Proceedings of the Eighteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment
(AIIDE 2022)

EM-Glue: A Platform for Decoupling Experience Managers and Environments

Giulio Mori,[1] **David Thue,**[1,2] **Stephan Schiffel**[1]

[1]Department of Computer Science, Reykjavik University, Menntavegur 1, 102, Reykjavik, Iceland
[2]School of Information Technology, Carleton University, 1125 Colonel By Drive, Ottawa, ON, K1S 5B6, Canada
giulio17@ru.is, david.thue@carleton.ca, stephans@ru.is

Abstract

Experience Management uses AI technologies to improve people's experiences within an interactive application by changing the environment while the experience is underway. Game-related research in this field has a trend where each experience manager is built in a way that is tightly integrated with the environment that it can change. One consequence of this integration is that it becomes difficult to compare one manager to another in a single environment, or a single manager to itself across multiple environments. With this paper, we propose a solution for decoupling experience managers from the environments that they can change, through the use of an intermediate software platform. We describe the structure of the platform, a protocol that facilitates communication between a manager and an environment, and how normal communication happens. Moreover, we introduce the *Camelot Wrapper*, software built to extend the interactive visualization engine *Camelot* and connect it to our platform.

1 Introduction

Experience Management is a subfield of Artificial Intelligence (AI) that studies intelligent systems with the name of "experience managers" (EMs), which improve people's experiences within an application according to one or more given metrics. Riedl et al. (2008) first defined the term "Experience Management". However, we describe it as Thue did: the process of optimizing a player's experience in an interactive environment (e.g., a game) by changing that environment while the experience is underway (Thue 2015). An *environment* is described by three components: a collection of states that players can observe, a set of actions that players can take, and dynamics that explain how player actions in each state of the environment lead to new states arising.

Unfortunately, it is common among researchers to design and develop EMs that are built directly into the games that are used to evaluate them (Weyhrauch 1997; Mateas and Stern 2003; Nelson et al. 2006; Riedl et al. 2008; Thue et al. 2007, 2011; Porteous, Cavazza, and Charles 2010; Ramirez and Bulitko 2015; Robertson and Young 2019). The result is that research in the field is fragmented and largely independent, making it hard to draw conclusions that generalize well across the field (Roberts and Isbell 2008; Mori, Thue, and Schiffel 2019). We identify three sub-problems.

First, since the results of testing EMs depend on the context of the environment used (Roberts and Isbell 2008), 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. The platform that we present in this paper simplifies the process of connecting an EM to different environments, thus enabling the data gathering needed to gain insights into how the EM works across different contexts.

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. An example is Ramirez and Bulitko's (2015) work, where they had to re-implement the EMs of both the *Automated Story Director* (Riedl et al. 2008) and *PaSSAGE* (Thue et al. 2007) 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. Our platform allows new EMs to be created in a way that keeps them separate from the environment(s) in which they are tested, avoiding the work of separating and reconnecting that we explained above. By simplifying the process of comparing multiple EMs, we aim to make such comparisons more widely accessible and common.

Third, another barrier to comparing multiple EMs is that very few are released after the publication of their related research papers. When an EM is made to use our platform, it becomes easier for later researchers to conduct experiments using that EM (potentially in new environments), and we hope that this potential for future comparisons will encourage more researchers to release their EMs.

In this work, we present *EM-Glue*, a software architecture and platform that decouples EMs from environments while facilitating communication between them. The platform enables the exchange of messages in a standardized way so that

different EMs and environments can work together using a common architecture. We designed a communication protocol with two phases: initialization and regular communication. The initialization starts with a handshake, which sets up the communication so that both parts can establish a working exchange and have the data they need. After the initialization process, the protocol manages the exchange of messages between the EM and the environment. Finally, we present the *Camelot Wrapper*, software that extends the functionality of *Camelot* (Shirvani and Ware 2020) to create the first environment that can communicate with our platform.

2 Related Work

Roberts and Isbell (2008) performed a qualitative analysis of different approaches to drama management (a kind of experience management focused on drama) based on a list of desiderata. In their work, they shed light on two problems with how drama managers are evaluated: dependency to the content, and the tight integration between drama managers and games. Our platform aims to address both problems. First, we provide software that enables experience management with detached EMs and environments, avoiding any tight integration between the two. Second, our platform supports evaluating an EM in multiple environments, which mitigates its dependency on the context during evaluation.

One of the first attempts to separate EMs and game environments was *Mimesis* (Young et al. 2004). 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. Our 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. 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; and *game-specific*, for when an intelligent agent has been developed to work into a specific game engine. Our 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. Our platform requires that the EMs and environments use an explicit declarative model of actions (see Section 3 for more details).

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. The functionalities are divided into modules that are independent software components that communicate with the platform via a set of messages. Our approach differs from theirs because *OPARIS* targets Interactive Storytelling systems and it focuses on narratives, whereas our platform focuses on EMs and aims to remain agnostic about the content of each environment.

One work that can be used to build an environment for our platform is *Camelot* (Samuel et al. 2018; Shirvani and Ware 2020), 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 our platform, we 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).

An example of a system that could be used as an EM with our platform was made by Porteous, Cavazza, and Charles (2010), in which a PDDL-based narrative replanner is used to produce multiple variants of narratives and control story pacing. Connecting their system to our platform could be achieved by decoupling it from its integrated 3D visualization engine and implementing our communication protocol. However, they use PDDL 3.0 in their system while we use only PDDL in the current version of our platform. As we mention in Section 6, we aim to support newer versions of PDDL in a later version of the platform.

3 Overview of the Platform

We aim to facilitate communication between an EM and an environment. To do so, we need them to agree on a protocol

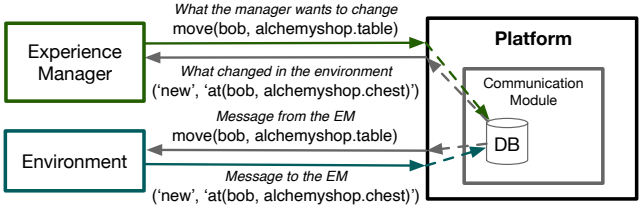

Figure 1: An overview of our platform's design. The Experience Manager (EM) and the Environment are external modules. The Environment sends a tuple to update the EM with what is happening in the environment and the EM sends the action that it wants to apply.

for communication. For this reason, we designed an architecture that can be used as a middle layer between the two, which routes the messages from one side to the other using a set of common APIs. The platform is open source and available on GitHub[1]. Figure 1 shows a high level overview of how the platform is designed.

The design has two main parts: external modules and the platform itself. The external modules include EMs and environments. Examples of the things that an EM can do include placing an item in a chest or changing a Non-Player Character (NPC). The environment has two goals: visualize the experience while allowing interaction with the user, and keep track of what is happening in the experience using a high level state representation. The state representation is important because the messages that are exchanged through the platform describe how the state changes after each event happens in the environment.

The platform contains a module that facilitates communication between EMs and environments. 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. While facilitating 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. We describe the design and development of the communication module in Section 4.

For EMs and environments to understand each other, a common language is needed. The messages that are exchanged need to contain a high level overview of what is happening in the environment, so that the EM can interpret a message and understand the current state to act upon it. The EM also needs to have a set of possible actions to choose from, described with any conditions that need to be met for correct execution as well as the effects that each action has on the state once they are executed. The literature offers many different languages that can be used to represent these aspects of the experience (McDermott et al. 1998; Love et al. 2008; Martens 2015; Perez-Liebana et al. 2016; Chen and Guy 2018; Ware et al. 2022), each of them solving problems of representation that other languages have.

For this platform, we decided to use the Planning Domain Definition Language (PDDL) (McDermott et al. 1998). PDDL was designed for describing problems in the field of automated planning, which is notably different from our context. Nevertheless, PDDL is well-known across the field and it has been used with EMs in the past (Porteous, Cavazza, and Charles 2010; Thue et al. 2016; Diamanti and Thue 2019; Porteous et al. 2021). Moreover, learning PDDL is supported by many online resources, which helps to maintain the accessibility of our platform. In Section 6, we discuss the limitations of using PDDL in our setting. Since the design of the platform's software and communication protocol are independent from the language that the EMs and environments use, it would be straightforward to exchange PDDL with another language. We are open to collaborating with the community to find another specification language that might be better suited for this application.

[1]https://github.com/liogiu2/EM-Glue

4 Communication Module

The purpose of the communication module is to facilitate the conversation between EMs and environments. It is composed of two main parts: the software structure and the communication handshake protocol.

To begin developing this module, we surveyed the space of possible technologies for communication, prioritizing easy inclusion of new external modules like EMs and environments. We first analyzed the standard input/output communication method, which is used by Camelot (Shirvani and Ware 2020). We found that this technique does not work well with concurrent requests. For example, if we need to write a message at the same time that we are waiting for a message to arrive, we become stuck in a deadlock situation. This is a problem that we faced while developing the Camelot Wrapper, as we discuss in Section 5.

We also analyzed sockets, the fundamental technology that enables network communication. An important downside of using sockets is that the learning curve is steep. We wanted 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, we decided to use a modern evolution of sockets that has become commonly used: web APIs (Tan et al. 2016). One of the most used Python libraries to implement web APIs is FastAPI (Ramírez 2022). It is straightforward to implement in a codebase and it 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.

The exchange of messages using the web API works as follows. The platform hosts a server where the functionality of the API operates, and it is constantly waiting for requests to activate that functionality. 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 content of the message to the body of the HTTP request. 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. 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 as a response to the most recent of such requests.

Errors have a different path compared to normal messages, because if an error occurs in the environment it might be of such gravity that the experience breaks. So, if an error occurs, the EM might need to take immediate action to solve it. In the case of Camelot and our wrapper (which we discuss

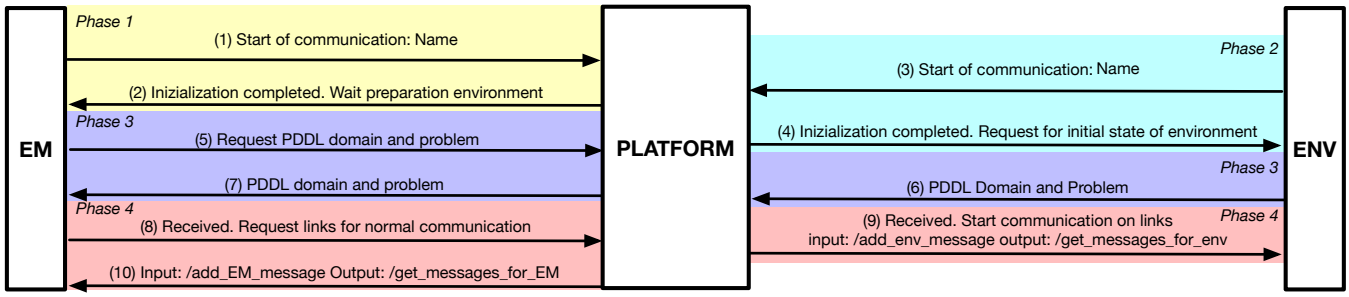

Figure 2: A summary of the steps of our platform's handshake protocol. (#) indicates the order of the steps of the protocol, where each step involves one message. Each coloured area represents a phase of the protocol. "EM" stands for experience manager and "ENV" stands for environment.

in Section 5), we attempt to avoid errors by validating every message before sending it to the environment.

One drawback of web 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. Fortunately, resolving this problem is relatively straightforward; 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). 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 version of the platform, we decided to not support this solution because it would involve the use of multi-threading in the client to obtain a smooth operation. Our goal is to keep the engineering of new managers as simple as possible, and multi-threading would add complexity to that work. However, in a later version of the platform, we plan to support both options so that developers can choose to use what they prefer.

4.1 Handshake Protocol

Our platform's handshake protocol is a standardized set of messages that connect an EM and an environment to each other through the platform. It was inspired by the TCP/IP three-way handshake protocol (Feit 2000). Figure 2 shows a diagram of the steps that are needed to successfully initialize the communication between an EM and an environment.

This handshake protocol involves the EM, platform, and environment simultaneously. It is divided in four phases, and each phase is used to handle a different part of the initial communication. It starts with *phase 1* assuming that the platform is up and running, because we need the API server to be ready to receive new requests in the initialization link for the EM (*/inizialization_em*). Once the the platform is ready, it launches the EM process automatically. The platform knows which EM and environment to use based on a JSON file (Pezoa et al. 2016) that holds the paths for their executables.

When started, the EM creates a request to the initialization link with the name of the EM as a parameter (step 1). After receiving the initial message from the EM, the platform adds the received data to the database (DB). The platform reads the message and replies to this request with the confirmation that the initialization is completed, and that it is waiting for the environment to be ready (step 2). In the meantime, the platform starts the environment and waits for a request to arrive from the environment to start *phase 2*. When started, the environment sends a new request on the environment's initialization link (*/inizialization_env*) with the parameter representing its name (step 3). The platform 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). Then, the EM prepares a request on the link (*/inizialization_em*) to start *phase 3* and sends a message to request the PDDL domain and problem (step 5). The environment creates a request on the URL (*/inizialization_env*) to reply to the previous request of the platform by providing a domain and initial state, attaching the PDDL domain and problem (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 (*/inizialization_em*) to confirm that it correctly received the PDDL data and asks for the URLs for normal communication (step 8). The platform then confirms the correct receipt of the data to the environment and sends the URLs for normal communication (step 9). Finally, the platform replies to the EM by sending the normal communication URLs to it as well (step 10).

We designed this handshake protocol in a way that accounts for the drawback of this type of web APIs that we discussed earlier in Section 4. Specifically, the external modules must always make the first request, to allow the platform to respond with the data that is needed. In addition, another aspect that we considered during the design process is that it can easily be adapted to languages other than PDDL. This allows the platform to be updated later with a different language for representing experience management tasks, which helps to future-proof the protocol.

In practice, the order of phases 1 and 2 could be interchangeable. We decided that the EM should go first to allow the EM or the platform choose which environment to use, when more than one environment is available.

4.2 Normal Communication

During normal communication between an EM and an environment, two types of messages are exchanged.

First, each message from an EM to an environment represents an action that the environment must execute. 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 that comes from the domain. For example, in "openfurniture(bob, alchemyshop.chest)", "openfurniture" is the name of the action and "bob" and "alchemyshop.chest" are the entities involved in the action.

The second type of message is one that goes from an environment to an EM. This message describes all of the updates that happen to the world state of the environment, and is 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 or a new entity. An example of such a tuple is: ('`new`', 'at(bob, alchemyshop.chest)'). The first element can have one of three 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)[2], 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 name of the entity. This message tells the EM that the `new_entity` action was executed with success. Player actions are not directly reported; the environment only communicates the relations that change while the game is being played. However, we are planning to add a message type to report player behaviour explicitly in the future, as we discuss in Section 6.

5 Camelot Wrapper

We used *Camelot* (Samuel et al. 2018; Shirvani and Ware 2020) to create an environment for the first prototype of our platform. For an environment to work with our platform, it should have the following characteristics: 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, and 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). To the best of our knowledge, Camelot is the only visualization engine that meets these requirements while also being able to connect to multiple different EMs. However, we could not use Camelot as-is for three reasons.

First, as introduced in Section 2, Camelot requires a highly specific set of instructions to work, while our platform uses PDDL to exchange information between each environment and EM to increase its generality. We thus needed a way to translate between Camelot instructions and our platform's PDDL-based instructions.

Second, Camelot requires almost everything that happens in a player's experience to be controlled by a connected EM; this includes responding to most player inputs (e.g., interacting with NPCs or objects) as well as moving and animating all NPCs; only fine-grained player movement is enabled directly by Camelot. In effect, the majority of a Camelot environment's dynamics must be implemented outside of Camelot itself. As a result, comparing two EMs in a single Camelot environment becomes difficult, as both EMs would need to consistently implement the same dynamics. It would also then be difficult to test those EMs in a different environment that implemented more of its own dynamics (e.g., a commercial role-playing game).

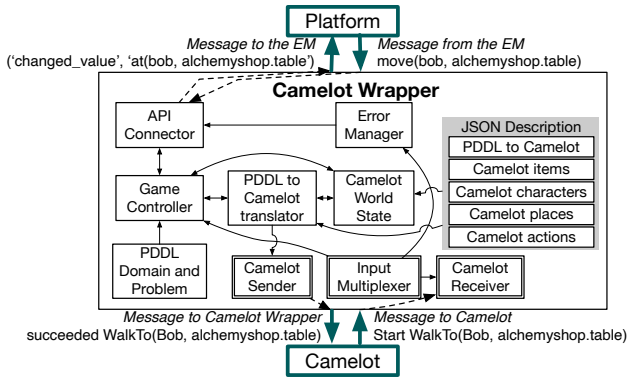

Figure 3: A schematic diagram of the Camelot Wrapper's main components. Double boxes indicates that the component is executed in a separate thread. The arrows indicate the flow of information between components.

The third reason is that Camelot is designed to not keep track of the state of the environment's world (Shirvani and Ware 2020). In our platform, we need the state of the environment to be shared with the EM using a common language (PDDL) to allow the managers to make decisions about what needs to happen in the environment.

To overcome these three concerns, we implemented a wrapper to act as a middle layer between the platform and Camelot. This wrapper is open source and available on GitHub[3]. It translates from PDDL instructions to Camelot instructions, it handles more of the environment's dynamics and low-level interactions than Camelot does on its own, and it keeps track of the current state of Camelot's world. Figure 3 describes the design of the *Camelot Wrapper*, which spans ten different components and is implemented in Python v3.9.1 (Van Rossum and Drake 2009). We briefly discuss each component in turn.

5.1 External Communication

We begin with the components that facilitate external communication: the API connector, the Camelot receiver thread, the Camelot sender thread, and the input multiplexer thread. When dealing with I/O operations, there is a problem of possibly long waits when standing by for new messages to come. For this reason, all of the components that deal with communication run on separate threads or processes. This

[2] A relation that is not explicitly recorded is assumed to be false.

[3] https://github.com/liogiu2/Camelot-Wrapper

choice adds complexity to the wrapper's code because it requires concurrency and queues for the exchange of messages, but the final result benefits from a smoother operation.

API Connector. As we introduced in Section 4, this component is a service that allows communication via HTTP with the web API that is running in the platform. It makes HTTP requests on the link provided with the communication protocol every 250 milliseconds to check for new messages. When it needs to pass a new message to the platform, it makes an HTTP request on the link for sending environment messages, and shares the changes to the world state.

Camelot Receiver and Sender Threads. Camelot communicates with EMs (or, in this case, our wrapper) via standard I/O. A problem with using this type of communication is that writing and reading are mutually exclusive. This means that if the wrapper is waiting for a message from Camelot to arrive, it cannot send any messages *to* Camelot, nor vice-versa, and this can result in deadlocks. One solution might be to interrupt the read operation after a certain amount of time, but this would require operating system level calls that not all operating systems support (e.g., Microsoft Windows does not allow it). Instead, our solution uses two threads and manages concurrency using thread locks and thread events.

Input Multiplexer Thread. The objective of this component is to sort the messages that come from Camelot into specific queues to handle each type of message in a different way. It can sort the messages into five categories: success, location, input, error, and other, according to the criteria listed in Appendix B. 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.

5.2 Translation

The platform uses six components to translate between PDDL and Camelot: the game controller, the PDDL Domain, the PDDL Problem, the Camelot world state, the PDDL to Camelot translator, and the error manager.

Game Controller. The game controller hosts the main loop of the Camelot Wrapper, and its job is to control 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. This process parses the PDDL domain and problem files and sends instructions to Camelot.

PDDL Domain. The domain file describes all of the possible PDDL actions that the game designer wants to allow to happen in the environment, along with the Camelot instructions that are necessary to realize each action. For example, if the game designer wants to have an action called `fight` (that is not currently in the available actions that Camelot allows), they can do so by including this action into the PDDL domain file and adding an entry into a JSON file (pddl_actions_to_camelot) 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). When an EM sends a message to execute the action `fight`, the PDDL to Camelot translator will look into the JSON file mentioned above, and the game controller will send Camelot all of the Camelot instructions that are specified for that action in the JSON.

PDDL Problem. The problem file defines the initial state of the Camelot environment. The initial state of the environment may be incomplete, in a sense, since we allow actions to create new entities as described in Section 4.2. The problem file will include the virtual locations that the environment requires, how they are connected (using the `adjacent` predicate), the characters and their attributes, and the objects in the world and their attributes.

The game controller's startup process continues by sending Camelot the instructions that were generated from the parsed initial state. These instructions are generated using the PDDL to Camelot translator component (described later in this section). During the initialization process, the wrapper also starts the communication with the platform using the protocol explained in Section 4.1. Once the startup process is finished, the game controller hosts the main loop of the Camelot Wrapper, where it executes the methods to handle different aspects of the game. For example, when it receives a success message from the input multiplexer, it applies the effects of an action to the Camelot world state.

Camelot World State. This component keeps track of the world state of the Camelot environment. It has methods that allow initializing the world state and applying an action to change the world state. To initialize the world state, it transforms the parsed PDDL domain and problem into a PDDL data framework that we developed (see Appendix A), which simplifies updating the world state as the truth values of PDDL relations change. When the Camelot world state component receives an action to apply from the PDDL to Camelot translator, it changes the relations that the "effect" part of the action says to change. This process happens only when a success message is received from Camelot that corresponds to an action that was sent. This way, if an action fails, we do not need to revert back to the previous state.

PDDL to Camelot Translator. This component translates from PDDL to Camelot instructions in two situations: to initialize the world state, and to convert PDDL actions to Camelot instructions. To initialize the world state, the translator receives a parsed PDDL domain and problem from the game controller, and it transforms the initial state specified in the problem into Camelot entities and objects. For example, if the PDDL problem file declares an "object" as `AlchemyShop - location`, it translates this declaration into the Camelot instruction `CreatePlace(alchemyshop, AlchemyShop)`. It does this transformation both for the "objects" and "init" part of the PDDL problem file.

The translation from PDDL actions to Camelot instructions is facilitated by a JSON-based dictionary (Pezoa et al. 2016). Each key in the dictionary is the name of a PDDL action that can be applied, and each value

is a list of commands that are the Camelot instructions (formatted as `action_name` and `action_args`) that should be executed when the PDDL action is applied. For example, suppose that an EM asks to apply the PDDL action `openfurniture`. The JSON entry for the key `openfurniture` lists the following Camelot Actions: `"OpenFurniture"`, `"DisableIcon"` and `"EnableIcon"`, which open the furniture and configure the player interface accordingly. So, when an EM asks for `openfurniture` to be executed, the Camelot Wrapper sends Camelot the instructions to open the furniture and to change the interface text and icon from "open" to "close".

Error Manager. Finally, the error manager is the component that handles any errors that come 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. Moreover, the error manager handles errors that come from the Camelot Wrapper if it intercepts minor problems that do not require EM intervention to be solved. For example, if there is a letter case mismatch in an entity part of a relation, the error manager automatically corrects this problem.

6 Discussion and Future Work

We developed the software presented in this paper as a way to explore the challenges involved in decoupling EMs from environments and to test the feasibility of our solutions. As can be seen in the software itself, we developed a basic EM to verify the platform's functionality, connecting the EM to Camelot through our platform and the Camelot Wrapper. This verification offers the first evidence that our solutions are feasible, in that we can successfully set up and sustain functional communications between an EM and an environment with only loose integrations between them.

In future versions of the platform, we plan to support an increasing number of use cases that are not currently supported. One example is to allow the environment to provide the EM with arbitrary domain-specific information, sent alongside the updates of the world state. This addition would expand the potential set of EMs that could work with the platform, because environment authors could include supplementary metadata that an EM might use to guide its reasoning (e.g., to execute certain types of actions only when the main character has enemies).

Another potential addition includes support for narrative intervention (Riedl, Saretto, and Young 2003), 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.

We also plan to change how the environment reports player actions to experience managers. As described in Section 4.2, 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. In the next version of the platform, we plan to change this behaviour by adding a message that allows each environment to send player actions directly to the EM.

When choosing to use PDDL in its basic form, we were aware of its limitations, such as its inability to represent actions whose effects are long-lasting, poor support for numbers, and more. If we decide to use a different language, 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. We 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 (Fox and Long 2003; Gerevini and Long 2005), so it might be useful to update the Camelot Wrapper to support a later version of PDDL. We are also keen to collaborate in identifying or developing a language that would better suit the needs of experience management tasks.

Looking forward, we plan to test the platform further by connecting more EMs to the platform (either new or pre-existing) and integrating environments that target more diverse genres of games. Such tests will help us further explore the strengths and limits of our approach, and improve how we decouple experience managers from their environments.

A PDDL Framework

Figure 4 shows the structure of the PDDL data framework that we use to represent the state throughout the project with all the components needed. Most of the components are derived directly from the PDDL specification files, and they are described in the Planning Wiki (Adam et al. 2022).

B Criteria for Sorting Camelot Messages

The Camelot Wrapper's input multiplexer sorts Camelot messages based on the following criteria:

- A success message is one that starts with `succeeded`. Camelot sends this message when an action previously sent by the platform is executed with success.
- A location message is one that starts with `input` and continues with one of the following substrings: `started walking`, `stopped walking`,

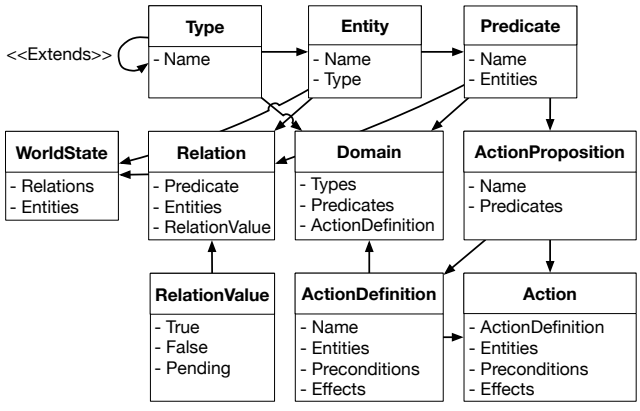

Figure 4: A diagram showing how we structured the PDDL data framework that we use to represent the state of the Camelot environment, within the Camelot Wrapper.

`arrived`, 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 is one that 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.

Acknowledgments

We are grateful to have received financial support for this work from both The Icelandic Centre for Research - Rannís (Project Grant #184937-053) and the Reykjavík University Research Fund.

References

Adam, G.; Benjamin Jacob, R.; Christian, M.; Enrico, S.; Felipe, M.; Francisco Martin, R.; Henry, S.; Jan, D.; Mau, M.; and Jonathan, M. 2022. Planning.Wiki - The AI Planning and PDDL Wiki. https://planning.wiki/ref/pddl. Accessed: 2022-08-15.

Chen, T.; and Guy, S. J. 2018. GIGL: A Domain Specific Language for Procedural Content Generation with Grammatical Representations. In *Proceedings of the 14th AIIDE*, 9–16. AAAI Press.

Diamanti, M.; and Thue, D. 2019. Automatic Abstraction and Refinement for Simulations with Adaptive Level of Detail. In *Proceedings of the 15th AIIDE*, 17–23. AAAI Press.

Feit, S. 2000. *TCP/IP*. USA: McGraw-Hill, Inc.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *CoRR*.

Gerevini, A.; and Long, D. 2005. Plan Constraints and Preferences in PDDL 3. The Language of the Fifth International Planning Competition. Technical report, Department of Electronics for Automation, University of Brescia, Italy.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2008. General game playing: Game description language specification. Technical report, Stanford.

Martens, C. 2015. Ceptre: A Language for Modeling Generative Interactive Systems. In *Procedings of the 11th AIIDE*, 51–57. AAAI Press.

Mateas, M.; and Stern, A. 2003. Façade: An experiment in building a fully-realized interactive drama. In *Game developers conference*, volume 2, 4–8.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL—The Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control.

Mori, G.; Thue, D.; and Schiffel, S. 2019. A Structured Analysis of Experience Management Techniques. In *Proceedings of the 15th AIIDE*, 174–180. AAAI Press.

Nelson, M. J.; Mateas, M.; Roberts, D. L.; and Isbell Jr., C. L. 2006. Declarative Optimization-Based Drama Management in Interactive Fiction. *IEEE Computer Graphics and Applications*, 26(3): 32–41.

Perez-Liebana, D.; Samothrakis, S.; Togelius, J.; Lucas, S. M.; and Schaul, T. 2016. General Video Game AI: Competition, Challenges, and Opportunities. In *Proceedings of the 30th AAAI*, AAAI'16, 4335–4337. AAAI Press.

Pezoa, F.; Reutter, J. L.; Suarez, F.; Ugarte, M.; and Vrgoč, D. 2016. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web*, 263–273. International WWW Conferences.

Porteous, J.; Cavazza, M.; and Charles, F. 2010. Applying Planning to Interactive Storytelling: Narrative Control Using State Constraints. *ACM Trans. on Intelligent Systems and Technology*, 1(2): 1–21.

Porteous, J.; Ferreira, J. F.; Lindsay, A.; and Cavazza, M. 2021. Automated narrative planning model extension. *Autonomous Agents and Multi-Agent Systems*, 35(2): 1–29.

Ramirez, A.; and Bulitko, V. 2015. Automated Planning and Player Modeling for Interactive Storytelling. *IEEE Transactions on Comput. Intell. and AI in Games*, 7(4): 375–386.

Ramírez, S. 2022. FastAPI. github.com/tiangolo/fastapi. Accessed: 2022-08-15.

Riedl, M.; Saretto, C. J.; and Young, R. M. 2003. Managing Interaction between Users and Agents in a Multi-Agent Storytelling Environment. In *Proceedings of the 2nd International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03, 741–748. New York, USA: ACM.

Riedl, M. O.; Stern, A.; Dini, D.; and Alderman, J. 2008. Dynamic experience management in virtual worlds for entertainment, education, and training. *International Transactions on Systems Science and Applications*, 4(2): 23–42.

Roberts, D. L.; and Isbell, C. L. 2008. A survey and qualitative analysis of recent advances in drama management. *Inter. Trans. on Systems Science and Applications*, 4(2): 61–75.

Robertson, J.; and Young, R. M. 2019. Perceptual experience management. *IEEE Trans. on Games*, 11(1): 15–24.

Samuel, B.; Reed, A.; Short, E.; Heck, S.; Robison, B.; Wright, L.; Soule, T.; Treanor, M.; McCoy, J.; Sullivan, A.; Shirvani, A.; Garcia, E. T.; Farrell, R.; Ware, S.; and Compton, K. 2018. Playable experiences at AIIDE 2018. In *Proceedings of the 14th AIIDE*, 275–280. AAAI Press.

Shirvani, A.; and Ware, S. G. 2020. Camelot: a modular customizable sandbox for visualizing interactive narratives. In *Proceedings of the 12th Intelligent Narrative Technologies workshop at the 16th AIIDE*, 8 pages, CEUR-WS, Vol-2862.

Szilas, N.; Boggini, T.; Axelrad, M.; Petta, P.; and Rank, S. 2011. Specification of an Open Architecture for Interactive Storytelling. In *Interactive Storytelling*, 330–333. Berlin, Heidelberg: Springer Berlin Heidelberg.

Tan, W.; Fan, Y.; Ghoneim, A.; Hossain, M. A.; and Dustdar, S. 2016. From the Service-Oriented Architecture to the Web API Economy. *IEEE Internet Computing*, 20(4): 64–68.

Thue, D.; Bulitko, V.; Spetch, M.; and Romanuik, T. 2011. A Computational Model of Perceived Agency in Video Games. In *Proceedings of the 7th AIIDE*, 91–96. AAAI Press.

Thue, D.; Bulitko, V.; Spetch, M.; and Wasylishen, E. 2007. Interactive Storytelling: A Player Modelling Approach. In *Proceedings of the 3rd AIIDE*, 43–48. AAAI Press.

Thue, D.; Schiffel, S.; Árnason, R. A.; Stefnisson, I. S.; and Steinarsson, B. 2016. Delayed Roles with Authorable Continuity in Plan-Based Interactive Storytelling. In *Interactive Storytelling*, 258–269. Springer International Publishing.

Thue, D. J. 2015. *Generalized Experience Management*. Ph.D. thesis, Department of Computing Science, University of Alberta, Canada.

Van Rossum, G.; and Drake, F. L. 2009. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace.

Ware, S. G.; Garcia, E. T.; Fisher, M.; Shirvani, A.; and Farrell, R. 2022. Multi-agent narrative experience management as story graph pruning. *IEEE Transactions on Games*. (forthcoming).

Weyhrauch, P. W. 1997. *Guiding interactive drama*. Ph.D. thesis, Carnegie Mellon University.

Young, R. M.; Riedl, M. O.; Branly, M.; Jhala, A.; Martin, R.; and Saretto, C. 2004. An architecture for integrating plan-based behavior generation with interactive game environments. *Journal of Game Development*, 1(1): 51–70.