 6.31: Part of the implementation of the `__init__` method of the `Conversation` class.

When a new object of the `Conversation` class is created, the `__init__` method is called. It takes as input the name of the conversation and the name of the *Yarn Spinner* file that contains the conversation. Then, it compiles the *Yarn Spinner* file using the `ysc compile` command that is run using the `subprocess` library. The compiling process produces two files: a `.yarnc` file that contains the compiled conversation and a `.csv` file that contains the compiled conversation in a format that is easier to read.

These files are loaded in the `YarnRunner` library once the `prepare` method of the `Conversation` class is called. Listing 6.32 shows the implementation of the `prepare` method.

```

1 def prepare(self, player_name : str, npc_name : str):
2     with open(os.path.join(os.path.dirname(__file__), 'narrative/output/'+self.name+'.yarnc'), 'rb') ←
3         ↪ as story_f:
4         with open(os.path.join(os.path.dirname(__file__), 'narrative/output/'+self.name+'.csv'), 'r') ←
5             ↪ as strings_f:
6             self.runner = YarnRunner(story_f, strings_f, autostart=False)
7     def update_player_model(fighter, method_actor, storyteller, tactician, power_gamer):
8         logging.info("Updating player model with parameters: {}, {}, {}, {}, {}".format(fighter, ←
9             ↪method_actor, storyteller, tactician, power_gamer))
10        message = ('update_player_model', str((fighter, method_actor, storyteller, tactician, ←
11            ↪power_gamer)))
12        self._platform_communication.send_message(message)
13        self.runner.add_command_handler("update_player_model", update_player_model)
14    self.runner.resume()
15    self.player_name = player_name
16    self.npc_name = npc_name
17    self._prepared = True

```

Listing 6.32: Implementation of the `prepare` method used for conversations.

The `prepare` method takes as input the name of the player and the name of the NPC. It then loads the compiled files that were generated by the `ysc compile` command into the `YarnRunner` library. It creates a function called `update_player_model` that is used to update the player model. This function is called directly in the `.yarn` conversations when the player makes a choice that needs to update the

player model in the experience manger. When this function is called in the conversation, a message of type `update_player_model` (as described in Section 5.3.3) is sent to the experience manager containing the parameters that need to be updated in the player model. More details about the player model feature are provided in Section 6.2.1.3. In addition, I discuss limitations and future work related to this feature respectively in Sections 7.3 and 7.4. The `prepare` method also calls the `resume` method of the `YarnRunner` library to start the conversation. Finally, it sets the `_prepared` attribute to `True` to indicate that the conversation is ready to be run.

Once the conversation is prepared, it can be run using a set of methods of the `Conversation` class that I describe while discussing the implementation of the `ConversationController` class. However, I want to mention another method of the `Conversation` class that is used to create the line of dialogue in a way that Camelot can interpret. This method is shown in Listing 6.33.

```

1 def get_camelot_setdialog_string(self) -> list:
2     return_list = []
3     line_of_dialog = self.run_one_line_conversation()
4     return_list.append(self._prepare_line(line_of_dialog))
5     if self.has_line():
6         return_list.append("{}|{} ".format('next', "Next line"))
7         return return_list
8     elif self.is_finished():
9         return_list.append("{}|{} ".format('end', "End Dialog"))
10        return return_list
11    else:
12        for choice in self.get_choices():
13            choice_string = " "
14            text = self._prepare_line(choice['text'])
15            choice_string += "{}|{} ".format(choice['index'], text)
16        return_list.append(choice_string)
17    return return_list

```

Listing 6.33: Implementation of the method to prepare the messages for the conversation in Camelot.

This method is called once the conversation has started and it is used to create the line of dialogue that is sent to Camelot. The method first calls the `run_one_line_conversation` method to run the conversation until the next line of dialogue is reached. It adds this line to the list that will be returned by the method. Then, it checks whether the conversation is finished, there are more lines of dialogue, or there are choices that need to be presented to the player. If the conversation is finished, it adds a button to the line of dialogue that allows the player to end the conversation. If there are more lines of dialogue, it adds a button to the line of dialogue that allows the player to continue the conversation. If there are choices, it adds a button for each choice that allows the player to make a choice. These options are what Camelot will return to the Camelot wrapper once the player makes a choice. In fact, in the final part of Listing 6.29, we can recall that when the player makes a choice, the wrapper receives an `input Selected` message with concatenated one of the following strings: a digit that indicates which choice the player made in the dialogue, `next` that indicates that the player wants to continue the conversation, or `end` that indicates that the player wants to end the conversation.

Conversation Controller Class When the conversation controller is created, it creates an instance of the `Conversation` class for each conversation that is located in the `narrative` folder. When a `start_conversation` message is received from the experience manager, the handler of the message calls the method of the `ConversationController` class that is used to start the conversation. Listing 6.34 shows the implementation of this method.

```

1 def start_camelot_conversation(self, conversation_name : str, player_name: str, npc_name : str):
2     self.conversations[conversation_name].prepare(player_name, npc_name)
3     self._camelot_action.action("SetLeft", [player_name], True)
4     self._camelot_action.action("SetRight", [npc_name], True)
5     self._prepare_and_send_camelot_setdialog_command(conversation_name)
6     self._camelot_action.action("ShowDialog", [], True)

```

Listing 6.34: Implementation of the method used to start a conversation in Camelot.

The method takes as input the name of the conversation, the name of the player and the name of the NPC. It then calls the `prepare` method of the `Conversation` class to prepare the conversation. Then, it starts setting up a conversation in Camelot by sending a `SetLeft` message to Camelot to set the player on the left of the conversation screen and `SetRight` message to set the NPC on the right of the conversation screen. After that, it calls a method to start preparing the conversation and send a `SetDialog` message to Camelot to start the conversation. Finally, it sends a `ShowDialog` message to Camelot to show the conversation on the screen.

The `_prepare_and_send_camelot_setdialog_command` calls a method of the `Conversation` class to get the line of dialogue that is sent to Camelot. Then, it sends a `SetDialog` message to Camelot to set the line of dialogue on the screen.

6.1.11 Scene Controller

The scene controller is responsible for managing cutscenes in the environment. A cutscene is an author-defined set of actions that are executed when a `start_scene` message needs to be executed. A cutscene is useful because when the experience manager wants to execute a predefined set of actions that are part of the narrative, it can send a `start_scene` message to the scene controller and the scene controller will execute the actions that are part of the scene. Similarly to conversations, scenes are a high level dynamic whose execution is automated. I needed a scene controller to allow the experience manager to control the execution of a sequence of actions with just one message. These actions can be formed by both PDDL actions and Camelot instructions and they are listed in a JSON file. The JSON file has the following structure of keys:

- **name**: it specifies the name of the scene as a string. This is the name that will be sent to the experience manager at the beginning of the experience.
- **metadata**: it specifies the metadata of the scene as a dictionary. This metadata is used by the author to augment the scene with additional information. In the case of *PaSSAGE*, this metadata is used to specify a set of weights that

indicates the type of player that is more likely to like the scene once executed. These metadata are sent to the experience manager at the beginning of the experience.

- **preconditions:** it specifies the preconditions of the scene as a list of strings. These preconditions are used to check whether the scene can be executed or not and they follow the PDDL specification. These preconditions are sent to the experience manager at the beginning of the experience.
- **instructions:** it specifies the instructions of the scene as a list of dictionaries. These instructions are used to execute the scene and they can be either PDDL actions or Camelot instructions. Each dictionary is composed of two keys: **type** and **commands**. The **type** key specifies the type of the instruction as a string and it can be either PDDL or Camelot. The **commands** key is a list of strings representing the PDDL actions or Camelot instructions (based which is the **type**) that needs to be sent to Camelot during the cutscene execution. These instructions are an optional argument that can be sent to the experience manager at the beginning of the experience if the author requires. In the case of *PaSSAGE*, these instructions are not sent.

6.1.11.1 Implementation

The implementation of the scene controller is similar to the implementation of the conversation controller. There are two classes: the `SceneController` class and the `Scene` class. The class named `SceneController` is responsible for managing the scenes in the game. The `Scene` class is used to describe a scene in the game.

Scene Class The `Scene` class hosts all the information of a scene. When an instance of the `Scene` class is created by the scene controller, it is initialized with the name of the scene, the metadata of the scene, the preconditions of the scene and the instructions of the scene. It then hosts all the methods needed to execute the scene. The most important method is the one that generates the instructions to send to Camelot, that is shown in Listing 6.35.

```

1 def get_generator_instruction(self):
2     if self.executed:
3         return None
4     for instruction in self._instructions:
5         for command in instruction["commands"]:
6             return_instruction = (instruction["type"], command)
7             self._instructions_sent.append(return_instruction)
8             yield return_instruction

```

Listing 6.35: Implementation of the method that generates the instructions that need to be sent to Camelot when a scene is executing.

I developed this method as a Python generator. A Python generator is a special type of iterator that generates values on the fly using the `yield` keyword. The `yield` keyword pauses the function execution and returns the current value. When the function is called again, it resumes execution from where it left off and generates the next value. The `get_generator_instruction` method is used by the

scene controller to get the instructions of the scene. The method iterates over the instructions of the scene and returns the first instruction that has not been executed yet. The method returns the instruction as a tuple of two elements: the type of the instruction and the command of the instruction.

Scene Controller Class The `SceneController` class works similarly to the conversation controller class. Its duty is to create an instance of the `Scene` class for each scene in the game and to manage the execution of the scenes. To create instances of the `Scene` class, the scene controller gets all the `.json` files in the `scenes` folder and it creates an instance of the `Scene` class for each of them passing all the elements contained in the file. Then, it implements the methods that are used to execute the scenes.

```

Data:  $m$  = start scene message
Result: Scene execution
message_parts  $\leftarrow$  split  $m$  by ' ';
if message_parts[0] is 'start_scene' then
  start_scene(message_parts[1]);
  while instruction  $\leftarrow$  get_generator_instruction();
  do
    if instruction.type is 'PDDL' then
      | execute_pddl_action(instruction.command);
    else
      | execute_camelot_instruction(instruction.command);
    end
  end
end

```

Figure 6.7: Algorithm used to handle the scene execution.

The most important part of this class is the scene execution that spans over different methods. Figure 6.7 shows the algorithm that explains how the scene execution works. A scene starts execution when the experience manager sends a `start_scene` message. When this message is received, the scene controller starts the scene by creating the generator associated with that scene. Then, it iterates over the generator and it sends one instruction at a time to Camelot for execution. Every time Camelot reports that an instruction has been executed, the scene controller sends the next instruction to Camelot. The `while` loop that is shown in Algorithm 6.7 is not implemented as a python loop in the scene controller class. It is a call to a function that happens in the main loop of the `GameController` to keep all the other aspects of the wrapper running even when a scene is being executed.

6.2 PaSSAGE

PaSSAGE² is an experience manager developed by Thue et al. (2007). PaSSAGE is a system for interactive storytelling that utilizes an automatically learned player model to inform its narrative decisions. This player model is used to estimate the playing styles that the current player prefers, and applies this information to adjust the content of the interactive story in real time. The player model can keep track of the player’s preferences using five categories of play style: *fighter*, *method actor*, *storyteller*, *tactician*, and *power gamer*. The player model is primarily learned during the dialogue sessions, and it is based on the replies the player gives to the NPCs in the game. Once the player model has some data, the manager decides which encounter should occur next based on the player’s preferences. An encounter is a section of gameplay that typically involves actions executed by the player and NPCs. Each encounter is annotated with metadata identifying how players with different playstyles might enjoy the encounter.

The first version of PaSSAGE was developed within a game called *Annara’s Tale*, a single-player, story-based video game where the player controls a story character in a virtual environment. *Annara’s Tale* was developed using the Aurora Neverwinter Toolset [14]. Figure 6.8 shows a series of screenshots representing different aspects of the game.

When I had the opportunity to access the game’s source code, I found no separation between the experience manager’s decisions and the game environment. Therefore, this game’s experience manager and environment were developed using the Thue and Bulitko’s (2018) joint perspective. Many of the scripts controlling the game environment contained part of the logic to decide which action the experience manager would execute next.

There are three main reasons why I have decided to use PaSSAGE as an example of an experience manager in my dissertation. Firstly, the decision-making operations of PaSSAGE are relatively simple: it involves keeping track of a player model and selecting an encounter based on that model. Consequently, compared to more complex managers, converting PaSSAGE from a joint to a disjoint perspective should be relatively straightforward while still offering a valuable example to explain how this conversion is done. Secondly, the PaSSAGE experience manager has been reimplemented to be used in a number of research projects [132, 134, 94, 156], thus making it well known across the field and a great candidate to be used as an example of an experience manager. Finally, because my supervisor, Prof. David Thue, is the developer of PaSSAGE, I have been granted access to the manager’s source code and gained insights into its design. Since converting from a joint to a disjoint perspective requires a deep understanding of the manager’s design, accessing the source code has been crucial to my ability to use PaSSAGE as the experience manager in my dissertation.

In Section 4.4, I classified the method used by PaSSAGE as a “hand-made function” decision constraint. However, in Section 4.9, when answering question 1.a, I stated that the protocol would support experience managers that use an

²As for Camelot in the previous section, from now on, I will not use italic font when referring to PaSSAGE since it is an integral part of this section.



Figure 6.8: Screenshot of the game *Annara's Tale* taken from Figure 5.9 of Thue (2015) dissertation [127]. Top left: the player's character, Annara. Top right: Maedorn Forest. Bottom left: dialogue with a non-player character. Bottom right: combat.

abstract state representation with preconditions and effects. Despite this mismatch, I chose to implement PaSSAGE as the experience manager in my dissertation because I believe it can be modified to use an abstract state representation with preconditions and effects. In the original version, the hand-made function used by PaSSAGE directly reads the game state and does not employ preconditions and effects. However, in the process of designing this manager to use the disjoint perspective, I need to transform the way it reads directly the game state to rely on a abstract state representation that uses preconditions and effects. As a result, the hand-made function can be transformed to use preconditions and effects to limit the possible encounters the PaSSAGE experience manager can evaluate.

To convert PaSSAGE from a joint to a disjoint perspective, my first step was to understand the roles played by the experience manager and the environment. This involved understanding how the behaviour of the experience manager would need to adapt when paired with a different environment. In the original game, PaSSAGE receives updates to the player model based on the choices made during dialogue sessions. To ensure that PaSSAGE can receive the same information from

a different environment, it is important to identify which aspects of the game are tied to the environment and which are not. For instance, dialogues are part of the game environment, as they may vary depending on the environment in which they are executed. However, updating the player model does not need to change across different environments. Based on this analysis, it makes sense to have the environment host the dialogues, while the experience manager should focus on receiving the updates to the player model after the player has made their choices during the dialogue session. By separating these two elements, PaSSAGE can more easily adapt to different environments without sacrificing the core functionality of the player model updates. This way of thinking about the dialogue system can also be extended to other experience managers that work with dialogues that update the player model. They might not use the same approach as PaSSAGE with the five values representing the player model, but they will still need to receive updates to the player model after the player has made their choices during the dialogue session.

To ensure a successful conversion to a disjoint perspective, another important aspect to consider is how PaSSAGE would execute encounters. An encounter typically consists of a section of gameplay involving the player and non-player characters. In the original implementation, the experience manager would select the subsequent encounter and execute the corresponding encounter script. However, the implementation of an encounter varies depending on the game environment. So, with the switch to a disjoint perspective, the experience manager would still need to decide which encounter to execute next but delegate the execution to the environment. For instance, an encounter designed for a space-based game where the player controls a spaceship will be different from an encounter designed for a game where the player controls a character in a medieval fantasy world. In the spaceship game, the encounter could involve the player navigating a spaceship and crashing into space debris, while in the medieval fantasy game, the encounter involve the player agent navigating a dungeon and finding a locked door. In both cases, the encounter is designed as an adventure that ends with an unexpected event, but the execution in the game varies based on the environment. This suggests that the encounter implementation is inherently tied to the game environment and should be considered part of it. Therefore, the experience manager should focus on selecting the subsequent encounter and leave the execution to the environment. Like dialogue, the encounter execution can also be extended to other experience managers that work with author-defined scenes.

The result of this analysis is that the environment should be the one that hosts the dialogues and encounters, while the experience manager should be responsible for interpreting the data that come from the environment and selecting the following dialogue or encounter to execute. This analysis influenced the development of the Camelot Wrapper to support conversation and scenes, but other experience managers can also benefit from these features.

6.2.1 Implementation

In this section, I present the implementation of PaSSAGE using the design pattern defined in this dissertation. The design pattern involves separating experience manager and environment using an external platform that facilitates the communication between the two (as presented in Section 5.1). I developed PaSSAGE as an open source Python script that is available on GitHub [73]. Figure 6.9 shows a schematic diagram of the main components of PaSSAGE. The implementation of PaSSAGE is divided into five components: API connector, world state, player model, encounter, and experience manager. In this section, I describe each of these components in detail.

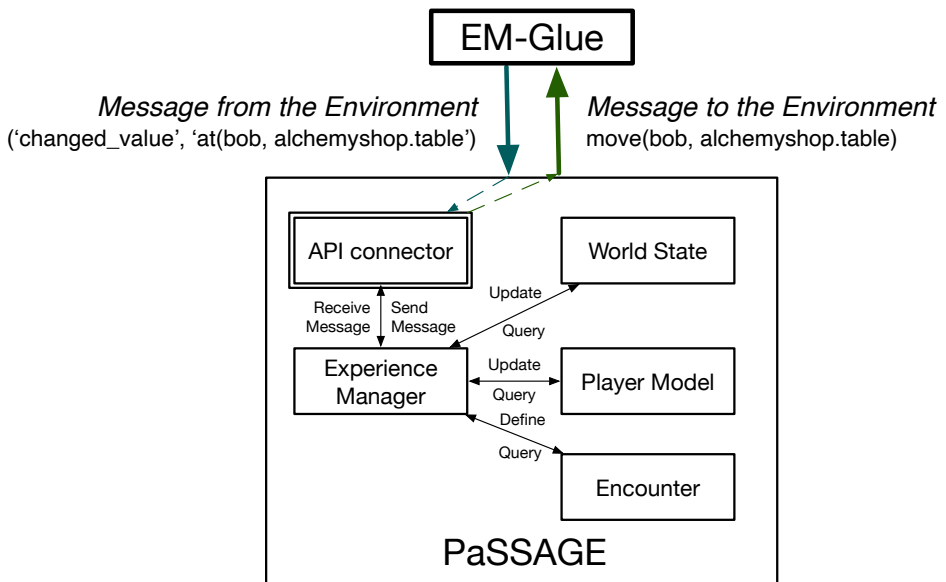


Figure 6.9: A schematic diagram of the PaSSAGE main components. Double boxes indicate that the component is executed in a separate thread. The arrows indicate the flow of information between components.

6.2.1.1 API connector

The API connector facilitates the communication with EM-Glue through the available APIs. It plays a dual role in managing both inbound and outbound messages. To handle incoming messages, the API connector employs a service that uses the communication protocol’s link to make HTTP requests every 250 milliseconds, allowing for a fast reception of new messages. This service is critical for minimizing message delay, as Section 5.2 explains. On the other hand, for outgoing messages, the API connector utilizes an HTTP request on the designated link for sending the experience manager’s messages. The script for the API connector is based on the script developed in the Camelot Wrapper (see Section 6.1.1); thus, I will not

describe it in detail here.

6.2.1.2 World State

The world state is responsible for keeping track of the state of the environment by interpreting the messages related to the world state that comes from the environment. It uses the PDDL data framework library developed for the Camelot Wrapper to keep track of the PDDL state (see Section 6.1.5 for more details). The world state component has two primary duties. First, during the initialization of the experience manager, it receives the PDDL domain and problem from the environment and transforms it into the PDDL data framework representation.

The second duty of the world state component is to interpret the messages the environment sends to update the world state. In Section 5.3.3, I have described the environment's messages to the experience manager to update the world state. Each of these messages is composed of a tuple where the first element is the type of operation that must be performed in the state, and the second element is the details of what must be done. The world state component handles three of these four types of messages.

- **new.** When a tuple with this type of operation is received, the world state component must add a new relation to the world state. This happens when a relation was not previously instantiated in the world state and needs to be added.
- **changed_value:** When a tuple with this type of operation is received, the world state component needs to update the value of a relation in the world state (e.g., from TRUE to FALSE and vice-versa). This happens when a relation was previously instantiated in the world state and needs to be updated.
- **new_entity:** When a tuple with this type of operation is received, the world state component needs to add a new entity to the world state. This happens when the experience manager performs an action that creates a new entity in the world state. With PaSSAGE, this never happens unless the encounter execution creates a new entity.

All these operations are managed by a function that is shown in Listing 6.36.

```

1 def update_environment_state(self, changed_relations):
2     for item in changed_relations:
3         for real in item:
4             if rel[0] == 'new':
5                 self.environment_state.add_relation_from_PDDL(rel[1])
6             elif rel[0] == 'changed_value':
7                 PDDL_relation = self.environment_state.create_relation_from_PDDL(rel[1])
8                 environment_state_relation = self.environment_state.find_relation(relation = ←
9                 ←PDDL_relation, exclude_value = True)
10                environment_state_relation.modify_value(PDDL_relation.value)
11             elif rel[0] == 'new_entity':
12                 self.environment_state.add_entity_from_PDDL(rel[1])

```

Listing 6.36: The function that updates world state in PaSSAGE.

It receives in input a list of tuples, where each tuple is a message received from the environment. The function iterates over the list of tuples, and for each tuple, it iterates over the elements of the tuple. For each element, the function checks the type of operation and performs the corresponding operation on the world state. The function is called every time the experience manager receives a message from the environment.

6.2.1.3 Player Model

The player model component is responsible for maintaining the current state of the player model and updating it based on messages received from the environment. In PaSSAGE (as I described in Section 6.2), the player model records the player's preferred play style using five categories: *fighter*, *method actor*, *storyteller*, *tactician*, and *power gamer*. Each category is assigned a numerical value indicating the player's degree of preference, with higher values indicating greater preference. The message that comes from the environment reflects this representation of the player model. Specifically, when the first element of the tuple received by the experience manager is `update_player_model`, the player model component interprets the second element of the tuple as a string of values to be added or subtracted from the player model. As we can see from Listing 6.37, the second part of the message tuple consists of a set of numbers representing the adjustments to the player's preferences for each play style category. For instance, this message communicates that the player's preferences for fighter, method actor, and power gamer categories remain unchanged while increasing the player's preference for storyteller and decreasing their preference for tactician. The player model is responsible for interpreting these messages and updating its estimates of the player's preferences accordingly.

```
1 ('update_player_model', '(0, 0, 40, -40, 0)')
```

Listing 6.37: Example of a message that updates the player model.

The player model is implemented as a python class with name `PlayerModel` that is defined in Listing 6.38. This class hosts the player model as a set of attributes, one for each play style category. The class also provides a method to update the player model based on a message received from the environment. It retrieves the values from the strings and adds them to the current values of the player model.

```
1 class PlayerModel:
2     def __init__(self):
3         self.fighter = 0
4         self.method_actor = 0
5         self.storyteller = 0
6         self.tactician = 0
7         self.power_gamer = 0
8
9     def update_player_model(self, fighter = 0, method_actor = 0, storyteller = 0, tactician = 0, ↵
10         ↵power_gamer = 0):
11         self.fighter += fighter
12         self.method_actor += method_actor
13         self.storyteller += storyteller
14         self.tactician += tactician
15         self.power_gamer += power_gamer
```

```

def update_player_model_from_message(self, message):
17  message = message[1:-1]
    fighter, method_actor, storyteller, tactician, power_gamer = message.split(",")
19  self.update_player_model(int(fighter), int(method_actor), int(storyteller), int(tactician), int(↵
    ↵(power_gamer))

```

Listing 6.38: The PlayerModel class.

6.2.1.4 Encounter

The encounter component is responsible for hosting the description of the encounters that the experience manager can execute. The encounter handling process is divided into two phases. The first phase is the initialization of the encounter, where the encounter component receives the description of the encounter from the environment and stores it in the encounter component. During this process, it transforms the preconditions for the execution of the encounters into the PDDL data framework. This process allows the experience manager to test the preconditions of the encounters on the environment world state and decide which encounter it can execute.

The second phase is the execution of the encounter, where the experience manager retrieves the metadata and preconditions of the encounter to evaluate which one to execute. Then, it asks the encounter to generate the string for execution composed of the action `start_scene` with the name of the encounter as a parameter between parentheses.

6.2.1.5 Experience Manager

The experience manager component is the script that manages all the phases of the execution of the experience manager. The operations are divided into two phases: the initialization phase and the execution phase. During the initialization phase, the experience manager component follows the handshake protocol, described in Section 5.3.2, to establish a connection with the environment. Listing 6.39 shows the function that handles the handshake protocol in the experience manager.

```

1 def start_platform_communication(self):
    self.wait_platform_online()
3  #Handshake -- Phase 1
    message = self.platform_communication.get_handshake_message("PHASE_1", "message_1") + " ↵
    ↵PaSSAGE"
5  response = self.platform_communication.send_message(message, initialization = True)
    if response is None:
7      raise Exception("Error: Communication with platform failed.")
    if response['text'] != self.platform_communication.get_handshake_message("PHASE_1", "message_2"↵
    ↵):
9      raise Exception("Error: Received unexpected message")
    #Handshake -- Phase 3
11  self.wait_phase_3_start()
    message = self.platform_communication.get_handshake_message("PHASE_3", "message_5")
13  response = self.platform_communication.send_message(message, initialization = True)
    if response is None:
15  raise Exception("Error: Communication with platform failed.")
    if response['text'] == self.platform_communication.get_handshake_message("PHASE_3", "message_7"↵
    ↵):
17  self.PDDL_domain_text = str(response['domain'])
    self.domain = self._PDDL_parser.parse_domain(domain_str = self.PDDL_domain_text)
19  self.PDDL_problem_text = str(response['problem'])

```

```

21     self.problem = self._PDDL_parser.parse_problem(problem_str = self.PDDL_problem_text)
22     self.encounters_received = jsonpickle.decode(str(response['additional_data']))
23     for item in self.encounters_received['encounters']:
24         self.encounter_initialization(item)
25     else:
26         raise Exception("Error: Received unexpected message")
27     #Handshake -- Phase 4
28     message = self.platform_communication.get_handshake_message("PHASE_4", "message_8")
29     response = self.platform_communication.send_message(message, initialization = True)
30     if response is None:
31         raise Exception("Error: Communication with platform failed.")
32     if response['text'] == self.platform_communication.get_handshake_message("PHASE_4", "↔
33         ↪message_10"):
34         self.platform_communication.receive_message_link = response['get_message_url'].replace("/", ↔
35         ↪")
36     self.platform_communication.send_message_link = response['add_message_url'].replace("/", "")
37     else:
38         raise Exception("Error: Received unexpected message")

```

Listing 6.39: Function that handles the handshake protocol in the experience manager.

The handshake protocol comprises four phases, as described in Section 5.3.2. Since phase 2 is irrelevant to the experience manager, the experience manager focuses on phases 1, 3, and 4. It starts by sending a message to EM-Glue by concatenating the manager’s name to the standard message. Then, it waits for the response from EM-Glue and checks if the response is the expected one. The next step is to wait until EM-Glue communicates that phase 3 has started. Once phase 3 has started, the experience manager sends a message to EM-Glue requesting the PDDL domain, problem files, and additional data. The additional data is a JSON file that contains the description of the encounters that the experience manager can execute. The experience manager waits for the response from EM-Glue. Once it receives it, it stores the PDDL domain, problem files, and the additional data in the experience manager component. Once stored, it executes the methods for initializing this data and checks that everything is correct. The final step is to send a message to EM-Glue to confirm that all the data was successfully received and ask for the links for regular communication. The experience manager waits for the response from EM-Glue, and once it receives it, it stores the links for regular communication in the API connector component.

Once the initialization phase has ended, the experience manager component starts the execution phase. Figure 6.10 shows the algorithm used to handle the main loop of execution in PaSSAGE. The main loop starts by receiving a message from EM-Glue. If the message is not empty, it checks if the message is an update of the player model. If it is, it updates the player model using the method described in Section 6.2.1.3. If it is not, it applies the message to the world state using the method described in Section 6.2.1.2. Then, it retrieves the available encounters by checking the preconditions of the encounters against the world state. Once the available encounters are retrieved, it selects the encounter to execute by using the player model and the metadata of the encounter. Finally, it generates the message to send to EM-Glue and sends it.

To select the encounter to execute, the experience manager uses the player model and the metadata of the encounter to calculate a score for each encounter. The score is calculated by multiplying the player model’s score for each of the

```

while EM-Glue is running do
   $m \leftarrow$  receive message from EM-Glue;
  if  $m$  is not None then
    if  $m[0]$  is 'update_player_model' then
      |  $player\_model \leftarrow$  update player model( $m[1]$ );
    else
      |  $worldstate \leftarrow$  apply message( $m$ );
      |  $available\_encounters \leftarrow$  get available encounters( $worldstate$ );
      |  $encounter\_to\_execute \leftarrow$  select
      |   encounter( $available\_encounters, player\_model$ );
      |  $message \leftarrow$  generate message( $encounter\_to\_execute$ );
      | send message( $message$ );
    end
  end
end

```

Figure 6.10: Algorithm used to handle the main loop of execution in PaS-SAGE.

encounter's metadata indicating a multiplying factor for each of the five player model types. An example of such metadata can be found in Appendix B. The function that calculates the score for each of the available encounters and selects the one with the highest score is shown in Listing 6.40.

```

1 def get_most_suited_encounter_dot(self, available_encounters: list[Encounter]) -> Encounter:
2     dict_pm = self.player_model.get_dict()
3     max_value = 0
4     encounter_to_return = None
5     for encounter in available_encounters:
6         value = 0
7         for key in dict_pm:
8             if key in encounter.metadata['target-model']:
9                 value += dict_pm[key] * encounter.metadata['target-model'][key]
10            if value > max_value:
11                max_value = value
12                encounter_to_return = encounter
13     return encounter_to_return

```

Listing 6.40: Function that selects the encounter to execute.

The function starts by retrieving the player model as a dictionary. Then, it iterates through the available encounters and calculates the score for each encounter. The score is calculated by multiplying the player model's score for each of the encounter's metadata indicating a multiplying factor for each of the five player model types. The encounter with the highest score is then returned.

6.2.2 Lessons Learned

When transforming an existing experience manager to use my proposed design pattern, there are several steps that researchers can take to ensure a smooth transition. The first step is to play multiple experiences using the original environment to gain

a deeper understanding of the key decision points for the experience manager. By doing so, researchers can identify which parts of the environment are critical for the experience manager to function properly and which parts are not. This step is crucial to ensure that the experience manager can work effectively with the new design pattern.

After identifying the key decision points, researchers should then access the source code to gain an understanding of where these decision points are implemented. By analyzing the code, researchers can identify the key aspects of the functionalities of the experience manager. These can range from the data structures used to store information to the methods used to retrieve it. Additionally, if the experience manager has a player model, this model should also be examined to determine how it interacts with the environment. This process should be accompanied with reading the research paper(s) that describe the experience manager. Usually research papers are an abstract view of the functionalities of the experience manager, and the source code is the concrete implementation of these functionalities. For example, during the analysis of the original PaSSAGE, I identified that the experience manager was using direct access to the world state to retrieve information about the environment and select the encounters that it could execute. This type of access is a common feature of experience managers developed with a joint perspective. However, when transforming the experience manager to use a disjoint perspective via the design pattern, this access needs to be abstracted and encapsulated in preconditions using a PDDL specification.

Another important aspect to consider when transforming an existing experience manager to use my design pattern is the need to identify the data that the experience manager requires to function properly. This data can vary depending on the goals of the experience manager and the specifics of the environment(s) it operates in. For example, in the case of PaSSAGE, the experience manager needed to access the player model based on five values that identified the player's playstyle. However, a given environment might not have this data available in its original design. When such a situation arises, there are two approaches to consider. The first approach is to find a workaround solution that doesn't require the missing data directly from the environment. For example, in the current design pattern, communication about when an action is executed is not supported. However, this information can be inferred from the updates to the world state that the experience manager receives. Thus, the experience manager can still function effectively without explicit communication about action execution.

The second approach is to modify the environment to include the missing data required by the experience manager. This approach can be more complex and may require an intervention in the environment's code. In the case of PaSSAGE, the initial version of the conversation controller in the Camelot wrapper did not support communication of the data needed by the player model. Therefore, I had to modify the conversation controller to include the player model in the messages that it sent to the experience manager. This allowed the experience manager to access the player model and make decisions based on it. This kind of adaptation may be needed when an environment do not support a particular feature that the experience manager requires. While modifying the environment can be a more

involved process, it can be a worthwhile investment. Rather than starting from scratch with every new experience manager, this approach can enable the creation of a foundation of reusable features that benefit the entire community.

6.3 Random Experience Manager

One type of experience manager used in testing and as a baseline for comparison with other experience managers is the random experience manager [82, 120]. As its name suggests, this type of experience manager relies on random number generator to determine the progression of the game. To show the capability of EM-Glue to support multiple experience managers and expand the available experience managers that can work with the platform, I have decided to implement a random experience manager designed to work with EM-Glue. This experience manager is designed to work with any environment that might be implemented in EM-Glue. The only requirement is that the environment must have a set of scenes that the experience manager can choose to execute when the preconditions are met.

In this implementation, the random experience manager receives messages from EM-Glue and updates the world state accordingly. As the game progresses, the random experience manager tests the preconditions of the scenes (also called encounters in the previous section) sent by the environment during the initialization phase, to determine which ones can be executed. Precondition testing involves checking to see if the necessary conditions are in place for a particular encounter to occur. For example, if a player needs to collect a certain item before proceeding to the next level, the precondition for that scene would be the presence of that item. When multiple scenes can be executed based on the current world state, the random experience manager selects one using a uniform random policy. This means that each of the available scenes has an equal probability of being selected, and it does not require any metadata to work.

While the random experience manager may seem simplistic compared to other, more sophisticated experience managers, it serves an important role in the development experience managers. By providing a baseline against which other experience managers can be measured, it helps developers identify strengths and weaknesses in their designs and refine these systems accordingly. As a result, there are other experience managers in literature that utilize a random approach to represent the player model [82, 120].

6.3.1 Implementation

The random experience manager that I developed in this project bears some similarities to the PaSSAGE experience manager that was described in Section 6.2.1. Just like the PaSSAGE experience manager, I implemented this experience manager using the Python programming language, and the source code for the implementation is publicly available on GitHub [76]. I built the experience manager around three key components, which work together to enable the system to execute encounters and manage the user experience. The first of these components is the encounter class, which is responsible for representing the different encounters that

the experience manager can execute. This class contains all of the necessary information about each encounter, including the name and the preconditions described when the encounter can be executed. The implementation of the encounter class is detailed in Section 6.2.1.4.

The second key component of the experience manager is the API communication class. This class is responsible for managing all of the communication between the experience manager and EM-Glue. It handles tasks such as sending requests to the EM-Glue API, processing responses from the API, and handling any errors that may occur during the communication process. The implementation of this class is described in detail in Section 6.1.1.

Finally, the third component of the experience manager is the experience manager class itself. This class serves as the central hub for the system, managing all of the interactions between the encounter class, the API communication class, and EM-Glue. It is responsible for handling the initialization phase of the communication protocol and for selecting which encounter to execute next, monitoring the progress of the encounter, and updating the user interface as needed. The implementation of this class ties together all of the other components of the experience manager, and it is the core of the system's functionality. This class is similar to the one presented in Section 6.2.1.5, except for the handling of the player model and the algorithm that is used to select the encounter to execute. The algorithm that regulates the encounter selection process is described in Figure 6.11.

```

while EM-Glue is running do
   $m \leftarrow$  receive message from EM-Glue;
  if  $m$  is not None then
     $worldstate \leftarrow$  apply message( $m$ );
     $encounters \leftarrow$  get available encounters( $worldstate$ );
    if  $len(encounters) == 1$  then
      |  $encounter\_to\_execute \leftarrow encounters[0]$  ;
    else if  $len(encounters) > 1$  then
      |  $encounter\_to\_execute \leftarrow$ 
      |    $encounters[random(0, len(encounters) - 1)]$ ;
    else
      | continue;
    end
     $message \leftarrow$  generate message( $encounter\_to\_execute$ );
    send message( $message$ );
  end
end

```

Figure 6.11: Algorithm used to handle the main loop of execution in the Random experience manager.

The main loop starts by receiving a message from EM-Glue. If the message is not empty, it applies the message to the world state using the method described in Section 6.2.1.2. Then, it retrieves the available encounters by checking the precon-

ditions of the encounters against the world state. Once the available encounters are retrieved, it checks how many encounters are available. If there is only one encounter available, it selects that encounter to execute. If there are more than one encounter available, it selects one of them using a uniform random policy. Finally, it generates the message to send to EM-Glue and sends it.

6.4 Evaluation

The evaluation of this case study is divided into two parts. With the first type of evaluation, I want to show that the PaSSAGE experience manager developed to work with EM-Glue is equivalent to the original joint perspective implementation. This evaluation is presented in Section 6.4.1. The second type of evaluation is a comparison between the PaSSAGE experience manager and the Random experience manager to show that they are effectively different experience managers that can be connected to EM-Glue. This evaluation is presented in Section 6.4.2.

6.4.1 Joint vs Disjoint PaSSAGE

To ensure the successful conversion from joint to disjoint perspective, the converted system must exhibit equivalent behaviour to the original system, including performing the same tasks, making the same decisions, and providing the same information that is needed to make those decisions. In this case, to test the equivalence of the two systems, I have divided the evaluation into three parts.

First, the environment should provide all necessary data to the experience manager, just as the original implementation of PaSSAGE's test environment did. This can be proven by showing the correlation between the key points that the original environment used to inform the decision of the experience manager (player model updates during a conversation and triggers for encounters) with how they are implemented in my case study. Demonstrating this point is important to prove that my environment can provide the experience manager with all necessary data, just as the original environment did.

Second, the behaviour of PaSSAGE in the joint perspective should be equivalent to its behaviour in the disjoint perspective. This can be proven by comparing the algorithm used in the original implementation of PaSSAGE with the algorithm that PaSSAGE uses in the case study. Demonstrating this point is important to prove that my implementation of PaSSAGE using the disjoint perspective is valid.

Lastly, all aspects of how data flows between the EM and the Environment (in the joint version) must be preserved in the disjoint version. This can be proven by showing that all the messages that EM-Glue and the protocol provide are successfully routed from the environment to the manager and vice versa. Demonstrating this point is important to prove that EM-Glue's features and functionality are sufficient to allow the EM and the environment to exchange all of the information they exchanged in the joint version.

6.4.1.1 Environment Data

To demonstrate that the environment provides all necessary data to the experience manager, I need to show that the key points that the original environment used to inform the decision of the experience manager are implemented in my case study. There are two main features that the original environment used to inform the decision of the experience manager: player model updates during a conversation and triggers for encounters.

Let us start analyzing the original implementation of PaSSAGE with the *Annara's Tale* game. The updates of the player model in the original implementation of PaSSAGE were triggered by the choices that the player made during a conversation. Figure 6.12 shows a screenshot of the implementation of *Annara's Tale* taken

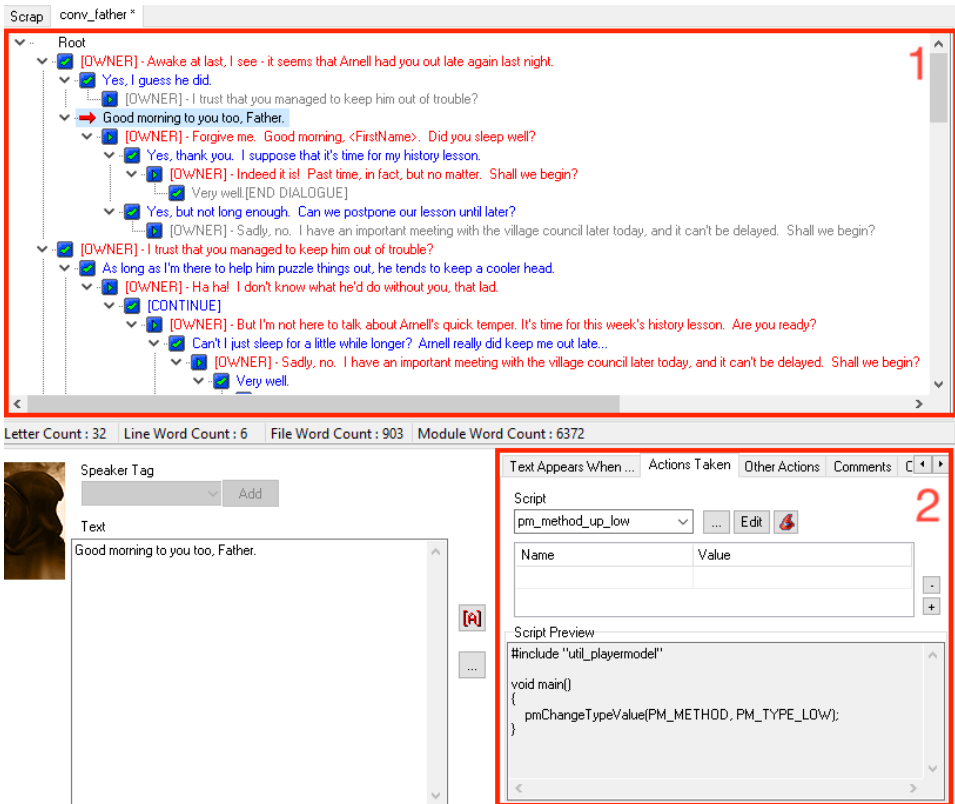


Figure 6.12: Screenshot of the implementation of a conversation in *Annara's Tale* taken from the Aurora Neverwinter Toolset. Box #1 shows the conversation. Box #2 shows the action that the environment will execute when the selected line of conversation is chosen.

from the Aurora Neverwinter Toolset (ANT) [14] analyzing a conversation named `conv_father`. In the image, I have highlighted two sections of the ANT interface: Box #1 shows the lines of the conversation, and Box #2 shows the action that the

environment will execute when the selected line of conversation is chosen. If we analyze Box #1, we can see two colors of lines of the conversation. The red font indicates that that line of conversation will be said by the NPC that the player is talking to. Meanwhile, the blue font indicates that that line of conversation will be said by the player.

If in the ANT interface we click a line with blue font, we can see that the Box #2 will show the action that the environment will execute when the player chooses that line of conversation. These actions were used by the original implementation of Annara's Tale to update the player model. For example, if during the conversation the player will choose the line "Good morning to you too, father" instead of the line "Yes, I guess he did", the environment will update the player model to add a certain value to the `PM_Method` attribute. It does that by executing the script `pm_method_up_low` that uses a method of the player model to update the value of the attribute by adding the constant `PM_TYPE_LOW`. There are two constants that are used as values to add to the player model: `PM_TYPE_LOW` which has the value of 40 and `PM_TYPE_HIGH` which has the value of 100. These updates to the player model are controlled by a set of scripts that are executed when the player chooses a line of conversation. The name of these scripts are formatted in a way that shows the attribute that will be updated and the value that will be added to the attribute. These scripts are similar to each other, it only changes the type within the player model to update and the constant that is added or subtracted from the attribute. There are five types that the player model keeps track of: "Figher", "Method Actor", "Storyteller", "Tactician", and "Power Gamer".

As a result, each player line (blue font) of the conversation may host the execution of a script that updates the player model. Not all the lines have a script associated with them that updates the player model, and it depends on how the author created the conversation. For example, the line "Yes, I guess he did" does not have a script associated with it, while the line "Good morning to you too, father" does.

These dynamics are also implemented in the environment of my case study. As previously mentioned, the Camelot Wrapper's conversation controller manages the conversation dynamics in the case study's environment. Each conversation is contained in a Yarn Spinner file, which contains all the conversation lines as they are in the ANT interface. An example of such a conversation can be found in Appendix A, which transcribes the conversation depicted in Figure 6.12. To ensure consistency between the Yarn file and the scripts that are executed after a line of conversation is selected, an `update_player_model` function with the same values as the original implementation is included in the yarn file every time there is an update to the player model in the original version of PaSSAGE. When the Yarn engine retrieves a conversation line, it also triggers the associated function, which is specified in the `prepare` function of the `encounter` class. This function generates a message for the EM, containing the updated player model values based on the conversation dynamics. More details on this method can be found in Section 6.1.10.1.

This comparison demonstrates that the behaviour of the environment in the joint perspective is equivalent to the behaviour of the environment in the disjoint

perspective. There is only the difference that, in the disjoint perspective using EM-Glue, there are more steps needed to communicate the updates of the player model to the experience manager since all the communications between the two parts needs to be done through EM-Glue.

6.4.1.2 Behavioural Equivalence

To demonstrate that the behaviour of PaSSAGE in the joint perspective is equivalent to its behaviour in the disjoint perspective, I need to compare the algorithm used in the original implementation of PaSSAGE with the algorithm that PaSSAGE uses in this case study. For this comparison to work, I need to show the similarities in how they work on three aspects: the trigger that prompts PaSSAGE to select an encounter, the algorithm to select which is the highest rated player model, and the choice of the encounter based on the player model.

6.4.1.2.1 Trigger for Encounter Selection

The original implementation of Annara’s Tale used a trigger in the environment’s map to start the encounter selection process. As we can see from Figure 6.13, the trigger is the green rectangle that is placed in the map that is highlighted by the Box #2. When the player enters the trigger, the encounter selection process starts by executing the `trg_cue_cta` script highlighted in Box #1.

The `trg_cue_cta` script hosts the start of the encounter selection process. The code related to this script is shown in Figure 6.14. We can see that if the adaptation is enabled it executes a function called `encRequestEncounterFromSet` that starts the encounter selection process based on the player model. I will explain this function in Section 6.4.1.2.2. Once the encounter is selected, it starts the execution of the selected encounter.

When looking at things from a disjoint perspective, the experience manager lacks the same level of access to the environment as in a joint perspective. As a result, the experience manager cannot tell when a player enters a trigger, making it impossible to use the same approach for starting the encounter selection process. To address this, I opted to use the trigger mechanism as encounter preconditions instead. Encounter preconditions refer to the conditions that must be fulfilled for an encounter to become available for execution. In Section 6.1.11, I described how the encounter preconditions are implemented in the case study. When the PaSSAGE experience manager receives the encounters, it also receives the preconditions of the encounters (as described in Section 6.2.1.4). The preconditions for each encounter describe the relations that indicate the position that the player should have in the map in order for the encounter to be available for execution. An example of the definition of such preconditions for an encounter that is available to play in the test environment can be found in Appendix B. Then, every time there is an update in the relations of the world state sent by EM-Glue, the algorithm that handles the experience manager’s execution checks which encounters are available based on these preconditions (Figure 6.10).

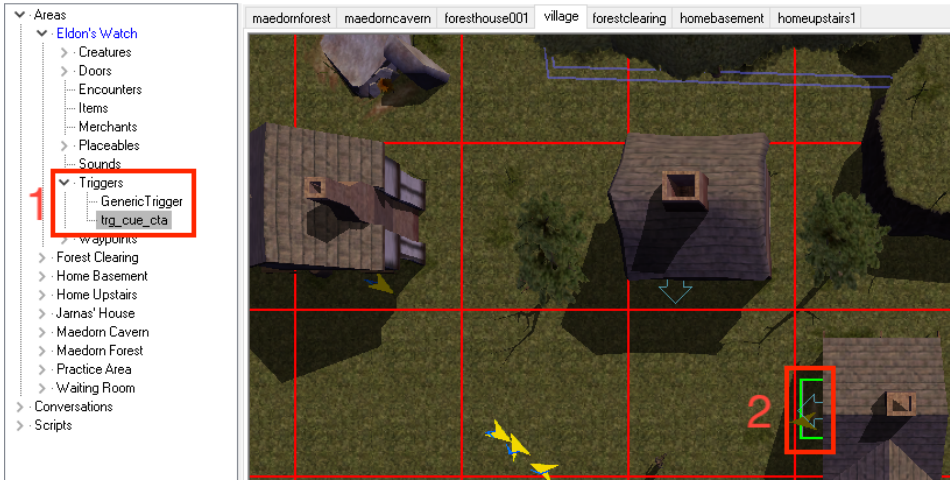


Figure 6.13: Visualization of the “Eldon’s Watch” map of the original implementation of PaSSAGE taken from the Aurora Neverwinter Toolset. Box #1 shows the definition of the trigger script. Box #2 shows where the trigger is placed in the map.

```

void main()
{
    if(ADAPTATION_ENABLED) {
        string enc = encRequestEncounterFromSet(ENC_SET_CALL);
        encExecuteSpecificEncounter(enc);
    }
}

```

Figure 6.14: Code from the `trg_cue_cta` script that hosts the start of the encounter selection process of the “Eldon’s Watch” map of the original implementation of PaSSAGE.

6.4.1.2.2 Encounter Selection Algorithm

In PaSSAGE, encounters are annotated to describe the player preferences they relate to. These annotations are then used to identify encounters that are more likely to be enjoyable for a particular player, based on their player model. Each encounter in the game is associated with an `init` script that initializes the encounter and sets its parameters, including the types of player that are more likely to find it enjoyable.

The `init` script for the `mercy` encounter, shown in Figure 6.15, initializes variables to handle the execution of the encounter. It then uses the `encSetBranchWeight` function to specify two types of player that are likely to enjoy the encounter. Specifically, the script sets the type “Storyteller” as the most likely to enjoy the encounter and also indicates that the type “Fighter” might also enjoy it. By using these annotations, PaSSAGE can select encounters that are best suited to a player’s

```

void main()
{
    string enc = ENC_MERCY;
    string branch;

    encSetMaxTimesRun(enc, MERCY_MAX_TIMES_RUN);
    encSetTriggerTimeout(enc, MERCY_TRG_TIMEOUT);

    branch = MERCY_STANDARD;
    encDeclareBranch(enc, branch);
    encSetBranchWeight(enc, branch, PM_STORY, PM_LARGE);
    encSetBranchWeight(enc, branch, PM_FIGHT, PM_SMALL);
}

```

Figure 6.15: Implementation of the `init` script that defines which type of the player model is more likely to enjoy the encounter `mercy`.

preferences, creating a personalized experience. Each type can be assigned one of four possible weight values, including `PM_SMALL` (1), `PM_MEDIUM` (2), `PM_LARGE` (4), and `PM_ULTRA` (8).

After defining these annotations, the encounter selection algorithm runs to identify the encounter most likely to be enjoyed by the player based on their player model and the encounter annotations. Figure 6.16 shows the implementation of this algorithm in the original version of PaSSAGE.

The `encRequestEncounterFromSet` function in Figure 6.14 uses the function in Figure 6.16 to select the encounter from the set that is most likely to be enjoyed by the player, based on their player model and the annotations associated with the encounters. This function first retrieves the list of available encounters, and then calculates the score of each encounter by iterating over the list and using the player model and the weights of the types defined in the encounter’s `init` script. Finally, it selects the encounter with the highest score for execution.

In the disjoint version of PaSSAGE, encounter weights are defined in the metadata section of the JSON file that describes the encounter in the environment. An example of this definition for the `mercy` encounter can be seen in Appendix B. This metadata is defined by a dictionary containing the names of the player model types as keys and their corresponding weights as values.

During initialization of the experience, the experience manager retrieves all the available encounters and their associated metadata from the environment. As described in Section 6.1.11, this description includes the metadata used for the encounter selection process. During execution, the experience manager selects the encounter that is most likely to be enjoyed by the player based on their player model and the weights of the types defined in the encounter’s metadata. The implementation of this process can be seen in Section 6.2.1.5.

When comparing the two steps involved in selecting encounters, we can observe that they differ in their implementation approach. In a joint approach, the expe-


```

string encRequestEncounter(int quality, string specificEnc="", string setname = "")
{
    [...]
    for(i = 0; i < size && (response < quality
        || quality == ENC_BEST_QUALITY); i++)
    {
        if (specificEnc == "")
        {
            enc = arrayGet(aEncArray, i);
        }
        else
            enc = specificEnc;
        // check encounter suitability by testing the weights on each branch
        // against those in the player model - high responses win
        string branches = enc + "_branches";
        arrayShuffle(branches);
        int numBranches = arrayGetSize(branches);
        int j;
        for(j = 0; j < numBranches && (response < quality
            || quality == ENC_BEST_QUALITY); j++)
        {
            branch = arrayGet(branches, j);
            string branchWeights = encGetBranchWeights(enc, branch);
            response = intdictDot(branchWeights, modelType1Values);
            if(quality == ENC_BEST_QUALITY && response >= maxResponse)
            {
                maxEnc = enc;
                maxBranch = branch;
                maxResponse = response;
            }
        }
    }
}

```

Figure 6.16: Implementation of the function that selects the encounter that is most likely to be enjoyed by the player based on the player model and the annotations of the encounters.

rience manager and environment are tightly coupled without any clear separation, leading to an implementation that reflects this design. On the other hand, the disjoint perspective aims for a more generalizable approach that can be applied to other systems beyond this specific one. To achieve the same outcome as the joint approach, I needed to extract the key functionality and adapt it to work with the metadata of the encounters. This involved defining weights for the player model's types, which in the joint approach was a direct function call to the experience manager, but in the disjoint approach was implemented as a dictionary in each encounter's metadata. Despite the differences in implementation, both approaches effectively annotate which types of player are more likely to enjoy the given encounter.

In the joint perspective, the algorithm that selects the encounter to be executed next was implemented in a way that was already general. The process of converting this algorithm to a disjoint approach was straightforward, as it only required extracting the key functionality, and adapting it to work with the encounter metadata and the Python implementation of the experience manager.

6.4.1.3 Data Exchange

To demonstrate the equivalence between the original implementation of PaSSAGE and the disjoint version, the final step is to verify that data exchanged via EM-Glue is routed correctly and promptly. To achieve this, I ran a case study using the PaSSAGE experience manager and the Camelot environment with the Camelot wrapper. I documented the execution with screenshots, presented in Figure 6.17, where the left column displays the PaSSAGE script execution and the right column displays the environment’s dialogues. To validate that EM-Glue routes the data accurately, I focused on the encounter selection and the player model data exchange.

Let us begin with the first row of Figure 6.17 (Image 1 and 2), where the initial encounter is executed and the player initiates a conversation with an NPC. In this case, PaSSAGE does not need to perform any selection between encounters because only one encounter is available. Upon examining the dialogue, we can observe that the player interacts with the initial dialogue that is presented in Appendix A. To demonstrate the exchange of player model data, I opted to select the second dialogue option, highlighted in red in the figure, which is “Good morning to you too, Father”. This particular option triggers an update to the player model, which must be sent to the experience manager.

After the player clicks, the environment sends a message to the experience manager that contains the update to the player model. In fact, if we focus on the red box within the Image 3, where the execution of the experience manager is displayed, we can see that the experience manager received a message that updates the player model, and proceeded to adjust the internal representation of the player model accordingly. Specifically, it added the value of 40 to the “Method Actor” type of the player model. Once the message is sent, the environment presents the subsequent line of dialogue in the conversation as depicted in the Image 4 of Figure 6.17.

I proceeded with the dialogue by making random selections until it reached its conclusion. The last line of the dialogue is reflected in the third row of Figure 6.17 (Image 5 and 6). In Image 5, we can observe (highlighted with the red box) the updated player model that was modified by the experience manager with the input received from the environment. The resulting player model now has the following values: “Fighter”: 0, “Method Actor”: 40, “Storyteller”: 100, “Tactician”: 0, and “Power Gamer”: 140.

Upon the completion of the dialogue, the encounter was also concluded, prompting me to exit the room that the player was in. Upon exiting the room, the experience manager was presented with two options for the subsequent encounter: **mercy** and **bounty**. Based on the player model’s, the experience manager opted to initiate the **bounty** encounter (also known as “Second-2” in the code). The initiation of the conversation for the **bounty** encounter is displayed in the final row of Figure 6.17 (Image 7 and 8).

Section 5.4.1 specifies that the EM-Glue framework logs all message exchanges during an execution in a database. This feature can be leveraged to verify that data is being transmitted correctly and promptly. Figure 6.18 displays a database dump of the case study’s execution. By analyzing the database dump, we can observe that the environment transmits updates of the world state and player model to the

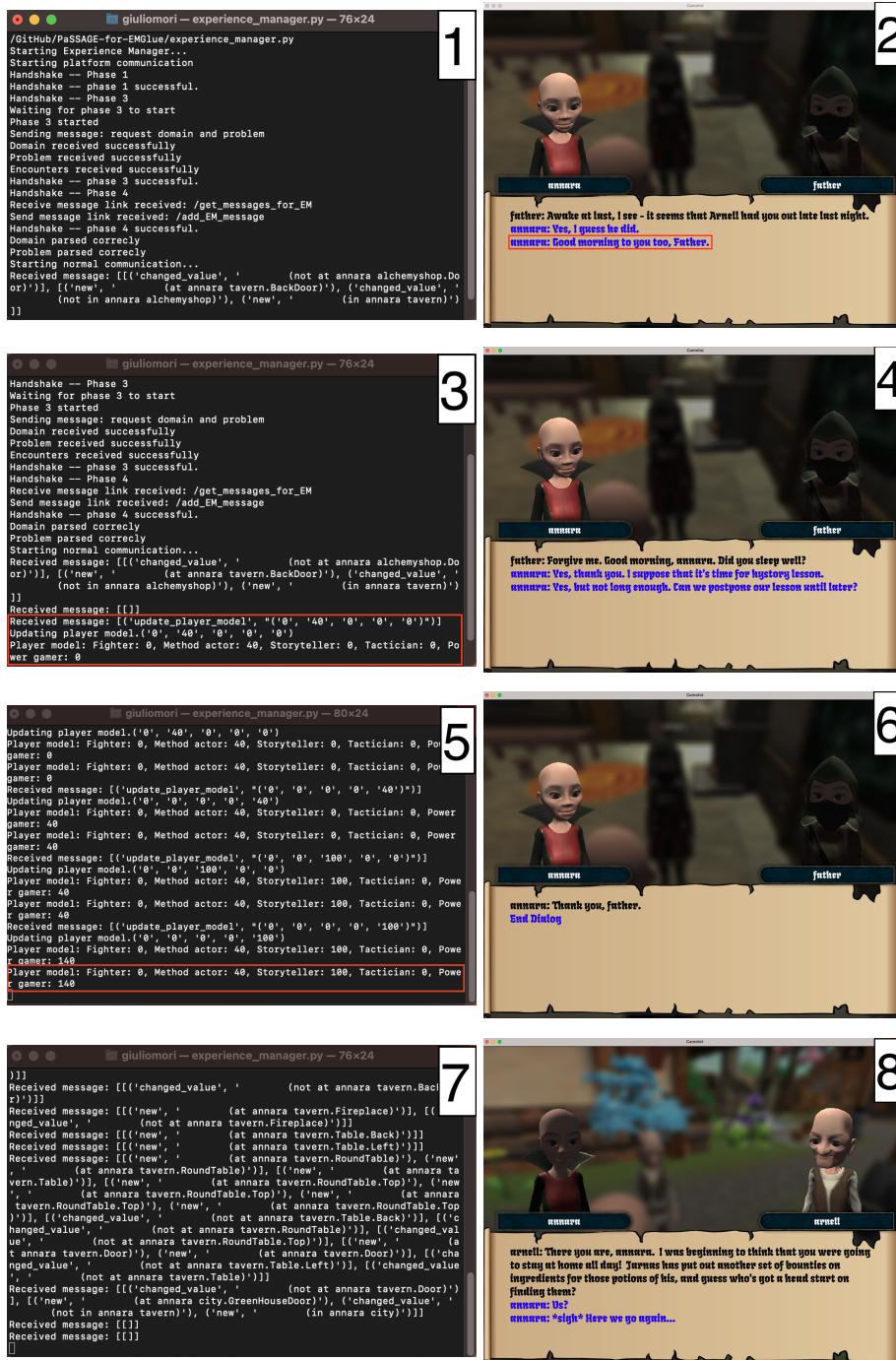


Figure 6.17: Screenshots of the execution of the case study. The first column shows the execution of the PaSSAGE script. The second column shows the execution of the environment with the dialogues. Red boxes shows highlights that are relevant to the discussion. On the top right corner of each image there is a number that indicates the order of the execution and I will refer to it in the text.

experience manager. In turn, the experience manager sends the chosen encounter to the environment by using the `start_encounter` action, as we can see from the messages with `id_message` 13 and 34.

id_message	text
11	[[{"py/tuple": ["changed_value", " (not at bob alchemyshop.Door)"]}]
12	[[{"py/tuple": ["new", " (at bob tavern.BackDoor)"], {"py/tuple": ["changed_value", " (not in bob alchemyshop)"], {"py/t...
13	start_encounter(First)
15	{"py/tuple": ["update_player_model", "(0, '40', '0', '0', '0')"]}
16	{"py/tuple": ["update_player_model", "(0, '0', '0', '0', '40')"]}
17	{"py/tuple": ["update_player_model", "(0, '0', '100', '0', '0')"]}
18	{"py/tuple": ["update_player_model", "(0, '0', '0', '0', '100')"]}
19	[[{"py/tuple": ["changed_value", " (at father tavern.Fireplace)"]}]
20	[[{"py/tuple": ["changed_value", " (not at bob tavern.BackDoor)"]}]
21	[[{"py/tuple": ["new", " (at bob tavern.Table.Back)"]}]
22	[[{"py/tuple": ["new", " (at bob tavern.Table.Left)"]}]
23	[[{"py/tuple": ["new", " (at bob tavern.RoundTable)"], {"py/tuple": ["new", " (at bob tavern.RoundTable)"]}]
24	[[{"py/tuple": ["new", " (at bob tavern.Table)"]}]
25	[[{"py/tuple": ["new", " (at bob tavern.RoundTable.Top)"], {"py/tuple": ["new", " (at bob tavern.RoundTable.Top)"], {"py...
26	[[{"py/tuple": ["changed_value", " (not at bob tavern.Table.Back)"]}]
27	[[{"py/tuple": ["changed_value", " (not at bob tavern.RoundTable)"]}]
28	[[{"py/tuple": ["changed_value", " (not at bob tavern.RoundTable.Top)"]}]
29	[[{"py/tuple": ["new", " (at bob tavern.Door)"], {"py/tuple": ["new", " (at bob tavern.Door)"]}]
30	[[{"py/tuple": ["changed_value", " (not at bob tavern.Table.Left)"]}]
31	[[{"py/tuple": ["changed_value", " (not at bob tavern.Table)"]}]
32	[[{"py/tuple": ["changed_value", " (not at bob tavern.Door)"]}]
33	[[{"py/tuple": ["new", " (at bob city.GreenHouseDoor)"], {"py/tuple": ["changed_value", " (not in bob tavern)"], {"py/tup...
34	start_encounter(Second-2)
37	[[{"py/tuple": ["changed_value", " (at arnell city.Bench)"]}]
38	[[{"py/tuple": ["changed_value", " (not at bob city.GreenHouseDoor)"]}]
39	[[{"py/tuple": ["new", " (at bob city.GreenHouseDoor.In)"]}]
40	[[{"py/tuple": ["changed_value", " (not at bob city.GreenHouseDoor.In)"]}]

Figure 6.18: Database dump of the execution of the case study.

6.4.2 PaSSAGE vs Random EM

The aim of this second evaluation is to show that PaSSAGE and the random experience manager are meaningfully different. I do so by examining their algorithms for selecting the next encounter to be executed. In the preceding section (Sections 6.2 and 6.4.1), I presented a detailed analysis of PaSSAGE. In Section 6.3, I explained that the random experience manager's only deviation from PaSSAGE was the random selection of the next encounter to be executed. Therefore, by focusing on

the selection algorithms of the two experience managers, we can understand the differences between them.

PaSSAGE is designed to interpret two types of information from the environment: updates to the world state and updates on the player model. Firstly, it uses the updates to the world state to keep a record of the current state of the environment. PaSSAGE uses these updates to determine when an encounter is available for execution by testing all encounters' preconditions every time a new update is received. The second type of update is the updates on the player model. PaSSAGE reads and interprets the contents of these messages and updates an internal representation of the player model, which is stored in a class object. When it is time to select an encounter and multiple encounters are available, PaSSAGE uses the stored player model and the weights provided in the encounter description to calculate a score for each available encounter. The encounter with the highest score is then chosen and dispatched for execution in the environment.

In contrast to PaSSAGE, the random experience manager does not interpret updates on the player model that come from the environment and solely focuses on updates of the world state. The random experience manager also uses the updates on the world state to maintain a record of the current state of the environment. Whenever there is a new update, it tests the preconditions of the encounters. When multiple encounters are available, it chooses one at random to be executed.

At a fundamental level, PaSSAGE and the random experience manager differ in their approach to decision-making when selecting the next encounter to execute in a given environment. PaSSAGE uses a deterministic algorithm that takes into account the current state of the environment and the player model to evaluate and score available encounters based on their preconditions and player model weights. This process ensures that the same input will consistently produce the same output, making it predictable and repeatable. In contrast, the random experience manager's decision-making process is stochastic, which means that it involves a random element where each encounter has the same probability of being executed. The manager selects the next encounter at random from the set of available encounters that meet the preconditions, without considering the player model or any other information about the environment.

6.4.2.1 Random EM Execution

To further emphasize the distinction between the two experience managers and demonstrate that the random experience manager can also be connected with EM-Glue, I ran the random experience manager in the version of Annara's Tale implemented in Camelot using the Camelot Wrapper and present several screenshots in Figure 6.19.

The initial screenshot (Image 1) showcases logs from the execution of the random experience manager script. The two red boxes in the image highlight the name of the script being executed and the console displaying a welcome message from the script. The subsequent screenshot (Image 2) exhibits the execution of the environment with the dialogues. Similarly to the previous execution with the PaSSAGE experience manager (Section 6.4.1.3), I opted for the second option of the dialogue to demonstrate that the environment sends updates to the player model



Figure 6.19: Screenshots of the execution of the case study showing the random experience manager. The first column shows the execution of the random experience manager script. The second column shows the execution of the environment with the dialogues. Red boxes shows highlights that are relevant to the discussion. On the top right corner of each image there is a number that indicates the order of the execution and I will refer to it in the text.

as it did before. However, unlike before, the player model isn't updated by the experience manager and the message is ignored, as depicted in the third screenshot (Image 3). Then, I performed the same choices in the conversation as I did in the previous execution. However, this time, the random experience manager performed the selection of the next encounter at random, resulting in the choice of the mercy

encounter, as shown in the fifth and sixth screenshot (Image 5 and 6). In fact, if we compare Image 6 of Figure 6.19 and Image 8 of Figure 6.17, we can see that the dialogue is different.

These screenshots show that the random experience manager can be effectively connected to EM-Glue and that it can be used to execute encounters in the environment. The environment does not know which experience manager is being used and it does not need to be modified to accommodate the random experience manager. As a result, this execution, in combination with the previous execution showed in Figure 6.17, demonstrates that the interchangeability of the experience managers was achieved in this case study.

Chapter 7

Discussion

The objective of my research was to delve into how to achieve the separation and partial-interchangeability of experience managers and environments in the field of experience management. Experience management has been a rapidly growing field in recent years, with an increasing focus on providing high-quality user experiences across various digital platforms. However, it has historically suffered from a lack of standardization and interoperability between experience managers and environments. This deficiency has created several problems for developers in creating new experience managers without the need to implement a new environment from scratch. In the past, developers were required to create a new environment for each experience manager, which led to increased development time and costs. The situation has improved in the past few years with the creation of environments that multiple experience managers can use. However, the communication between experience managers and environments was not standardized, which created a situation where even though multiple experience managers could use an environment, each of those experience managers could only be used with that specific environment. In a perfect world, we would want a solution that allows an experience manager to be used in any environment and vice-versa. This would allow developers to reduce development time, as they could develop an experience manager that could be used across different environments. Furthermore, it would enable them to create more meaningful evaluations of experience managers since they could be tested in different environments and compared with other experience managers in the same environment.

The solution to achieving partial-interchangeability of experience managers and environments lies in creating a shared way of communicating between them. A common way of communicating and rules that define what should be communicated would enable developers to create experience managers that can be used across multiple environments and environments that can be used with multiple experience managers. This could be achieved by developing a communication protocol allowing all experience managers and environments to communicate interchangeably. As a first step towards this direction, in this dissertation, I have designed, presented, and tested a set of tools that experience managers and environments can use to

achieve separation and partial-interchangeability between them.

The remainder of this chapter is organized as follows. First, in Section 7.1, I will discuss the research questions I have presented in Section 1.3. Then, in Section 7.2, I will discuss the contributions that I have made with my research and the benefits that they can bring to the field of experience management. Finally, in Section 7.3, I will discuss the limitations of my approach, as it is presented in this dissertation.

7.1 Research Questions

This section will address the research questions I presented earlier (in Section 1.3). The first research question, along with its sub-questions (1.a, 1.b, and 1.c), relates to the literature review that I presented in Chapter 4. Since I have already discussed the answers to these questions in Section 4.9, I will focus on answering the remaining research questions (questions 2, 3, 4, and their respective sub-questions) in this section.

2. Can the communication be facilitated in a way that does not require changing the internals of the experience manager or environment?

To facilitate the communication between experience managers and environments, without necessitating alterations to their internal structures, it is important to design a flexible communication component. The importance of a flexible communication component derives from the findings of the literature review (Section 4.9), which revealed that experience managers and environments are often implemented using different technologies. This component should allow system developers to transmit the data that is required by the systems without restrictions on the nature of the messages transmitted. To realize this objective, the selection of a technology capable of facilitating a flexible and extensible communication component is critical. In addition, it is necessary to establish a set of protocols and rules to govern the interactions between the experience manager and the environment. These rules and protocols should also be easy to expand as the requirements of each manager and environment dictate.

In alignment with the outlined requirements, I have designed a communication component that led to the creation of EM-Glue. The design process started with the conceptualization of an architecture aimed at dissociating the communication implementation from the experience manager or the environment. This choice derives from the fact that when analyzing the existing experience managers and environments developed using a disjoint perspective, I have seen that the communication is often implemented as an integral part of the experience manager or the environment. However, this is less than ideal as it prompts one of the two systems to shape the communication, forcing the other to comply, resulting in the interchangeability of either the experience manager or the environment, but not both. Take, for instance, *Mimesis* [150], an experience manager that creates a structure to enable the integration of other environments. However, it doesn't provide for the substitution of the experience manager itself. Consequently, I decided to assign the responsibility of communication to a separate component. The architecture is

based on the concept of a communication component serving as an intermediary layer between the experience manager and the environment. This component manages the communication between the two entities, thereby ensuring adherence to the required protocols and rules. Once I delineated the architecture of the framework, the subsequent step was to discern its feasible implementation. This necessitated a technology that is easily implementable by the developers of both the managers and environments and simultaneously offers flexibility and extensibility. I identified RESTful APIs as a technology fulfilling these prerequisites. It allows message transmission through a simple HTTP request, and data can be dispatched in a format that is simple to parse and expand. The final task involved defining the protocols and rules that regulate communication between the experience manager and the environment. During this phase, it was crucial to understand the data requiring exchange between the experience managers and environments. For this reason, in question 4.a, I will propose a datasheet that can be used to facilitate this process. Moreover, the protocols needed to be designed in a way that would allow for the extension of this protocol. This adaptability is instrumental for facilitating partial interchangeability across various experience managers and environments. The following questions will delve into a more detailed discussion on these topics.

2.a. What data needs to be exchanged between the experience manager and the environment?

To ensure both separation and partial interchangeability between the experience manager and the environment, the data that is exchanged is based on an abstract representation of the environment's world state, with the flexibility to incorporate additional data as needed. Typically, the experience manager determines how an experience unfolds in response to what is happening in the environment, guided by the optimization of specific metrics. To understand the environment's state, the environment must provide a representation of the world state during the initialization of the experience (using the PDDL domain and problem specification), as well as regular updates as the experience progresses. Equipped with information on the initial state and potential actions, the experience manager can devise plans and strategies to interact with the environment and attain desired outcomes. The ongoing updates serve to inform the experience manager of the current state of the experience, allowing for necessary adjustments in developed plans and strategies. The conjunction of this data empowers the experience manager to keep track of the history of the experience, which can be employed to compute metrics that shed light on various aspects of the experience. For instance, such metrics could help in discerning player preferences, analyzing the structure of the narrative, and understanding player behaviour, among other possibilities.

Beyond this primary data, the experience manager might need supplementary data from the environment to function or the experience manager might need to communicate additional information to the environment. The nature of this data can vary, based on the specifics of the experience manager's implementation and the characteristics of the environment. For instance, the additional data exchanged might involve information that enables the experience manager to receive updates

about the player model from the environment. For example, for the successful reimplementaion of PaSSAGE, as discussed in Section 6.2, I needed to equip the environment with the ability to dispatch updates to the player model via the `update_player_model` message.

2.b. What are the key components necessary to create a protocol that is general across experience managers and environments?

The key component that is necessary to create a protocol that is general across experience managers and environments is a way of communicating that enables developers of experience managers and environments to extend the information that is exchanged to meet their requirements. This is important because experience managers and environments can be designed in a multitude of ways, and their corresponding tasks may differ significantly, even within the subset of experience managers and environments that I targeted in this dissertation. To fulfill the extensibility of the communication, in Sections 5.3.2 and 5.3.3, I have designed a protocol that is divided into two parts: initiation of the experience, and normal communication. A crucial aspect for generalization in this initialization protocol lies in the environment's capacity to incorporate additional data beyond the PDDL domain and problem, which is essential for the functioning of the experience manager. For example, in the case study involving the Camelot Wrapper with PaSSAGE, I chose to incorporate the encounter information into this additional data field of the initiation protocol.

The normal communication protocol's versatility is highlighted in the way developers of managers and environments can expand it to include additional messages required for transmitting supplementary data. For instance, in the case study of the Camelot Wrapper with PaSSAGE, I decided to include the message type `update_player_model` in the normal communication protocol. This was done to notify the experience manager whenever a new player model update occurred in the environment.

3. How does using the platform impact the constraints experienced by developers during the implementation process?

Developing an experience manager or an environment that is compatible with the framework poses some constraints on the developers of these components. These constraints are primarily related to the structure of the protocol and the information that is exchanged between the experience manager and the environment. We can identify two types of constraints: design constraints and technical constraints. Design constraints encompass the decisions that developers must make during the design phase of an experience manager or environment. When opting to utilize a framework, these choices become crucial in ensuring compatibility with a range of experience managers or environments. The design constraints may involve establishing a flexible and modular architecture that can readily adapt to various requirements. This necessitates careful consideration of factors such as interoperability, scalability, and extensibility. By addressing these design constraints early on, developers can ensure the versatility and future-proofing of their solution.

For instance, let's consider the case study involving the Camelot Wrapper. The design of the Camelot Wrapper was specifically made to be used by multiple experience managers. Consequently, I opted for an open-source, modular, and flexible architecture that facilitates the addition of new features and functionalities. This decision was made at the beginning of the design process and proved beneficial when I later decided to incorporate new features for use by the experience manager.

These design constraints carry significant importance, as they can profoundly impact the development process of the experience manager or environment. However, a developer may choose to design an experience manager or environment while disregarding these design constraints and opting for a more rigid design. Although this approach may result in a simpler implementation process, it can also restrict the adaptability of the experience manager or environment, thereby nullifying the potential benefits of interchangeability that the framework offers.

In addition to the previously mentioned constraints, another important design constraint arises from the framework's reliance on messages based on an abstract representation of the environment's world state. This design constraint can impose limitations on the suitability of certain systems to function as experience managers or environments. For instance, an experience manager that requires direct access to the game engine of the environment would not align with the framework's design. Since the framework operates through abstract representations of the world state, an experience manager reliant on specific details of the game engine may not integrate with the framework's messaging system. By acknowledging and adhering to these design constraints, developers can ensure compatibility and interoperability between different experience managers and environments within the framework. Early decision-making in the design phase becomes pivotal, as it sets the foundation for creating adaptable and versatile components capable of effectively utilizing the framework in conjunction with diverse experience managers or environments.

The technical constraints are associated with the actual implementation of the experience manager and environment. These constraints primarily involve the communication between the experience manager and the environment. The framework relies on RESTful APIs for communication, which necessitates the implementation of a communication component capable of formatting and dispatching HTTP requests to the environment. Additionally, it should be able to parse the messages exchanged between the environment and the experience manager. The parsing of these messages depends on the message format chosen by the developer of the experience manager or environment. To simplify the decision-making process for the message format, I have provided a set of guidelines in my answer to Question 4.a. These guidelines assist developers in ensuring compatibility with the framework.

I documented how I navigated of these design and technical constraints during my transformation of the PaSSAGE experience manager to a disjoint perspective, to ensure compatibility with the framework. This process helped in assessing the level of constraints experienced by developers during the implementation process. Although the conversion required significant effort, it was primarily due to the original implementation's reliance on a specific game engine and proprietary language. Once these engine-specific dependencies were removed, the integration into the new framework's communication system was relatively smooth. This case study high-

lights that while EM-Glue does impose certain constraints on the developers of experience managers and environments, the framework nonetheless offers a certain degree of flexibility, which enables developers to utilize it alongside a selected range of experience managers and environments.

This case study illustrates that the degree of flexibility that the protocol allows is instrumental in shaping the ability of the experience manager to have access to the information it requires to function. In addition, the framework's adaptability allows developers to implement a variety of strategies to bridge the differences between the environment and the experience manager, rather than mandating a specific methodology. In this way, the developers have the freedom to adopt an approach that is best suited for their application.

3.a. What are the steps that developers of an existing experience manager, initially developed using a joint perspective, need to follow to achieve a disjoint perspective implementation while adhering to the framework's rules and protocols?

The process of converting an experience manager from a joint to a disjoint perspective while adhering to the framework's rules and protocols involves several steps. Given the differences between each experience manager, it may not be possible to propose a set of steps that are valid for each experience manager. However, given the experience that I acquired during the conversion process of PaSSAGE, I can propose the thought process that I would apply when approaching a new conversion project. The steps provided are generalized and might require alterations to fit specific cases.

- **Comprehensive system understanding:** This process should start with a thorough understanding of the experience manager they intend to convert. This implies comprehending its components and the relationships between them and determining the boundaries between the environment and the experience manager. This process may require a detailed analysis of the paper describing the experience manager's functionality (if available), a thorough examination of the source code, and playing multiple experiences to gain a deeper understanding of the experience manager's functionality. When converting PaSSAGE, I had access to the source code, the paper describing the experience manager's functionality, and also information from the developer of the project. However, I still had to play multiple experiences to gain a deeper understanding of the experience manager's functionality.
- **Framework compatibility identification:** The next step is to understand if the experience manager possesses all the characteristics necessary to comply with the framework's rules and protocols. These specific characteristics can be found in Section 4.9. If the experience manager does not possess these characteristics, the developer must determine how to bridge the differences between the experience manager and the framework. This process may require the developer to modify the experience manager to comply with the framework's rules and protocols. In the case of PaSSAGE, the experience manager in its original implementation did not possess all the necessary

characteristics to comply with the framework’s rules and protocols. Consequently, I had to modify the experience manager to get the information about the environment from the abstract representation of the world state that comes from the environment via the framework, rather than from the game engine, as it did in the original implementation.

- **Assessment of data required:** The following step is to understand the data that the experience manager requires to function. This includes identifying the data that the environment must provide to the experience manager during the initialization of the experience and the data that the environment must provide to the experience manager during the experience. During this process, the framework that I propose in response to Question 4.a will help the developers to identify the data that needs to be exchanged between the experience manager and the environment. When converting PaSSAGE, to understand the data that was necessary, I had to analyze the source code and the paper describing the experience manager’s functionality.
- **Modification and addition of data:** In cases where the environment does not provide all data necessary for the experience manager’s operations, the developer must comprehend how to procure this necessary information. This scenario could lead to two potential solutions. The first approach involves modifying the environment to deliver the required data. This could also require adding messages to the normal communication protocol, as delineated in my answer to Question 2.b. The second strategy requires the developer to estimate this information from the available data. For instance, if the developer seeks information about the player’s actions, they could extrapolate this information from the historical records of the environment’s world state. When converting PaSSAGE, I had to modify the environment to provide the player model data that was necessary for the experience manager’s operations.
- **Implementation of communication protocols:** Once the environment provides all the data necessary, developers should implement the communication layer with EM-Glue. This includes implementing the handshake protocol, as discussed in Section 5.3.2, to initiate communication between the experience manager and the environment. Furthermore, it should ensure ongoing communication by implementing the normal communication protocol plus any added messages as detailed in Section 5.3.3. During the conversion of PaSSAGE, I implemented the communication with the framework with a python class that I described in Section 6.2.1.1.

This process is a general guide to achieving a disjoint perspective in an experience manager that was originally developed using a joint perspective. Some additional steps might be required depending on the specific system in question.

3.b. What are the steps that developers of an environment need to follow to achieve a disjoint perspective implementation while adhering to the framework’s rules and protocols?

For an environment to be compatible with the proposed framework, it must embody

a set of specific characteristics. First, it needs to declare the available actions alongside their preconditions and effects. Such information can be leveraged by managers to direct the course of the game. Second, the environment should accept instructions that allow modifications to the game’s content during gameplay. This might involve relocating non-player characters or generating new items to enhance the player’s experience. Third, it must maintain an abstract representation of the game world. This requirement enables the communication of how the experience is evolving to the manager. Last, the environment should independently manage the game’s core dynamics, such as controlling the physics engine and facilitating interactions with basic items. As an interactive visualization engine, Camelot did not possess all these characteristics. For this reason, I decided to implement an intermediary layer (Camelot Wrapper) between Camelot and EM-Glue that would provide the missing functionalities; see Section 6.1.

After having verified the presence of the previously mentioned characteristics within the environment, the developer’s next task involves the implementation of the communication layer with EM-Glue. This requires the development of a communication component that handles the handshake protocol and the normal communication protocol, as described in Sections 5.3.2 and 5.3.3. When developing Annara’s Tale using Camelot, I integrated the communication with EM-Glue into the Camelot Wrapper.

In case the environment does not possess all the characteristics required by the framework, the developer must identify the missing characteristics and implement them. This is a complex task that requires a thorough understanding of the environment’s internal structure and the ability to modify it. In this case, there is not a set of steps that can be applied to all environments, as each environment is different and it may be implemented in a different way. In my case study, I decided to extend the functionalities of *Camelot* [117], to make it compatible with the design pattern. Camelot was originally developed for use by experience managers and was designed with high flexibility. Despite this, I needed to add new functionalities to make it compatible with EM-Glue and its design pattern. Instead of modifying *Camelot* directly, I developed an intermediary layer that would bridge it and the framework. This approach allowed me to maintain the flexibility of the original system and avoided any changes to the underlying code. However, developing this intermediary layer required a significant amount of development time.

4. How can a designer anticipate what work is needed to connect an experience manager to an environment using the framework?

To ensure that an experience manager or environment can adhere to the framework’s rules and protocols, it is crucial to conduct a comprehensive analysis of their characteristics. This analysis helps designers gain a better understanding of the information that the environment presents and the information that the experience manager needs to manage the user’s experience effectively. One effective method for performing this analysis is to use Thue’s (2015) GEM framework [127], as I demonstrated in the literature review presented in Chapter 4. The GEM framework provides a foundation and a collection of interrelated “building blocks”, each

of which addresses a distinct sub-problem of experience management. A detailed explanation of the GEM framework and its building blocks is provided in Section 4.3. By examining the decision constraint block of the experience manager (as discussed in Section 4.8), we can gain insight into the types of information that are exchanged and the structure of the information required to manage the user's experience. This analysis can help designers identify any potential mismatches between the information provided by the environment, the information required by the experience manager, and what is currently supported by the framework developed with EM-Glue. Once these potential mismatches are identified, the designer can then develop strategies to address these mismatches.

When attempting to implement an experience manager or environment to conform to a framework's rules and protocols, it is probable that one or more mismatches will emerge. Some of these mismatches may be simple to resolve, while others may require a more difficult redesign of the experience manager and environment, or they may not be feasible to address at all. To assist a designer in recognizing early in the process whether an experience manager or environment can be adjusted to comply with the framework's rules and protocols, I present an analysis of the different categories of decision constraint functions found in the literature review in Section 4.4. I also discuss the difficulties that may arise and how they may potentially be resolved. The analysis is divided into two parts. First, I present a matrix of compatibility between experience managers and environments, which will provide an overview of the expected compatibility between the different categories of decision constraint functions when using EM-Glue. Second, I present some examples (taken from the literature review in Section 4.4) of how the different categories of decision constraint functions can be implemented to work with EM-Glue and the adaptations that might be needed. This preliminary analysis is based on the experience that I gained from implementing the case study and from the analyses of the systems that I performed during the literature review. A more thorough analysis would be needed to determine if any particular experience manager or environment can be effectively adapted to work with the framework, following the methodology outlined in Section 6.2.2 along with the steps given in my answers to Questions 3.a and 3.b. Only by conducting a thorough analysis will it be possible to identify any additional challenges or limitations that need to be addressed to ensure successful integration of the systems with the framework.

Matrix of Compatibility. I begin my analysis of the different categories of decision constraint functions with some estimates of how compatible the different categories are within the framework. In my literature review (refer to Section 4.9), I showed that the decision constraint function serves as a layer between experience managers and environments. This building block is responsible for allowing the experience manager to limit the set of available actions to evaluate. However, if we pivot our perspective and view the decision constraint function as a layer that separates the experience manager from the environment, it becomes evident that the environment, too, plays a role in narrowing down the set of available actions that the experience manager needs to evaluate. From this perspective, we can say that the decision constraint function is a joint product of the environment

		Environment			
		Plot Points	Preconditions and Effects	Graph Representation	Hand-made Function
EM	Decision Constraint				
	Plot Points	High	High	Medium	Medium
	Preconditions and Effects	High	Very High	High	High
	Graph Representation	Medium	High	High	Low
Hand-made Function	Medium	High	Low	High	

Table 7.1: Matrix of compatibility between the different categories of decision constraint functions from the environment and experience manager. The values in the table can range between “Very High”, “High”, “Medium”, and “Low”. “Very High” indicates that it would be potentially easy to adapt and connect the experience manager to the environment using EM-Glue. “Low” indicates that it would be potentially difficult to adapt and connect the experience manager to the environment using EM-Glue.

and the experience manager. As a result, if we want to assess the compatibility level of an experience manager to work with a particular environment, we need to consider the characteristics of both the environment and the experience manager regarding the decision constraint function. To visualize this idea, I present a matrix of compatibility in Table 7.1 that shows the combinations of decision constraint functions from the environment and experience manager and my estimates of the compatibility level of having them working together using the framework. In the table, there are four possible values that represent the degree of compatibility: *Very High*, *High*, *Medium*, and *Low*.

The term *Very High* suggests that it would be potentially very easy to adapt and connect the experience manager to the environment using EM-Glue, due to a probable compatibility between the two. This circumstance arises when both the experience manager and the environment are constructed with the *Preconditions and Effects* category of decision constraint function.

The label *High* indicates a considerable chance of successfully adapting and integrating the experience manager with the environment using EM-Glue. Nonetheless, to facilitate this connection, modifications to either the experience manager or the environment may be necessary to ensure compatibility. Such alterations could include the conversion of a particular representation to the Planning Domain Definition Language (PDDL) employed by EM-Glue, or other similar adjustments.

Medium suggests that it would be potentially possible to adapt and use the experience manager with the environment via EM-Glue. However, significant modifications to either the experience manager or the environment are needed to make them compatible. For instance, suppose there is an intention to link an experience manager, which utilizes plot points, with an environment that employs a graph representation. The task of connecting them using EM-Glue might involve the creation of two translation layers. The first layer would convert plot points into PDDL and vice versa, while the second layer would translate PDDL into the graph representation and vice versa.

Finally, *Low* implies a potential challenge in adapting and integrating the experience manager with the environment using EM-Glue. Under this circumstance, the experience manager and environment may not naturally be compatible, and if compatibility exists, achieving it could prove to be arduous. This value has been attributed to situations where one system employs a graph representation, and the other uses a hand-made function to constrain the decisions of the experience manager.

Plot Points. My examination of how certain experience managers might be adapted to operate with EM-Glue starts with two examples of experience managers that use plot points. When an experience manager employs plot points to limit its decisions, there is a good probability that it could be compatible with the framework, with some modifications from the original implementation.

For example, the system described in Nelson and Mateas (2005) [78] uses plot points with ordering constraints to abstract the story’s content. It could be adapted for use with EM-Glue by defining plot points with ordering constraints using the PDDL specification in the environment and sharing this data with the experience manager through the `additional_data` field of the handshake protocol. The experience manager would then interpret the PDDL plot points and use them to guide its decisions.

Similarly, Lee et al. (2014) [50] describes a system that could be modified to work with EM-Glue. The approach would be similar to the one used for the previous experience manager but without requiring ordering constraints.

Preconditions and Effects. When an experience manager uses preconditions and effects to constrain the experience manager’s decisions, it is likely that it can be adapted to work with EM-Glue. In this case, the experience manager would need to be modified to use the PDDL specification to define the preconditions and effects of the actions that it can perform. Another possible approach could also be to implement a custom translation layer that translates PDDL into the language used by the experience manager and viceversa. However, PDDL has some limitations on the elements that can be represented (as I discuss in Section 7.3), thus this conversion might not always be possible.

For example, the AI manager in Façade [62] uses a custom beat sequencing language where the author annotates each beat with selection knowledge consisting of preconditions, weights, weight tests, priorities, priority tests, and story value effects. The translation to adapt the system to be used with PDDL and EM-Glue could be possible if each beat would be translated to a specification similar to a scene in the Camelot Wrapper. In this way, each beat could be represented as a scene where the preconditions are written in the scene specification, and the additional data that are needed are converted to be written in the metadata section of the scene. Then, the experience manager would receive this data during the initialization phase and use it to guide its decisions.

Regarding the Tomaszewski’s (2011) system [137], it is possible to adapt the experience manager for use with EM-Glue by following the same methodology that I employed for PaSSAGE encounters. Furthermore, there is a possibility to

convert the experience manager to use PDDL, given that it solely requires PDDL to monitor the state and execute scenes when they are available. Nonetheless, a thorough analysis is necessary to determine if PDDL can handle the type of information that the experience manager needs to track the state.

The system proposed by Ware and Young (2016) has the potential to be adapted for use with EM-Glue. However, in order to do so, modifications to the experience manager would be necessary to enable it to use PDDL specifications for defining the preconditions and effects of its actions. As with the previous system, a careful analysis is required to determine if PDDL is capable of handling the specific information that the experience manager must track to monitor the state effectively.

The system proposed by Farrell et al. (2019) appears to have a high potential for adaptation to work with EM-Glue. This is because it already uses STRIPS as a representation language, which happens to be the same language on which PDDL is based. The system was also developed using a disjoint perspective.

Regarding the system proposed by De Lima et al. (2018), there are potential opportunities for it to be used with EM-Glue. However, it is worth noting that the system manages the user's experience by providing quests. Therefore, a more detailed and thorough analysis is required to determine if the system can be effectively adapted to work with EM-Glue.

Graph Representation. When a graph is used to limit the possible options that the experience manager has to consider when making a decision, it may be possible to adapt the experience manager to work with EM-Glue. In fact, the graph representation is an abstract representation of the state of the world, and so it may work with the framework. As mentioned in Section 4.4, there are two types of graph representations that I found in my analysis of the literature. First, a node represents a world state, while an edge represents an action that can change the state of the world to another state. Second, a node represents an action or an event that can occur in the game, and an edge represents a transition (as an ordering constraint) to a subsequent action (node) that can happen.

In the case of an experience manager that uses the first type of graph representation, the conversion to use EM-Glue would be straightforward because the manager already needs the information about the world state and possible actions to make decisions. In this case, we would need a translation layer between the abstract state representation of the environment and the graph representation that the experience manager uses to make decisions.

As an example of this conversion, consider the system proposed by Poo Hernandez et al. (2015). They used a narrative graph encoded as states and actions using PDDL. This experience manager is a perfect candidate to be used with EM-Glue with minimal changes. These changes include the conversion to be used in combination with EM-Glue (e.g., implementation of the communication protocol) and the ability to convert an arbitrary PDDL domain and problem into a graph representation.

In the case of an experience manager that uses the second type of graph representation, the conversion to use EM-Glue would be more complex. It is an evaluation that needs to be made on a case by case basis and it depends on the

kind of information that the nodes and edges represent.

For example, Endrass et al. (2014) [27] uses Sceneflow, a hierarchical and concurrent state chart, which specifies the logical and temporal sequence in which scenes are run. In this case, the nodes represent distinct dialogue contexts or phases of discourse and the edges represent ordering constraints between dialogue lines. This particular style of developing an experience manager, where the abstract state of the environment is not used as main parameter taken into consideration for the experience manager to decide, makes it difficult to adapt it to work with EM-Glue.

Hand-made Function. When an experience manager uses a hand-made function to constrain the experience manager’s decisions, it is possible to adapt the experience manager to work with EM-Glue. In fact, I have demonstrated in the case study that Thue et al.’s (2007) experience manager (PaSSAGE) [131], which uses a hand-made function as its decision constraint, can be adapted to work with EM-Glue with some modifications to how the original system works.

Combination of Approaches. When an experience manager uses a combination of approaches to constrain the experience manager’s decisions, the analysis to understand if it can be adapted to work with EM-Glue becomes more complex to generalize. The developer needs to analyze the line between the experience manager and the environment, what is the core information that is shared between the two, and define a solution that can be used with EM-Glue.

4.a. How can a designer recognize when an experience manager and an environment can be connected using the framework?

To identify when an experience manager and an environment can be connected using the framework, the designer needs to analyze the information that the environment shares with the experience manager. As I have previously mentioned, the communication protocol is designed to be flexible and enable developers of experience managers and environments to define additional messages that they can use to communicate with each other. However, if another designer wants to use a particular experience manager with a particular environment, they need to determine what messages are supported by the two systems to be able to connect to each other and work together. To help simply this task, I propose a data sheet that can be used to describe the information that an experience manager and an environment each share with one another. This data sheet should be filled out by the developers of each experience manager and each environment, so that it can be used by other developers to determine how easily the two systems might be connected using the framework. Each repository of experience managers and environments that are compatible with EM-Glue should include this datasheet under a section with name “Table of Compatibility” of the `README.md` file.

The datasheet should contain three tables indicating how the developer implemented the flexible part of the communication protocol. The first table contains the structure of the additional data field defined in the handshake protocol (Section 5.3.2) sent by the environment or accepted by the experience manager. Since

the additional data field is a JSON object, the developer needs to specify the name of each field, the type of data that it contains, and additional information to allow other developers to understand the structure of the field. As a result, the table is composed of three columns: “Key”, “Format”, and “Description”. The “Key” column contains the name of the field, the “Format” column contains the type of data that the field contains, and the “Description” column contains additional information that the developer wants to share with other developers. An example of the first table of the datasheet is shown in Table 7.2.

Key	Format	Description
"encounters"	list of dictionary	<p>Each dictionary has 4 keys "name", "description", "metadata", "preconditions"</p> <p>"name": string, indicates the name of the encounter. The EM will send this name to have an encounter executed.</p> <p>"description": string, description of the encounter that the author defined.</p> <p>"metadata": dictionary, with key "target-model" containing a list of the features that the author has defined as target for the encounter</p> <p>"preconditions": string, a list of preconditions using PDDL</p>

Table 7.2: Example of the first table of the datasheet taken from the GitHub repository of the Camelot Wrapper.

The second table contains the types of messages that the experience manager can send or the environment can receive during the normal communication protocol (Section 5.3.3). In other words, it defines the messages that are accepted during the communication that starts from an experience manager and ends in an environment. In this case, the experience manager communicates actions that the environment must execute. As a result, the table is composed of three columns: “Message accepted”, “Format”, and “Description”. The “Message accepted” column contains the type of message that is accepted, the “Format” column contains the type of data that the message contains, and the “Description” column contains additional information that the developer wants to share with other developers. An example of the second table of the datasheet is shown in Table 7.3.

Message accepted	Format	Description
PDDL actions	string	Message containing the PDDL action to be executed

Table 7.3: Example of the second table of the datasheet taken from the GitHub repository of the Camelot Wrapper.

The third table contains the types of messages that the experience manager can receive or the environment can send during the normal communication protocol (Section 5.3.3). In other words, it defines the messages that are accepted during the communication that starts from an environment and ends in an experience manager. In this case, the environment communicates the updates of what is happening in the environment formatted as a tuple of the form (key, data). As a result, the table is composed of three columns: “Key”, “Format”, and “Description”.

The “Key” column contains the name of the field, the “Format” column contains the type of data that the field contains, and the “Description” column contains description that the developer wants to share with other developers. An example of the third table of the datasheet is shown in Table 7.4.

Key	Format	Description
"new"	string	New relation added to the world state
"changed_value"	string	Relation changed value in the world state
"new_entity"	string	Name of the entity that has been added to the world state. This message is result of a special PDDL action executed by the experience manager
"update_player_model"	string	An update to the player model happened in the environment. The string contains five values formatted as: "('value_1', 'value_2', 'value_3', 'value_4', 'value_5')"

Table 7.4: Example of the third table of the datasheet taken from the GitHub repository of the Camelot Wrapper.

To enhance our understanding of whether a given pairing of an experience manager and environment can be interlinked using the framework and the datasheet that I proposed, we can consider the example of the Camelot Wrapper and the PaSSAGE experience manager, both developed for EM-Glue. Figure 7.1 provides a comparative view of the datasheets for the Camelot Wrapper and the PaSSAGE experience manager. The first row of the table illustrates the structure of the additional data field defined in the handshake protocol. The second row presents the types of messages that the experience manager can transmit, or the environment can receive, as part of the normal communication protocol. The third row presents the types of messages the experience manager can accept, or the environment can dispatch, during the normal communication protocol.

To determine if the Camelot Wrapper and the PaSSAGE experience manager can be effectively connected with each other via EM-Glue, a comparative analysis of their respective datasheets is needed. The first row of the tables in Figure 7.1 provides a visual representation of the additional data field as defined in the handshake protocol. In this instance, we can see that the Camelot wrapper employs an additional data field composed of the key *encounters*. The PaSSAGE experience manager exhibits a similar characteristic, as it is capable of recognizing an additional data field with the *encounters* key. Given that both the Camelot Wrapper and the PaSSAGE experience manager share and process the same additional data field, it can be inferred that they can indeed initialize the communication with each other using EM-Glue. To understand whether the Camelot Wrapper and the PaSSAGE experience manager can communicate with each other during the normal communication protocol, a review of the second and third rows of the tables in Figure 7.1 is needed.

Upon examining the second row of the tables in Figure 7.1, it is apparent that the Camelot Wrapper is capable of receiving messages that contain PDDL actions in string format. Similarly, the PaSSAGE experience manager is able to send messages containing PDDL actions also formatted as strings. Given that the

Camelot Wrapper

Key	Format	Description	Key	Format	Description
		Each dictionary has 4 keys "name", "description", "metadata", "preconditions" "name": string, indicates the name of the encounter. The EM will send this name to have an encounter executed.			Each dictionary has 4 keys "name", "description", "metadata", "preconditions" "name": string, indicates the name of the encounter. The EM will send this name to have an encounter executed.
"encounters"	list of dictionary	"description": string, description of the encounter that the author defined. "metadata": dictionary, with key "target-model" containing a list of the features that the author has defined as target for the encounter "preconditions": string, a list of preconditions using PDDL	"encounters"	list of dictionary	"description": string, description of the encounter that the author defined. "metadata": dictionary, with key "target-model" containing a list of the features that the author has defined as target for the encounter "preconditions": string, a list of preconditions using PDDL

PaSSAGE experience manager

Message accepted	Format	Description	Message accepted	Format	Description
PDDL actions	string	Message containing the PDDL action to be executed	PDDL actions	string	Message containing the PDDL action to be executed

Key	Format	Description	Key	Format	Description
"new"	string	New relation added to the world state	"new"	string	New relation added to the world state
"changed_value"	string	Relation changed value in the world state	"changed_value"	string	Relation changed value in the world state
"new_entity"	string	Name of the entity that has been added to the world state. This message is result of a special PDDL action executed by the experience manager	"update_player_model"	string	An update to the player model happened in the environment. The string contains five values formatted as: "(value_1' value_2' value_3' value_4' value_5)"
"update_player_model"	string	An update to the player model happened in the environment. The string contains five values formatted as: "(value_1' value_2' value_3' value_4' value_5)"			

Figure 7.1: A side by side comparison of the datasheet of the Camelot Wrapper and the PaSSAGE experience manager. The column on the left refers to the datasheet of the Camelot Wrapper, while the column on the right refers to the datasheet of the PaSSAGE experience manager. We can note that the "new entity" message is not present in the datasheet of the PaSSAGE experience manager. This is because the PaSSAGE experience manager does not support the creation of new entities, but the Camelot Wrapper does. In this case, this is not a problem because this message can only occur in response to actions that the experience manager sends to the environment, and PaSSAGE does not send such actions.

Camelot Wrapper can receive and interpret messages with PDDL actions structured as strings, and the PaSSAGE experience manager can issue such messages, it can be deduced that communication from the experience manager to the environment is feasible.

The last analysis that we need to do is to look at the third row of the tables in Figure 7.1. This analysis reveals that the Camelot wrapper transmits four distinct message types: *new*, *changed_value*, *new_entity*, and *update_player_model*. Similarly, the PaSSAGE experience manager can accommodate three types of messages: *new*, *changed_value*, and *update_player_model*. Despite the PaSSAGE experience manager’s inability to process the *new_entity* message, compatibility between the experience manager and the environment remains intact. This is due to the *new_entity* message being generated only after the experience manager asks the environment to execute a special PDDL action that results in the creation of a new entity within the environment. As the PaSSAGE experience manager does not support the formation of new entities, there is no requirement for it to process the *new_entity* message. Having thoroughly examined all three tables within the datasheet, we can conclude that the Camelot Wrapper and the PaSSAGE experience manager are indeed compatible and can be successfully linked via EM-Glue, as demonstrated in Chapter 6.

7.2 Contributions and Benefits

The work presented in this dissertation offers numerous benefits and contributions to the experience management field. One of the primary contributions of this work is the literature review I conducted, which compared and analyzed a set of 24 experience managers developed in the past. By utilizing two theoretical frameworks established in the field (GEM [127] and Thue and Bulitko’s joint/disjoint perspectives), this analysis allowed for a comprehensive understanding of the various approaches used by researchers when developing experience managers found in the literature. As a result of this analysis, other researchers in the field can gain insights into how experience managers have been developed in the past, their challenges, and the lessons that can be learned from them. Through this literature review, I identified several trends that have been followed in implementing the experience managers that I analyzed. Future researchers can use this information to develop their experience managers while knowing the advantages and disadvantages of each approach. Moreover, the insights gained from the literature review can help inform future research in experience management. Researchers can design and implement more effective solutions by understanding the strengths and weaknesses of the various approaches and techniques used in developing experience managers. Additionally, the literature review can help identify gaps in the current understanding of experience management, leading to further research and development in the field.

The design pattern presented in this dissertation is a significant contribution to the field of experience management. I developed this design pattern to achieve a high level of separation and partial-interchangeability between a subset of experience managers and environments using a component, EM-Glue, responsible for the

communication between the two systems. It is based on the results of my literature review results as I described previously, which identified a trend in the implementation of experience managers where the communication with the environment is based on an abstract representation of the environment's state. The experience manager uses this abstract representation to make decisions about the actions that should be performed in the environment. Based on this finding, I created a design pattern that utilizes a third component responsible for the communication between the experience manager and the environment. The design pattern also includes a communication protocol that is composed of two phases, (i) initialization which is used to set up the experience in the environment and share the initial state of the environment with the experience manager, and (ii) normal communication that regulates the communication between the two systems.

The benefits of this design pattern are numerous. First, it enables the separation and partial-interchangeability between experience managers and environments. This means that experience managers can be developed independently of environments and vice versa. Adding a third component responsible for the communication between the experience manager and the environment delegates the responsibility of the communication to an independent component, which allows the experience manager and environment to focus on their specific functionalities and make their implementations independent from each other. This design pattern also allows for the integration of different experience managers and environments as long as they adhere to the protocols and rules specified. This is a significant benefit because it means that the same experience manager can be used in different environments, or different experience managers can be used in the same environment. This flexibility will make creating and testing new experiences easier once a set of compatible experience managers and environments have been developed. Additionally, this design pattern simplifies the development and maintenance of experience managers and environments. This is because a developer needs to develop a communication interface with EM-Glue once, and then the manager can talk with any environment that can connect to EM-Glue, without the need to develop a new interface for each environment. As a result, the development work derived from connecting an experience manager to an environment (or vice versa) is substantially reduced. Furthermore, it makes updating or replacing system components easier without affecting overall functionality.

My implementation of the design pattern presented in this dissertation is another contribution of this work to the field of experience management. It demonstrates that this approach can be applied successfully in a practical scenario and provides researchers with a reference implementation that can be used to develop their experience managers and environments. The implementation involved several key steps. First, my implementation of the third component, EM-Glue, manages the communication between experience managers and environments. I implemented this component using Python, utilizing RESTful APIs to communicate with the experience manager and environment. It handles the initialization of the communication and routing messages between the two systems. Secondly, I adapted an existing environment, *Camelot* [117], to be compatible with the design pattern. This required the development of a wrapper that receives messages from EM-Glue

and translates them into instructions that *Camelot* can understand. The wrapper also keeps track of the abstract state of the *Camelot* environment and handles low-level interactions that *Camelot* does not handle automatically. The Camelot wrapper is also a contribution to the experience management field. It provides other researchers who want to use *Camelot* as their environment (even without using the design pattern) with a wrapper that keeps track of the abstract state of the environment and handles low-level interactions. Finally, I implemented the *PaSSAGE* [131] experience manager to be compatible with the design pattern. This involved two steps. First, I needed to decouple the EM from the environment since they were developed using the Thue and Carstensdottir’s (2018) joint perspective. This required an in-depth analysis of the *PaSSAGE* codebase to identify when the experience manager was directing the environment and when it was receiving information from the environment. Secondly, I needed to generalize the work that *PaSSAGE* did and implement it to use EM-Glue.

The benefits of this implementation are numerous. Firstly, it demonstrates that the design pattern can be implemented in a practical scenario, which is a significant milestone in demonstrating the capability of this design pattern. Secondly, it provides a reference implementation that other researchers can use to develop their experience managers and environments. This reference implementation is valuable as it demonstrates how to implement the design pattern in practice and can be used for understanding the steps to follow when deciding to adopt the design pattern in their system. Thirdly, EM-Glue is available to the public and can be used by other researchers to develop and connect their experience managers and environments using the design pattern. This makes it easier for researchers to build new systems using the design pattern, as they can leverage the existing infrastructure provided by EM-Glue. Additionally, other researchers can now use *Camelot* (with the Camelot Wrapper) as an environment to develop their experience managers using EM-Glue, and they can test it in comparison with the *PaSSAGE* experience manager and the random experience manager. However, using the Camelot Wrapper may include some limitations that are discussed in Section 7.3. Finally, my description of the implementation process is valuable as it provides insights into adapting existing experience managers and environments to be compatible with the design pattern. This information is helpful for other researchers who may wish to adapt their systems and can accelerate the adoption of the design pattern in the field.

7.3 Limitations

The approach described in this dissertation presents some limitations that should be carefully considered when applying it in practical settings. One of the main assumptions underlying the proposed design pattern is that the experience manager and the environment are independent of each other. However, a literature review shows that many existing experience manager systems have been developed using a joint perspective, where the experience manager and the environment are tightly integrated. This suggests that the independent perspective proposed in this dissertation may not be the most common approach used when designing such systems.

It is unclear whether the joint perspective was adopted by choice or necessity in previous systems, as the motivations and constraints of their developers may have varied. Some researchers may have developed integrated systems to give the manager more direct control over the environment. Alternatively, some systems may have been developed with a joint perspective simply because it was the default approach in the field at the time.

Another limitation of the design pattern proposed in this dissertation is that it has been developed for a specific type of experience manager and environment. As noted in the literature review, numerous types of experience managers use different techniques to manage the experience, and various different environments have been created. Designing a generalizable and flexible design pattern that can be applied to all types of experience managers and environments is a challenging task that cannot be fully addressed in a single dissertation. Therefore, this work focuses on developing a design pattern that applies to a specific type of experience manager and environment while attempting to make it as generalizable as possible for future adaptations and extensions. This approach recognizes that there is no one-size-fits-all solution for designing interchangeable systems and that different types of experience managers and environments require different approaches and design patterns. As a result of this consideration, the proposed design pattern is intended to be a partial solution that covers only some possible combinations of experience managers and environments. As such, it is a first step towards establishing some interoperability and standardization within the experience management field. This dissertation provides a starting point for further research on this challenging problem.

Third, it would be useful to demonstrate that the design pattern I presented in this dissertation can work with multiple environments and other experience managers. Due to time constraints, I could only implement two experience managers and adapt one environment to work with the proposed design pattern. However, my extensive research into the design of this pattern and the open access protocol make it possible to adapt other experience managers and environments to work with the proposed design pattern. Therefore, future research should be well-poised to confirm the effectiveness of this design pattern in a broader setting.

Another limitation is the choice of PDDL as a language to represent the abstract state of the environment. When choosing to use PDDL in its basic form, I was aware of its limitations, such as its inability to represent actions whose effects are long-lasting, poor support for numbers, “closed world” assumption, and more. If I or the community decide to use a different language in the future, the platform can be easily adapted, as there is no connection between which language is used and the ways that the platform exchanges messages between environments and EMs. I cannot say the same for the Camelot Wrapper, however, since part of its core functionalities work using PDDL. Many of PDDL 1’s limitations have been solved in later versions of PDDL [31, 34], so it might be useful to update the Camelot Wrapper to support a later version of PDDL to be able to represent more complex abstract states.

The Camelot Wrapper currently has a limitation in that it only supports a player model composed of five numeric values in its conversation module. This

limitation exists because the Camelot Wrapper was initially developed to work with the PaSSAGE experience manager, which utilizes this type of player model. Currently, as far as I know, a general way of implementing a player model does not exist, thus making it difficult to support a general type of player model in the Camelot Wrapper from the start. However, it is important to note that this limitation is only present in the current implementation of the Camelot Wrapper. EM-Glue and the communication protocol used by the wrapper do not restrict the type of player model that can be used. In fact, the communication protocol only requires the use of a message of type `update_player_model` to send updates to the player model (as explained in Section 5.3.3) without specifying any restrictions on the content of the message. In Section 7.4, I discuss how this limitation could be addressed in future work.

7.4 Future Work

The work presented in this dissertation represents a step forward in experience management and opens up new possibilities for future research. In particular, four broad categories of future work could build upon the design pattern presented here. The first category of future work involves extending the design pattern to support additional types of experience managers and environments. This could be done by adapting the protocols and rules presented in the dissertation to accommodate the specific requirements of these new systems. For example, one potential direction could be to explore whether the existing protocols could be used by experience managers that rely on graph-based abstract state representations, rather than the preconditions and effects representations used in the current design pattern.

The second type of future work that could build upon the design pattern presented in this dissertation involves adapting other experience managers to use this design pattern. This would involve modifying existing experience managers or creating new ones to conform to the protocols and rules outlined in the dissertation, allowing them to work seamlessly with EM-Glue, Camelot (with the Camelot Wrapper), my implementation of PaSSAGE, and the random experience manager. Expanding the list of compatible experience managers would be a valuable contribution to the field of experience management, as it would help to establish the design pattern's applicability further. In addition to simply adapting other experience managers to work with the design pattern, there is also an opportunity for researchers to compare the performance of different experience managers in the same environment. By creating a testing environment and evaluating the performance of multiple experience managers within that environment, researchers could gain valuable insights into the relative strengths and weaknesses of different EM systems.

Another category of future work is to add functionalities to EM-Glue that would allow it to support more advanced features. One addition would be to include support for narrative intervention [97], where an EM can intervene before a player action has any chance to affect the environment's state. For example, if the player was about to shoot a critical NPC, an EM could cause the gun to jam. Another addition would be to change how the environment reports player actions

to experience managers. As described in Section 5.3.3, in the current version of the platform, the environment does not directly report when a player action occurs. If the EM wants to know which player action was executed, it must infer it from the changes in the world state. The next version of the platform could change this behaviour by adding a message that allows each environment to send player actions directly to the EM.

Another potential category of future work involves adapting other environments to be compatible with the design pattern. This could be done by adapting existing environments or creating new ones that follow the design pattern and rules presented in this dissertation. Increasing the number of compatible environments would be a step forward in demonstrating the design pattern's ability to adapt in more contexts. Another interesting aspect of this type of future work is the potential to compare the performance of a single experience manager in different environments. While there has been significant research on testing the performance of an experience manager within a single environment, there has been less exploration of how an EM system performs across different contexts. If multiple environments are compatible with the design pattern presented in this dissertation, researchers could create a testing framework that would allow them to evaluate the performance of an experience manager under a wide range of conditions.

In addition, the Camelot Wrapper has room for expansion, specifically in terms of supporting more advanced functionalities. One potential improvement is to add the `update_player_model` message type to be used beyond the conversation context. Currently, the Wrapper only accommodates a specific type of player model composed of five numeric values. To extend its capability, the Wrapper can be updated to support other types of data, such as a dictionary, which will allow for a wider range of player models. Another possible enhancement is to automate the management of player inventory and NPC fights. These are features that are not currently supported by the Camelot Wrapper but can be valuable additions to enhance its functionality.

Finally, one last area of future work could be adding functionality to EM-Glue to support a broader range of features beyond communication. For example, EM-Glue could be leveraged to monitor and analyze the experience in real-time, allowing researchers to better understand how the algorithm performs. Another potential use for EM-Glue could be to enable automated evaluations of the experience by integrating evaluation methodologies available in literature. This additional feature could provide researchers with valuable insights into the strengths and weaknesses of their algorithms. By leveraging the ability of EM-Glue to collect data, researchers can use this data to analyze the experience on the fly. This could save time and resources while gaining a more comprehensive understanding of their EM's performance.

Chapter 8

Conclusion

As this study comes to a close, it is essential to reflect on the research aims and questions and summarize the key findings in relation to them.

The primary goal of this research was to investigate if it is possible to separate experience managers from the environments in which they are used to achieve interchangeability between them. This objective necessitated a thorough analysis of the literature on experience managers, covering their design principles and implementation methods. To accomplish this, I comprehensively reviewed existing research studies on experience managers, analyzing their underlying architecture and functionalities. This enabled me to identify the key critical components in the communication between experience managers and environments. Based on this analysis, I developed a design pattern that involved creating an external component and communication protocols that could be used to delegate the communication between experience managers and environments to this separate component. This design sought to address the challenge of achieving separation and interchangeability by isolating the experience manager from its environment, allowing the possibility of replacing one of the two systems without affecting the other.

The next step in this research was to test the feasibility of this design pattern in a practical implementation. This involved selecting an existing environment and an two experience managers and adapting them to work with the design pattern presented, to study the effectiveness of my approach.

As part of my research, I investigated the field of experience management and made several contributions that advance our understanding of this field. One of the primary contributions of my work is the literature review that I conducted. This review compared and analyzed a set of experience managers available in the literature using two theoretical frameworks. By synthesizing a subset of the existing research on experience management, I was able to analyze how these managers were implemented, which can inform future research in this area.

Using the results of the literature review, I developed a design pattern intended to achieve a high level of separation and interchangeability between a subset of experience managers and environments. This design pattern is based on a component that is responsible for the communication between the experience manager and the

environment. The communication is regulated by a set of protocols defined in the design pattern. By developing this design pattern, I have provided a framework for researchers that supports the separation and interchangeability of experience managers and environments that can open up new possibilities for the field of experience management. These new possibilities include using the same experience manager with different environments or testing different experience managers in the same environment.

To study and demonstrate the effectiveness of my approach, I implemented the communication component as open-source software that other researchers can use: EM-Glue. Additionally, I implemented an existing experience manager (*PaSSAGE* [131]), a random experience manager, and create the compatibility with an existing engine that can create environments (*Camelot* [117]) in ways that follow my design pattern. This work demonstrates the possibility of retrofitting existing systems to use my design pattern, verifies the approach's effectiveness in a practical setting, and tests the ability to support interchangeability. Finally, I made an additional contribution by developing a Camelot wrapper that adds functionalities to Camelot. This wrapper expands the capabilities of the Camelot environment.

It is important to note several limitations of the approach presented in this research. Firstly, the design pattern is tailored to a specific subset of experience managers and environments, which may not be suitable for all cases. In fact, the characteristics that make an experience manager and environment suitable for this design pattern is that they need to work with an abstract state representation of the environment, and they have a defined set of actions with preconditions and effects that can be used to modify the environment. Nonetheless, since the pattern is open source, it can be adapted to other environments and experience managers. Secondly, I tested the effectiveness of the design pattern only in a single real-world scenario with two experience managers and one environment. Further testing with a larger set of experience managers and environments would be necessary to fully demonstrate its support for interchangeability. Lastly, although the protocols and EM-Glue are designed to be generic and adaptable to different languages, the Camelot Wrapper and PaSSAGE versions used in this research rely on PDDL at their core. Therefore, there may be limitations for those who want to use a different language.

These limitations could be addressed in future research. First, the design pattern could be extended to support other types of experience managers that do not rely on an abstract representation of the environment using preconditions and effects. Second, the design pattern could be tested with a larger set of experience managers and environments to demonstrate its capability of achieving partial-interchangeability. Lastly, the design pattern could be extended with a language other than PDDL.

In conclusion, my research investigated the possibility of achieving separation and interchangeability between experience managers and environments. Through a comprehensive literature review, I developed a design pattern that addressed the challenge of achieving separation and interchangeability by isolating the experience manager from its environment. I then demonstrated the implementation of this design pattern in a practical scenario, showcasing its capabilities. While the

design pattern has limitations and can only be applied to a specific subset of experience managers and environments, its open-source nature allows for adaptation to different scenarios. Future research can build upon this work by extending the design pattern to support other experience managers and testing it with a larger set of environments and experience managers. Overall, this research contributes to advancing the experience management field, opening up new possibilities for experimentation and testing.

Bibliography

- [1] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Commun.*, 7(1):39–59, March 1994. ISSN 0921-7126. 67
- [2] David W Aha and Matthew Molineaux. Integrating learning in interactive gaming simulators. In *Challenges of Game AI: Proceedings of the AAAI'04 Workshop*, 2004. 26
- [3] David W Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *Case-Based Reasoning Research and Development: 6th International Conference on Case-Based Reasoning, ICCBR 2005, Chicago, IL, USA, August 23-26, 2005. Proceedings 6*, pages 5–20. Springer, 2005. 26
- [4] Freyr Arinbjarnar. *Directed Emergent Drama*. Ph.d. dissertation, University of York, 2011. URL <https://etheses.whiterose.ac.uk/28840/>. 40
- [5] Maria Arinbjarnar and Daniel Kudenko. Schemas in directed emergent drama. In *Proceedings of the 1st Joint International Conference on Interactive Digital Storytelling: Interactive Storytelling, ICIDS '08*, page 180–185, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 9783540894247. doi:10.1007/978-3-540-89454-4_24. URL https://doi.org/10.1007/978-3-540-89454-4_24. 6
- [6] Maria Arinbjarnar and Daniel Kudenko. Bayesian networks: Real-time applicable decision mechanisms for intelligent agents in interactive drama. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 427–434, August 2010. doi:10.1109/ITW.2010.5593323. ISSN: 2325-4289. 40, 44, 49, 52, 54, 55, 60, 64, 72
- [7] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bénézet, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-López, Daniel Molina, Richard Benjamins, et al. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82–115, 2020. 34
- [8] R. S. Aylett, S. Louchart, J. Dias, A. Paiva, and M. Vala. *Fearnot! An Experiment in Emergent Narrative*, page 305–316. Springer-Verlag, Berlin, Heidelberg

- berg, 2005. ISBN 3540287388. URL https://doi.org/10.1007/11550617_26. 6
- [9] Ruth Aylett, Sandy Louchart, Anders Tyachsen, Michael Hitchens, Rui Figueiredo, and Carlos Delgado Mata. Managing emergent character-based narrative. In *Proceedings of the 2nd International Conference on Intelligent Technologies for Interactive Entertainment, INTETAIN '08*, Brussels, BEL, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 9789639799134. 6
- [10] Heather Barber and Daniel Kudenko. Dynamic generation of dilemma-based interactive narratives. In *Proceedings of the Third AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE'07*, page 2–7. AAAI Press, 2007. 6
- [11] Heather Barber and Daniel Kudenko. Dynamic generation of dilemma-based interactive narratives. In *3rd AI and Interactive Digital Entertainment Conference (AIIDE 2007)*, pages 2–7, Palo Alto, California, June 6-8 2007. AAAI Press. 47
- [12] Joseph Bates. Virtual reality, art, and entertainment. *Presence: The Journal of Teleoperators and Virtual Environments*, 1(1):133–138, 1992. 23
- [13] Michael Bayer. Sqlalchemy. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. aosabook.org, 2012. URL <http://aosabook.org/en/sqlalchemy.html>. 91
- [14] BioWare Corp. The Aurora Neverwinter Toolset. <http://nwn.bioware.com/>, 2002. 142, 155
- [15] Michael Booth. The AI systems of Left 4 Dead. Presentation at the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2009), October 2009. 46
- [16] Manuel Braunschweiler, Steven Poulakos, Mubbasir Kapadia, and Robert W. Sumner. A Two-Level Planning Framework for Mixed Reality Interactive Narratives with User Engagement. In *2018 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, pages 100–107, December 2018. doi:10.1109/AIVR.2018.00021. ISSN: null. 42, 44, 49, 52, 53, 55, 56, 60, 63, 66, 68, 72
- [17] Eric Cesar Jr, Esguerra Vidal, and Alexander Nareyek. A real-time concurrent planning and execution framework for automated story planning for games. In *Workshops at the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011. 26, 27
- [18] Yun-Gyung Cheong and R. Michael Young. Narrative Generation for Suspense: Modeling and Evaluation. In Ulrike Spierling and Nicolas Szilas, editors, *Interactive Storytelling*, Lecture Notes in Computer Science,

- pages 144–155, Berlin, Heidelberg, 2008. Springer. ISBN 978-3-540-89454-4. doi:10.1007/978-3-540-89454-4_21. 39, 44, 55, 56, 60, 61, 66, 68, 70, 72
- [19] Samuel Colvin. Pydantic, 2022. URL <https://docs.pydantic.dev/>. 93
- [20] Andrea Corradini, Manish Mehta, and Santiago Ontañón. Evaluation of a drama manager agent for an interactive story-based game. In *Proceedings of the 2nd Joint International Conference on Interactive Digital Storytelling: Interactive Storytelling*, ICIDS '09, page 246–257, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 9783642106422. 39, 44, 49, 51, 55, 56, 60, 63, 72
- [21] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine Learning*, 20(3):273–297, September 1995. ISSN 1573-0565. doi:10.1023/A:1022627411411. 33
- [22] Ionut Damian, Birgit Endrass, Peter Huber, Nikolaus Bee, and Elisabeth André. Individualized agent interactions. In *Proceedings of the 4th International Conference on Motion in Games*, MIG'11, pages 15–26, Berlin, Heidelberg, 2011. Springer-Verlag. 40, 41
- [23] Edirlei Soares De Lima, Bruno Feijó, and Antonio L. Furtado. Player behavior and personality modeling for interactive storytelling in games. *Entertainment Computing*, 28:32–48, December 2018. ISSN 1875-9521. doi:10.1016/j.entcom.2018.08.003. URL <http://www.sciencedirect.com/science/article/pii/S1875952118300120>. 42, 44, 49, 50, 60, 62, 66, 68, 72, 178
- [24] Michelangelo Diamanti and Giulio Mori. ML_literature_review, 2020. URL https://github.com/MichelangeloDiamanti/ML_literature_review. 34, 37
- [25] Michelangelo Diamanti and David Thue. Automatic abstraction and refinement for simulations with adaptive level of detail. In *Proceedings of the 15th AIIIDE*, pages 17–23. AAAI Press, 2019. 28, 86
- [26] Magy Seif El-Nasr. Interaction, narrative, and drama: Creating an adaptive interactive narrative using performance arts theories. *Interaction Studies*, 8(2):209–240, 2007. 6
- [27] Birgit Endrass, Christoph Klimmt, Gregor Mehlmann, Elisabeth André, and Christian Roth. Designing User-Character Dialog in Interactive Narratives: An Exploratory Experiment. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2):166–173, June 2014. ISSN 1943-0698. doi:10.1109/TCIAIG.2013.2290509. 40, 44, 49, 50, 72, 179
- [28] Rachelyn Farrell, Stephen G. Ware, and Lewis J. Baker. Manipulating Narrative Salience in Interactive Stories Using Indexter's Pairwise Event Salience Hypothesis. *IEEE Transactions on Games*, pages 1–1, 2019. ISSN 2475-1510. doi:10.1109/TG.2019.2905230. 43, 44, 49, 50, 55, 56, 60, 62, 72, 178

- [29] Sidnie Feit. *TCP/IP*. McGraw-Hill, Inc., USA, 2000. 86
- [30] Roy Thomas Fielding and Richard N. Taylor. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887. 83
- [31] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *CoRR*, 2003. 186
- [32] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. ISBN 0201633612. 17
- [33] Michael Genesereth and Yngvi Björnsson. The international general game playing competition. *AI Magazine*, 34(2):107–107, 2013. 27
- [34] Alfonso Gerevini and Derek Long. Plan constraints and preferences in pddl 3. the language of the fifth international planning competition. In *ICAPS 2006*, 2005. 186
- [35] Richard J. Gerrig and Allan B.I. Bernardo. Readers as problem-solvers in the experience of suspense. *Poetics*, 22(6):459–472, 1994. ISSN 0304-422X. doi:[https://doi.org/10.1016/0304-422X\(94\)90021-3](https://doi.org/10.1016/0304-422X(94)90021-3). URL <https://www.sciencedirect.com/science/article/pii/0304422X94900213>. 56
- [36] Lewis R. Goldberg. An alternative "description of personality": The big-five factor structure. *Journal of Personality and Social Psychology*, 59(6):1216–1229, 1990. 42
- [37] Lewis R Goldberg. An alternative" description of personality": the big-five factor structure. *Journal of personality and social psychology*, 59(6):1216, 1990. 68, 69
- [38] Eun Y. Ha, Jonathan P. Rowe, Bradford W. Mott, and James C. Lester. Goal recognition with markov logic networks for player-adaptive games. In *7th AAAI Conference on AI and Interactive Digital Entertainment, AIIDE'11*, pages 32–39. AAAI Press, 2011. URL <http://dl.acm.org/citation.cfm?id=3014589.3014596>. 5
- [39] Lars Kai Hansen and Peter Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (10):993–1001, 1990. 33
- [40] Brent Harrison and David L. Roberts. Analytics-driven dynamic game adaption for player retention in a 2-dimensional adventure game. In *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE'14*, page 23–29. AAAI Press, 2014. ISBN 1577356810. 41, 44, 49, 52, 55, 57, 60, 62, 66, 67, 72
- [41] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, jul 2006. 62

- [42] Sergio Poo Hernandez, Vadim Bulitko, and Emilie St. Hilaire. Emotion-based interactive storytelling with artificial intelligence. In *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'14, page 146–152. AAAI Press, 2014. ISBN 1577356810. 6, 41
- [43] Richard D Hipp. SQLite, 2020. URL <https://www.sqlite.org/index.html>. 91
- [44] Robin Hunicke and Vernell Chapman. AI for dynamic difficulty adjustment in games. In *Challenges in Game AI Workshop, Nineteenth National Conference on Artificial Intelligence*, pages 91–96, San Jose, California, 2004. AAAI Press. 5
- [45] Chris Klimas. Twine. <https://twinery.org/>. [Online; accessed 9-January-2023]. 43
- [46] Secret Lab. Yarn Spinner, 2022. URL <https://yarnspinner.dev/>. 136
- [47] Ari Lamstein and Michael Mateas. Search-based drama management. In *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*, pages 103–107, 2004. 23, 24
- [48] Brenda Laurel. *Computers as Theatre*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 1993. ISBN 0201550601. 54
- [49] Brenda Kay Laurel. *Toward the Design of a Computer-Based Interactive Fantasy System*. PhD thesis, Ohio State University, 1986. 4
- [50] Seung Y. Lee, Jonathan P. Rowe, Bradford W. Mott, and James C. Lester. A Supervised Learning Framework for Modeling Director Agent Strategies in Educational Interactive Narrative. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2):203–215, June 2014. ISSN 1943-0698. doi:10.1109/TCIAIG.2013.2292010. 40, 42, 44, 49, 54, 55, 72, 177
- [51] Andy Liaw, Matthew Wiener, et al. Classification and regression by random-forest. *R news*, 2(3):18–22, 2002. 33
- [52] Craig A. Lindley and Charlotte C. Sennersten. Game play schemas: From player analysis to adaptive game mechanics. In *International Conference on Game Research and Development, CyberGames '06*, pages 47–53. Murdoch University, 2006. ISBN 86905-901-7. URL <http://dl.acm.org/citation.cfm?id=1234341.1234351>. 5
- [53] Hui Liu, Qingyu Yin, and William Yang Wang. Towards explainable NLP: A generative explanation framework for text classification. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5570–5581, Florence, Italy, July 2019. Association for Computational Linguistics. doi:10.18653/v1/P19-1560. URL <https://www.aclweb.org/anthology/P19-1560>. 34

- [54] Edward Loper and Steven Bird. Nltk: The natural language toolkit. In *In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*. Philadelphia: Association for Computational Linguistics, 2002. 35
- [55] Brian Magerko. Story Representation and Interactive Drama. In *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'05, page 87–92. AAAI Press, 2005. 38, 44, 49, 51, 55, 58, 60, 63, 66, 67, 72
- [56] Brian Magerko, John E. Laird, Mazin Assanie, Alex Kerfoot, and Devvan Stokes. Ai Characters and Directors for Interactive Computer Games. In *Proceedings of the 16th Conference on Innovative Applications of Artificial Intelligence*, IAAI'04, page 877–883. AAAI Press, 2004. ISBN 0262511835. 67
- [57] Brian S. Magerko. *Player Modeling in the Interactive Drama Architecture*. PhD thesis, USA, 2006. AAI3224692. 6
- [58] Chris Martens. Ceptre: A Language for Modeling Generative Interactive Systems. In *Proceedings of the 11th AIIDE*, pages 51–57. AAAI Press, November 2015. 28
- [59] Michael Mateas. *Interactive Drama, Art, and Artificial Intelligence*. PhD thesis, Carnegie Mellon University, 2002. 46
- [60] Michael Mateas and Andrew Stern. Façade: An experiment in building a fully-realized interactive drama. In *Game Developers Conference (GDC'03)*, 2003. URL http://www.cp.eng.chula.ac.th/~vishnu/gameResearch/story_november_2005/MateasSternGDC03.pdf. 6, 38, 39, 58
- [61] Michael Mateas and Andrew Stern. A Behavior Language: Joint Action and Behavioral Idioms. *Life-Like Characters: Tools, Affective Functions, and Applications*, pages 135–161, 2004. 28
- [62] Michael Mateas and Andrew Stern. Procedural authorship: A case-study of the interactive drama Façade. In *Digital Arts and Culture (DAC)*, Copenhagen, November 2005. 46, 50, 177
- [63] Elsevier. Scopus, 2004. URL <https://www.scopus.com/>. 32
- [64] Ludeon Studios. Rimworld, 2013. URL <https://rimworldgame.com/>. 12, 29
- [65] Nintendo. Mario Kart 64. <http://mario.nintendo.com/>, 1996. 12
- [66] Valve Corporation. Left 4 Dead. www.l4d.com, 2008. 1, 12, 46
- [67] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - the Planning Domain Definition Language. 1998. 28, 52, 86

- [68] Manish Mehta, Steven Dow, Michael Mateas, and Blair MacIntyre. Evaluating a conversation-centered interactive drama. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems, AAMAS '07*, pages 1–8, Honolulu, Hawaii, May 2007. Association for Computing Machinery. ISBN 978-81-904262-7-5. doi:10.1145/1329125.1329135. URL <https://doi.org/10.1145/1329125.1329135>. 38, 44, 49, 50, 55, 58, 72
- [69] Donald Michie. Game-playing and game-learning automata. In *Advances in programming and non-numerical computation*, pages 183–200. Elsevier, 1966. 63
- [70] Ian Millington. *Idmillington/undum: A client-side framework for narrative hypertext interactive fiction.*, 2014. URL <https://github.com/idmillington/undum>. 41
- [71] Wookhee Min, Bradford Mott, Jonathan Rowe, Barry Liu, and James Lester. Player Goal Recognition in Open-World Digital Games with Long Short-Term Memory. In *25th International Joint Conference on AI*, pages 2590–2596, 2016. 46, 47
- [72] Giulio Mori. *EV_PDDL*, 2022. URL https://github.com/liogiu2/EV_PDDL. 115
- [73] Giulio Mori. *liogiu2/PaSSAGE-for-EMGlue: Implementation of PaSSAGE for EM-Glue*, 2023. URL <https://github.com/liogiu2/PaSSAGE-for-EMGlue>. 145
- [74] Giulio Mori. *liogiu2/Camelot-Wrapper: Middle layer between Camelot and EM-Glue*, 2023. URL <https://github.com/liogiu2/Camelot-Wrapper>. 105
- [75] Giulio Mori. *liogiu2/EM-Glue: Platform for Decoupling Experience Managers and Environments*, 2023. URL <https://github.com/liogiu2/EM-Glue>. 80, 91
- [76] Giulio Mori. *liogiu2/Random-EM-EM-Glue: Implementation of Random EM for EM-Glue*, 2023. URL <https://github.com/liogiu2/Random-EM-EM-Glue>. 152
- [77] Giulio Mori, David Thue, and Stephan Schiffel. A Structured Analysis of Experience Management Techniques. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, pages 174–180, Atlanta, Georgia, USA, October 2019. AAAI Press. 7, 50, 54, 57, 62
- [78] Mark J. Nelson and Michael Mateas. Search-based Drama Management in the Interactive Fiction Anchorhead. In *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE'05*, page 99–104. AAAI Press, 2005. 23, 24, 38, 44, 49, 55, 58, 60, 63, 72, 177

- [79] Mark J Nelson and Michael Mateas. Another Look at Search-Based Drama Management. In *23rd Conference on Artificial Intelligence*, pages 792–797, Chicago, IL, 2008. AAAI Press. ISBN 9781577353683. URL http://ifaamas.org/Proceedings/aamas08/proceedings/pdf/paper/AAMAS08_{ }0799.pdf. 5, 17
- [80] Mark J. Nelson, David L. Roberts, Charles L. Isbell, and Michael Mateas. Reinforcement learning for declarative optimization-based drama management. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS '06*, page 775–782, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933034. URL <https://doi.org/10.1145/1160633.1160769>. 6
- [81] Mark J. Nelson, David L. Roberts, Charles L. Isbell, Jr., and Michael Mateas. Reinforcement learning for declarative optimization-based drama management. In *5th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '06*, pages 775–782, New York, NY, USA, 2006. ACM. ISBN 1-59593-303-4. doi:10.1145/1160633.1160769. URL <http://doi.acm.org/10.1145/1160633.1160769>. 17, 59
- [82] M.J. Nelson, M. Mateas, D.L. Roberts, and C.L. Isbell. Declarative Optimization-based Drama Management in Interactive Fiction. *IEEE Computer Graphics and Applications*, 26(3):32–41, May 2006. ISSN 1558-1756. doi:10.1109/MCG.2006.55. 5, 6, 24, 38, 44, 49, 51, 55, 58, 60, 63, 64, 66, 69, 72, 152
- [83] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830, 2011. 35
- [84] Federico Peinado and Pablo Gervás. Transferring game mastering laws to interactive digital storytelling. In *International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, pages 48–54. Springer, 2004. 6
- [85] Neil Peirce, Owen Conlan, and Vincent Wade. Adaptive Educational Games: Providing Non-invasive Personalised Learning Experiences. In *2008 Second IEEE International Conference on Digital Game and Intelligent Toy Enhanced Learning*, pages 28–35, November 2008. doi:10.1109/DIGI.2008.30. ISSN: null. 39, 44, 49, 50, 55, 57, 60, 61, 72
- [86] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Simon M. Lucas, and Tom Schaul. General video game ai: Competition, challenges, and opportunities. In *Proceedings of the 13th AAAI, AAAI'16*, page 4335–4337. AAAI Press, 2016. 27
- [87] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON schema. In *Proceedings of the 25th*

- International Conference on World Wide Web*, pages 263–273. International WWW Conferences, 2016. 88
- [88] Richard R Picard and R Dennis Cook. Cross-validation of regression models. *Journal of the American Statistical Association*, 79(387):575–583, 1984. 35
- [89] Sergio Poo Hernandez, Vadim Bulitko, and Marcia Spetch. Keeping the Player on an Emotional Trajectory in Interactive Storytelling. In *AIIDE*, 2015. 41, 44, 49, 52, 55, 58, 60, 62, 66, 68, 72, 178
- [90] Julie Porteous, Marc Cavazza, and Fred Charles. Applying planning to interactive storytelling: Narrative control using state constraints. *ACM Trans. Intell. Syst. Technol.*, 1(2):1–21, dec 2010. ISSN 2157-6904. doi:10.1145/1869397.1869399. URL <https://doi.org/10.1145/1869397.1869399>. 6, 28, 86
- [91] Julie Porteous, João F Ferreira, Alan Lindsay, and Marc Cavazza. Automated narrative planning model extension. *Autonomous Agents and Multi-Agent Systems*, 35(2):1–29, 2021. 28, 86
- [92] Alejandro Ramirez and Vadim Bulitko. Automated planning and player modeling for interactive storytelling. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):375–386, 2014. 6
- [93] Alejandro Ramirez and Vadim Bulitko. Automated Planning and Player Modeling for Interactive Storytelling. *Transactions on Computational Intelligence and AI in Games*, pages 375–386, dec 2015. ISSN 1943-068X. doi:10.1109/TCIAIG.2014.2346690. URL <http://ieeexplore.ieee.org/document/6874544/>. 6, 7
- [94] Alejandro Ramirez and Vadim Bulitko. Automated Planning and Player Modeling for Interactive Storytelling. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):375–386, December 2015. ISSN 1943-0698. doi:10.1109/TCIAIG.2014.2346690. 41, 44, 49, 51, 55, 58, 60, 62, 66, 68, 72, 142
- [95] Alejandro Ramirez, Vadim Bulitko, and Marcia Spetch. Evaluating planning-based experience managers for agency and fun in text-based interactive narrative. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE’13, page 65–71. AAAI Press, 2013. ISBN 1577356071. 6
- [96] Sebastián Ramírez. FastAPI. github.com/tiangolo/fastapi, 2022. 83, 94
- [97] Mark Riedl, C. J. Saretto, and R. Michael Young. Managing interaction between users and agents in a multi-agent storytelling environment. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems - AAMAS '03*, page 741, 2003. ISBN 1581136838. doi:10.1145/860690.860694. URL <http://portal.acm.org/citation.cfm?doid=860575.860694>. 187

- [98] Mark Riedl, C. J. Saretto, and R. Michael Young. Managing interaction between users and agents in a multi-agent storytelling environment. In *2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03, pages 741–748, New York, NY, USA, 2003. ACM. ISBN 1-58113-683-8. doi:10.1145/860575.860694. URL <http://doi.acm.org/10.1145/860575.860694>. 41
- [99] Mark O. Riedl and Andrew Stern. Believable agents and intelligent story adaptation for interactive storytelling. In *Proceedings of the Third International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, TIDSE'06, page 1–12, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540499342. 6
- [100] Mark O. Riedl and Andrew Stern. Believable agents and intelligent story adaptation for interactive storytelling. In *3rd International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, pages 1–12, Darmstadt, DE, December 4-6 2006. Springer. 41
- [101] Mark O Riedl and Robert Michael Young. Narrative planning: Balancing plot and character. *Journal of Artificial Intelligence Research*, 39:217–268, 2010. 56
- [102] Mark O. Riedl, Andrew Stern, Don M. Dini, and Jason M. Alderman. Dynamic Experience Management in Virtual Worlds for Entertainment, Education, and Training. *International Transactions on Systems Science and Applications - ITSSA*, 4, 2008. URL <https://www.cc.gatech.edu/~riedl/pubs/itssa.pdf>. 1, 5, 6, 7
- [103] David L Roberts and Charles L Isbell. A Survey and Qualitative Analysis of Recent Advances in Drama Management. *International Transactions on Systems Science and Applications - ITSSA*, 4(2):61–75, 2008. 5, 6, 7, 17, 24, 48
- [104] David L Roberts, Merrick L Furst, Brian Dorn, and Charles L Isbell. Using influence and persuasion to shape player experiences. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, pages 23–30, 2009. 6
- [105] Justus Robertson and R Michael Young. Perceptual experience management. *IEEE Trans. on Games*, 11(1):15–24, 2019. 6
- [106] Michael E. Rose and John R. Kitchin. pybliometrics: Scriptable Bibliometrics Using a Python Interface to Scopus. *SoftwareX*, 10:100263, 2019. ISSN 2352-7110. 33
- [107] Jonathan P. Rowe, Lucy R. Shores, Bradford W. Mott, and James C. Lester. A framework for narrative adaptation in interactive story-based learning environments. In *Proceedings of the Intelligent Narrative Technologies III Workshop*, INT3 '10, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300223. doi:10.1145/1822309.1822323. URL <https://doi.org/10.1145/1822309.1822323>. 6

- [108] Jonathan P Rowe, Lucy R Shores, Bradford W Mott, and James C Lester. Integrating learning, problem solving, and engagement in narrative-centered learning environments. *International Journal of Artificial Intelligence in Education*, 21(1-2):115–133, 2011. 40, 42
- [109] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 3 edition, 2009. 3
- [110] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):660–674, May 1991. ISSN 2168-2909. doi:10.1109/21.97458. 33
- [111] Ben Samuel, Aaron Reed, Emily Short, Samantha Heck, Barrie Robison, Landon Wright, Terence Soule, Mike Treanor, Joshua McCoy, Anne Sullivan, Alireza Shirvani, Edward T. Garcia, Rachelyn Farrell, Stephen Ware, and Katherine Compton. Playable experiences at AIIDE 2018. In *Proceedings of the 14th AIIDE*, pages 275–280. AAAI Press, 2018. 29, 104
- [112] T. Schaul. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8, Aug 2013. doi:10.1109/CIG.2013.6633610. 27
- [113] Stephan Schiffel and Michael Thielscher. Representing and reasoning about the rules of general games with imperfect information. *J. Artif. Int. Res.*, 49(1):171–206, January 2014. ISSN 1076-9757. 27
- [114] Manu Sharma, Manish Mehta, Santiago Ontanón, and Ashwin Ram. Player modeling evaluation for interactive fiction. In *Proceedings of the AIIDE 2007 Workshop on Optimizing Player Satisfaction*, pages 19–24, 2007. 6
- [115] Manu Sharma, Santiago Ontanón, Manish Mehta, and Ashwin Ram. Drama management evaluation for interactive fiction games. In *AAAI Fall Symposium: Intelligent Narrative Technologies*, pages 139–146, 2007. 6
- [116] Manu Sharma, Santiago Ontañón, Manish Mehta, and Ashwin Ram. Drama Management and Player Modeling for Interactive Fiction Games. *Computational Intelligence*, 26(2):183–211, 2010. ISSN 1467-8640. doi:10.1111/j.1467-8640.2010.00355.x. URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8640.2010.00355.x>. 40, 44, 49, 51, 55, 57, 60, 63, 66, 67, 70, 72
- [117] Alireza Shirvani and Stephen G. Ware. Camelot: a modular customizable sandbox for visualizing interactive narratives. In *Proceedings of the 12th INT workshop at the 16th AIIDE*. AAAI Press, 2020. 12, 29, 38, 43, 73, 82, 104, 174, 184, 190
- [118] Rodrigo L. S. Silva and Frâncila Weidt Neiva. Systematic Literature Review in Computer Science - A Practical Guide. Technical Report RelaTeDCC 002/2016, Federal University of Juiz de Fora, Computer Science Department, 2016. URL <http://rgdoi.net/10.13140/RG.2.2.35453.87524>. 32

- [119] Sam Snodgrass, Omid Mohaddesi, Jack Hart, Guillermo Romera Rodriguez, Christoffer Holmgård, and Casper Hartevelde. Like PEAS in PoDS: The Player, Environment, Agents, System Framework for the Personalization of Digital Systems. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, FDG '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372176. doi:10.1145/3337722.3337756. URL <https://doi.org/10.1145/3337722.3337756>. 3, 24
- [120] Anne Sullivan, Sherol Chen, and Michael Mateas. From abstraction to reality: Integrating drama management into a playable game experience. In *AAAI Spring Symposium: Intelligent Narrative Technologies II*, pages 111–118, 2009. 39, 44, 49, 51, 55, 56, 60, 63, 66, 69, 72, 152
- [121] Anne Margaret Sullivan. *The Grail Framework: Making stories playable on three levels in CRPGs*. PhD thesis, UC Santa Cruz, 2012. 6
- [122] Nicolas Szilas, Thomas Boggini, Monica Axelrad, Paolo Petta, and Stefan Rank. Specification of an open architecture for interactive storytelling. In *Interactive Storytelling*, pages 330–333, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. 27
- [123] Wei Tan, Yushun Fan, Ahmed Ghoneim, M. Anwar Hossain, and Schahram Dustdar. From the service-oriented architecture to the web api economy. *IEEE Internet Computing*, 20(4):64–68, 2016. doi:10.1109/MIC.2016.74. 83
- [124] Michael Thielscher. General game playing in ai research and education. In *Proceedings of the 34th Annual German Conference on Advances in Artificial Intelligence*, KI'11, page 26–37, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642244544. 27
- [125] Michael Thielscher. The general game playing description language is universal. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, page 1107–1112. AAAI Press, 2011. ISBN 9781577355144. 27
- [126] Michael Thielscher. Gdl-iii: A description language for epistemic general game playing. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, page 1276–1282. AAAI Press, 2017. ISBN 9780999241103. 27
- [127] David Thue. *Generalized Experience Management*. PhD thesis, University of Alberta, Canada, 2015. xiv, 3, 5, 17, 25, 44, 45, 51, 57, 71, 73, 77, 143, 174, 183
- [128] David Thue and Vadim Bulitko. Procedural Game Adaptation: Framing Experience Management as Changing an MDP. In *5th Workshop on Intelligent Narrative Technologies*, pages 44–50, Palo Alto, California, 2012. AAAI Press. ISBN 9781577355854. 5

- [129] David Thue and Vadim Bulitko. Toward a Unified Understanding of Experience Management. In *Proceedings of the 14th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'18)*, Edmonton, AB, Canada, 2018. AAAI Press. xi, 8, 9, 10, 17, 31, 37, 38, 43, 44, 73, 77, 142, 183
- [130] David Thue and Elin Carstensdottir. Getting to the point: Toward resolving ambiguity in intelligent narrative technologies. In *Workshops on Intelligent Narrative Technologies and Intelligent Cinematography and Editing*, volume 9, 2018. 48, 185
- [131] David Thue, Vadim Bulitko, Marcia Spetch, and Eric Wasylshen. Interactive storytelling: a player modelling approach. In *Proceedings of the Third AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE'07*, pages 43–48, Stanford, California, June 2007. AAAI Press. xix, 5, 6, 7, 29, 39, 41, 44, 49, 51, 55, 57, 60, 61, 66, 67, 68, 72, 142, 179, 185, 190
- [132] David Thue, Vadim Bulitko, Marcia Spetch, and Michael Webb. Exaggerated claims for interactive stories. In *The Second Joint International Conference on Interactive Digital Storytelling*, pages 179–184, Guimarães, Portugal, December 2009. Springer Berlin / Heidelberg. 142
- [133] David Thue, Vadim Bulitko, Marcia Spetch, and Michael Webb. Socially consistent characters in player-specific stories. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE'10*, page 198–203. AAAI Press, 2010. 6
- [134] David Thue, Vadim Bulitko, Marcia Spetch, and Trevon Romanuik. A computational model of perceived agency in video games. In *AI and Interactive Digital Entertainment Conference (AIIDE)*, pages 91–96, Palo Alto, California, USA, 2011. AAAI Press. 6, 142
- [135] David Thue, Stephan Schiffel, Ragnar Adolf Árnason, Ingibergur Sindri Stefnisson, and Birgir Steinarsson. Delayed roles with authorable continuity in plan-based interactive storytelling. In Frank Nack and Andrew S. Gordon, editors, *Interactive Storytelling: 9th International Conference on Interactive Digital Storytelling, ICIDS 2016, Los Angeles, CA, USA, November 15–18, 2016, Proceedings*, pages 258–269. Springer International Publishing, 2016. ISBN 978-3-319-48279-8. 28, 86
- [136] Zach Tomaszewski. *Marlinspike: An Interactive Drama System*. PhD thesis, 2011. 6
- [137] Zach Tomaszewski. On the use of reincorporation in interactive drama. In *Proceedings of the 18th AIIDE Conference on Intelligent Narrative Technologies IV, AIIDEWS'11-18*, page 84–91. AAAI Press, 2011. 40, 44, 49, 50, 55, 57, 72, 177
- [138] Uvicorn. Uvicorn, 2022. URL <https://www.uvicorn.org/>. 94

- [139] Anton Vinogradov and Brent Harrison. Using multi-armed bandits to dynamically update player models in an experience managed environment. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 18(1):207–214, Oct. 2022. doi:10.1609/aiide.v18i1.21965. URL <https://ojs.aaai.org/index.php/AIIDE/article/view/21965>. 8
- [140] Pengcheng Wang, Jonathan Rowe, Wookhee Min, Bradford Mott, and James Lester. Interactive Narrative Personalization with Deep Reinforcement Learning. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 3852–3858, Melbourne, Australia, August 2017. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-0-3. doi:10.24963/ijcai.2017/538. URL <https://www.ijcai.org/proceedings/2017/538>. 42, 44, 60, 61, 66, 67, 70, 71, 72
- [141] Stephen G Ware. A computational model of narrative conflict. In *Proceedings of the 6th international conference on foundations of digital games*, pages 247–249, 2011. 56
- [142] Stephen G. Ware and R. Michael Young. Intentionality and Conflict in The Best Laid Plans Interactive Narrative Virtual Environment. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(4):402–411, December 2016. ISSN 1943-0698. doi:10.1109/TCIAIG.2015.2489159. Conference Name: IEEE Transactions on Computational Intelligence and AI in Games. 28, 41, 44, 49, 50, 55, 56, 60, 62, 72, 73, 86, 178
- [143] Stephen G. Ware, Edward T. Garcia, Alireza Shirvani, and Rachelyn Farrell. Multi-agent narrative experience management as story graph pruning. In *Proceedings of the 15th AIIDE*, pages 87–93. AAAI Press, 2019. 42, 44, 49, 52, 53, 60, 62, 72
- [144] Stephen G. Ware, Edward T. Garcia, Mira Fisher, Alireza Shirvani, and Rachelyn Farrell. Multi-agent narrative experience management as story graph pruning. *IEEE Transactions on Games*, 2022. (forthcoming). 29
- [145] Allan Weallans, Sandy Louchart, and Ruth Aylett. Distributed drama management: Beyond double appraisal in emergent narrative. In *Proceedings of the 5th International Conference on Interactive Storytelling, ICIDS’12*, page 132–143, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 9783642348501. doi:10.1007/978-3-642-34851-8_13. URL https://doi.org/10.1007/978-3-642-34851-8_13. 6
- [146] Sam Weaver. Yarn Spinner Interpreter for Python, 2022. URL <https://github.com/relaypro-open/YarnRunner-Python>. 137
- [147] Peter Weyhrauch. *Guiding interactive drama*. Ph.d. dissertation, Carnegie Mellon University, Pittsburgh, PA, 1997. 4, 5, 6, 23, 38, 46, 49, 56, 57
- [148] Georgios N. Yannakakis and Manolis Maragoudakis. Player modeling impact on player’s entertainment in computer games. In *Proceedings of the 10th*

- International Conference on User Modeling*, UM'05, page 74–78, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3540278850. doi:10.1007/11527886_11. URL https://doi.org/10.1007/11527886_11. 6
- [149] R. M. Young, M. E. Pollack, and J. D. Moore. Decomposition and causality in partial-order planning. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 188–194. AAAI Press, 1994. 68
- [150] R Michael Young, Mark O Riedl, Mark Branly, Arnav Jhala, RJ Martin, and CJ Saretto. An architecture for integrating plan-based behavior generation with interactive game environments. *Journal of Game Development*, 1(1): 51–70, 2004. 22, 25, 168
- [151] T. Young, D. Hazarika, S. Poria, and E. Cambria. Recent trends in deep learning based natural language processing [review article]. *IEEE Computational Intelligence Magazine*, 13(3):55–75, 2018. 33
- [152] Hong Yu and Mark O. Riedl. A sequential recommendation approach for interactive personalized story generation. In *11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '12, pages 71–78, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 0-9817381-1-7, 978-0-9817381-1-6. URL <http://dl.acm.org/citation.cfm?id=2343576.2343586>. 67
- [153] Hong Yu and Mark O. Riedl. Data-driven personalized drama management. In *9th Conference on AI and Interactive Digital Entertainment*, AIIDE'13, pages 191–197. AAAI Press, 2013. ISBN 1577356071, 978-1-57735-607-3. URL <http://dl.acm.org/citation.cfm?id=3014712.3014742>. 5
- [154] Hong Yu and Mark O Riedl. Optimizing Players' Expected Enjoyment in Interactive Stories. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, page 7. AAAI Press, 2015. 41, 44, 49, 50, 55, 57, 60, 63, 66, 67, 70, 72
- [155] Kristen Yu, Nathan R Sturtevant, and Matthew Guzdial. What is a Quest. In *Proceedings of the Intelligent Narrative Technologies Workshop at AIIDE*, 2020. xix, 6
- [156] Kristen K. Yu, Matthew Guzdial, and Nathan R. Sturtevant. Farmquest: A demonstration of an ai director video game test bed. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 18(1):288–290, Oct. 2022. doi:10.1609/aiide.v18i1.21977. URL <https://ojs.aaai.org/index.php/AIIDE/article/view/21977>. 8, 29, 73, 142
- [157] Rolf A Zwaan and Gabriel A Radvansky. Situation models in language comprehension and memory. *Psychological bulletin*, 123(2):162, 1998. 56

Appendix A

Yarn Spinner Conversation

```
1 title: Start_Initial
2 ----
3 Companion: Awake at last, I see – it seems that Arnell had you out late last night.
4 -> Player: Yes, I guess he did.
5 <<jump ArnellChatting>>
6 -> Player: Good morning to you too, Father.
7 <<update_player_model 0 40 0 0 0>>
8 Companion: Forgive me. Good morning, <FirstName>. Did you sleep well?
9 -> Player: Yes, thank you. I suppose that it's time for hystory lesson.
10 Companion: Indeed it is! Past time, in fact, but no matter. Shall we begin?
11 Player: Very well.
12 <<jump TalkingAboutKing>>
13 -> Player: Yes, but not long enough. Can we postpone our lesson until later?
14 <<jump Stall>>
15 ====
16 title: ArnellChatting
17 ----
18 Companion: I trust that you managed to keep him out of trouble?
19 -> Player: As long as I'm there to help him puzzle things out, he tends to keep a cooler head.
20 <<update_player_model 0 0 0 40 0>>
21 Companion: Haha! I don't know what he'd do without you, that lad.
22 <<jump HystoryLesson>>
23 -> Player: Sometimes a well-placed punch is the best kind of diplomacy.
24 <<update_player_model 100 0 0 0 0>>
25 Companion: Well now. I suppose you have a point, but I'm not sure how I feel about you ↩
26 ↩two becoming the village brawlers.
27 <<jump HystoryLesson>>
28 -> Player: Well fortune favours the bold, they say, and Arnell and I have certainly found our ↩
29 ↩share of fortune over the years.
30 <<update_player_model 0 0 0 0 40>>
31 Companion: Ha ha! That you have, that you have.
32 <<jump HystoryLesson>>
33 ====
34 title: HystoryLesson
35 ----
36 Companion: But I'm not here to tak about Arnell's quick temper. It's time for this week's ↩
37 ↩history lesson. Are you ready?
38 -> Player: Can't I just sleep for a little longer? Arnell really did keep me out late...
39 <<update_player_model 0 0 40 0 0>>
40 <<jump Stall>>
41 -> Player: Very well, let's begin.
42 <<jump TalkingAboutKing>>
43 ====
44 title: Stall
45 ----
```

Companion: Sady, no, I have an important meeting with the village council late today, **and** it ↵
 ↵can't be delayed. Shall we begin?

44 Player: Very well.
 <<jump TalkingAboutKing>>

46 ====
 48 title: TalkingAboutKing

Companion: Alright, let's get started then. We left off last time talking about the king – our ↵
 ↵king – back **when** he was but the young prince of Erafor.

50 Companion: Prince Vi'dal was a man of great skill **and** ambition, his sharpness of wit rivalled ↵
 ↵only by the speed of his blade. None in the realm would dare to challenge him, for ↵
 ↵public humiliation tempted no one.

Companion: A quiet time came over the kingdom, **and** Pince Vi'dal grew more **and** more ↵
 ↵restless. Having no one to challenge him, he could feel his skills slipping away.

52 -> Player: Couldn't he offer a reward to encourage challengers?
 <<update_player_model 0 0 0 40>>

54 Companion: Aye, **and** that's exactly what he did. It started off small – only a few bits of ↵
 ↵gold – but whenever **new** challengers stopped coming forward, the value of the ↵
 ↵reward increased.

Companion: Soon the royal coffers grew thin, the rewards stopped, **and** the prince's ↵
 ↵restlessness returned. Off he went into the countryside, travelling from village to ↵
 ↵village in search of a worthy opponent.

56 <<jump OldMan>>

-> Player: It sounds like the prince needed to find himself a princess...
 58 <<update_player_model 0 0 40 0 0>>
 Companion: Ha ha! That may have helped indeed! Love, however, was far from the prince's↵
 ↵mind. Off he went into the countryside, travelling from village to village in search ↵
 ↵of a worthy opponent.

60 <<jump OldMan>>

-> Player: Surely he could have found a worthy challenger in all of Erafor...
 62 <<update_player_model 0 0 0 40 0>>
 Companion: Aye, he thought so as well. Off he went into the countryside, travelling from ↵
 ↵village to village in search of a worthy opponent.

64 <<jump OldMan>>

-> Player: Couldn't he just demand that the people become skilled enough to fight him?
 66 <<update_player_model 40 0 0 0 0>>
 Companion: Such a demand would have led only to a revolt, I fear. Perhaps the Prince ↵
 ↵agreed, for off he went into the countryside, travelling from village to village in ↵
 ↵search of a worthy opponent.

68 <<jump OldMan>>
 ====

70 title: OldMan

72 Companion: One day, after a particularly unsuccessful village tour, Prince Vi'dal came across ↵
 ↵an old, crooked man **at** the side of the road.

Companion: "That I doubt", said the prince, "for I seek one who can match **either** my wits ↵
 ↵**or** my blade, **and** your mind seems as fuddled as your bones seem weak."

74 Companion: "You would be wise to **not** underestimate me," said the man, "but I have no ↵
 ↵quarrel with you. My offer stands: help me, **and** I will do what I can for you."

Companion: "If you offer no challenge," said the Prince, "then my time here is wasted." With ↵
 ↵that, Prince Vidal turned away **and** continued his journey along the road.

76 -> Player: He could have **at** least asked the old man what his **problem** was...
 <<update_player_model 0 0 100 0 0>>

78 <<jump MoralVidalStory>>

-> Player: From the old man's boasts, the Prince should have challenged him to a fight!
 80 <<update_player_model 100 0 0 0 0>>
 <<jump MoralVidalStory>>

82 -> Player: The Prince should have started a battle of wits—that old man seemed pretty ↵
 ↵shrewd to me.
 <<update_player_model 0 0 0 100 0>>

84 <<jump MoralVidalStory>>

-> Player: The old man should have reminded the Prince of his duty to his people.
 86 <<update_player_model 0 100 0 0 0>>
 <<jump MoralVidalStory>>

88 -> Player: The Prince should have helped the man – there might have been a big reward!
 <<update_player_model 0 0 0 0 100>>

90 <<jump MoralVidalStory>>
 ====

92 title: MoralVidalStory

```

----
94 Companion: Right you may be, my <boy/girl>, but alas, he did not, and we may never know ↵
    ↵what that poor old man had to offer.
Companion: Now tell me, <FirstName>: what can we learn from Prince Vi'dal's choice? Is ↵
    ↵there a moral to this story?
96 -> Player: "Personal ambition should not blind us to the needs of others."
    <<update_player_model 0 100 0 0 0>>
98 Companion: Well done indeed! Prince Vi'dal's ambition may well have cost him months of ↵
    ↵time spent searching the land.
    <<jump EndOfLesson>>
100 -> Player: "Strangers on the road may reward us for our help."
    <<update_player_model 0 0 0 0 100>>
102 Companion: True, true. Even old men tend to have a little gold on hand to help get ↵
    ↵themselves out of a bind.
    <<jump EndOfLesson>>
104 -> Player: "Sometimes even a prince needs a crack on the head."
    <<update_player_model 100 0 0 0 0>>
106 Companion: Sometimes indeed! I just wouldn't recommend trying it yourself.
    <<jump EndOfLesson>>
108 -> Player: "When care is taken, problems may be solved in unexpected ways."
    <<update_player_model 0 0 0 100 0>>
110 Companion: Indeed they can! Prince Vidal may have even found the old man's problem ↵
    ↵worthy of his skills.
    <<jump EndOfLesson>>
112 ===
    title: EndOfLesson
114 ----
Companion: Let's end today's lesson there – I must be off to meet with the village elders. The ↵
    ↵rest of the day is yours, <FirstName>.
116 Player: Thank you, father.
    ===

```

Listing A.1: Yarn Spinner conversation.

Appendix B

Scene Declaration

```
1 {
2
3   "name": "Second",
4   "description": "Mercy encounter",
5   "metadata": {
6     "target-model": {
7       "fighter": 1,
8       "method_actor": 0,
9       "storyteller": 4,
10      "tactician": 0,
11      "power_gamer": 0
12    }
13  },
14  "preconditions": "(and (in arnell City) (in bob City) (alive bob) (alive arnell))",
15  "instructions": [
16    {
17      "type": "Camelot",
18      "commands": [
19        "DisableInput()"
20      ]
21    },
22    {
23      "type": "PDDL",
24      "commands": [
25        "move-within-location(arnell, bob, City)"
26      ]
27    },
28    {
29      "type": "Camelot",
30      "commands": [
31        "Face(arnell, bob)",
32        "Dance(arnell)"
33      ]
34    },
35    {
36      "type": "PDDL",
37      "commands": [
38        "start_conversation(arnell, mrcy_callgiver)",
39        "move-within-location(arnell, City.Bench, City)"
40      ]
41    },
42    {
43      "type": "Camelot",
44      "commands": [
45        "EnableInput()"
46      ]
47    }
48  ]
49 }
```

```
48 } ]
```

Listing B.1: Example of a desclaration of a Scene in the Scene Controller module of the Camelot Wrapper.