DIEGESIS

A multi-agent Digital Interactive Storytelling framework using planning and re-planning techniques

Efstathios Goudoulakis

A thesis submitted in partial fulfilment of the requirements
of Liverpool John Moores University for the
degree of Doctor of Philosophy

June 2014

ABSTRACT

In recent years, the field of Digital Interactive Storytelling (DIS) has become very popular both in academic circles, as well as in the gaming industry, in which stories are becoming a unique selling point. Academic research on DIS focuses in the search for techniques that allow the creation of systems that can generate dynamically interesting stories which are not linear and can change dynamically at runtime as a consequence of a player's actions, therefore leading to different story endings.

To reach this goal, DIS systems usually employ Artificial Intelligence planning and re-planning algorithms as part of their solution. There is a lack of algorithms created specifically for DIS purposes since most DIS systems use generic algorithms, and they do not usually assess if and why a given algorithm is the best solution for their purposes. Additionally, there is no unified way (e.g. in the form of a selection of metrics) to evaluate such systems and algorithms.

To address these issues and to provide new solutions to the DIS field, we performed a review of related DIS systems and algorithms, and based on the critical analysis of that work we designed and implemented a novel multi-agent DIS framework called DIEGESIS, which includes –among other novel aspects- two new DIS-focused planning and re-planning algorithms.

To ensure that our framework and its algorithms have met the specifications we set, we created a large scale evaluation scenario which models the story of Troy, derived from Homer's epic poem, "Iliad", which we used to perform a number of evaluations based on metrics that we chose and we consider valuable for the DIS field. This collection of requirements and evaluations could be used in the future from other DIS systems as a unified test-bed for analysis and evaluation of such systems.

ACKNOWLEDGEMENTS

This work is dedicated to my twin daughters, Evdokia and Myranda, who were born a few months before I completed my research.

Firstly, I would like to thank my primary supervisor Professor Abdennour El Rhalibi for his continuous support, guidance, encouragement, and inspiration throughout my research which were crucial for the completion of it, as well as my second and third supervisors, Professor Madjid Merabti and Professor A. Taleb-Bendiab.

I would also like to thank my parents, Christos and Evdokia, my partner, Filian, as well as the rest of my family (with a special mention to my grandmother Eftichia and my uncle Vasilis who passed away last summer and were always proud of me pursuing a PhD) for the support, patience, and encouragement that they showed me both during the course of my research, and during my whole life.

Finally, I would like to thank Liverpool John Moores University for the scholarship that made this research possible and, last but not least, all my friends (whose list of names is lengthy to mention here) for their support and companionship.

DECLARATION

I declare that as the sole author of this doctoral thesis, the work contained herein is my own, unless explicitly stated otherwise. During the course of my research, the following research publications were produced to disseminate my work. Certain material and concepts from these publications will, by necessity, be presented within the context of this thesis.

- Goudoulakis, E., El Rhalibi, A., Merabti, M., and Taleb-Bendiab, A. 2011. Evaluation of Planning Algorithms for Digital Interactive Storytelling. *The 12th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2011)*.
- El Rhalibi, A., Goudoulakis, E., and Merabti, M. 2012. DIS Planning Algorithms Evaluation. *4th IEEE International Workshop on Digital Entertainment, Networked Virtual Environments, and Creative Technology*.
- Goudoulakis, E., El Rhalibi, A., Merabti, M., and Taleb-Bendiab, A. 2012. Framework for multi-agent planning and coordination in DIS. *Proceedings of the Workshop at SIGGRAPH Asia (WASA '12)*.
- Goudoulakis, E., El Rhalibi, A., Merabti, M., and Taleb-Bendiab, A. 2012. Opportunistic Multi-Agent Digital Interactive Storytelling System. *The 13th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2012)*.
- Duarte, R., Goudoulakis, E., El Rhalibi, A., and Merabti, M. 2013. A Conversational Avatar Framework of Digital Interactive Storytelling. *The 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2013)*.
- Goudoulakis, E., El Rhalibi, A., Merabti, M., and Taleb-Bendiab, A. 2013. Re-planning in Digital Interactive Storytelling. *The 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2013)*.
- Goudoulakis, E., El Rhalibi, A., Merabti, M., and Taleb-Bendiab, A. 2014. DIEGESIS – A Novel Multi-Agent Planning System for Digital Interactive

Storytelling. *ACM Journal Computers in Entertainment*, Issue 12, Vol. 3, September 2014 (in print).

TABLE OF CONTENTS

List of Figures

LIST OF TABLES

List of Abbreviations

AI – Artificial Intelligence

AM – Agents' Manager

BM – Battle Manager

CM – Choices Manager

DIS – Digital Interactive Storytelling

FGM – Futile Goals Manager

GIM – Goal Injection Manager

GUI – Graphical User Interface

HTN – Hierarchical Task Network

KB – Knowledge Base

LM – Level Manager

MAS – Multi-Agent System

NPC – Non-Player Character

OG – Output Generator

PDDL – Planning Domain Definition Language

STRIPS – Stanford Research Institute Problem Solver

TiM – Time Manager

TM – Transitioning Manager

UAM – Uncertain Actions Manager

UM – User Manager

VPM – Vantage Point Manager

WM – World Manager

1 INTRODUCTION

1.1. MOTIVATION

In recent years, the field of Digital Interactive Storytelling (DIS) has become very popular both in academic circles and in the gaming industry. The latter is a prosperous industry, since it has surpassed in revenue the music and movies industries. Stories and storytelling are becoming more important in games (for example in Role Playing Games) and are transforming into a unique selling point of them.

DIS systems can also be used in education and also in other entertainment areas (apart from games) such as in TV, movies, series, etc. For example, in the future the viewer may be able to interact and change the outcome of a story presented to her via animation. Such systems can also generate scripts for movies/series to help writers with ideas.

DIS is a very flourishing research area in academia and is a platform which allows us to do a multi-disciplinary research containing interesting and exciting areas to work on, such as multi-agent systems, planning, re-planning, etc., solving difficult problems (for example regarding real-time performance).

Also, we get the chance to combine all these disciplines and apply and use them in a different area from where they were traditionally used. For example, there is extensive research for planning algorithms with a focus on industrial themes, but not much in the DIS field.

Academic research on DIS focuses in the search for techniques that allow the creation of systems that can generate dynamically interesting stories which are not linear and that can change dynamically during runtime as a consequence of a player's actions, therefore leading to different story endings.

There have been numerous approaches attempting to reach this goal by employing Artificial Intelligence (AI) planning algorithms as part of the solution, but they typically do not discuss in detail or assess if and why a given planning algorithm was used and if that algorithm is the best solution for the story scenarios used or if it is suitable for other DIS scenarios. Also, there is a gap in the development and use of re-planning methods in existing systems, as no specific re-planning algorithm has been proposed to deal specifically with DIS. In this research we propose to contribute to the assessment of existing AI planning solutions for DIS, to create a novel multi-agent framework that provides new solutions for DIS, and to design new AI planning re-planning algorithms which will have been evaluated to be the most suitable for DIS characteristics.

1.2. Aim & Objectives

The proposed research aims at investigating AI planning and re-planning algorithms and exploiting their potential for the field of DIS, to evaluate their suitability for such systems and to develop new algorithms to improve them. To this end and to also provide more solutions to DIS research, a multi-agent DIS framework using planning and re-planning techniques will be specified, designed, implemented, and evaluated using appropriate DIS scenarios.

The objectives of this research work will be to consolidate the knowledge related to existing planning and re-planning algorithms for DIS, and develop a more generic multi-agent DIS framework providing a more robust, flexible and performant solution for a large class of DIS. In particular we will:

- Review the related work in DIS systems, as well as planning and re-planning algorithms related to DIS;
- specify the requirements of a multi-agent DIS framework which uses planning and re-planning techniques;
- design and implement a novel multi-agent DIS framework which utilises planning and re-planning techniques to generate a narrative;
- introduce mechanisms to generate different story outcomes and perspectives, for example using choices, goal injections, levels transitioning, and vantage points;

- design and implement a new planning algorithm for DIS, taking into account its performance and impact on the storyline quality;
- design and implement a new re-planning algorithm for DIS, taking into account its performance, minimal disruption to the original plan, and impact on the storyline quality;
- define the evaluation criteria, create evaluation scenarios, and evaluate the implemented framework and algorithms;
- exploit the research outcomes for generalisation and dissemination purposes.

1.3. Research Methodology

In order to achieve the research objectives that we outlined previously, the research methodology that we used during the course of our research follows an incremental and iterative model. In each iteration, starting with an initial idea, we performed a (never-ending) literature survey on the related work to identify problems in one (or more) of our fields of research. By analysing the data derived from this survey, we designed new and/or modified existing components of our framework, which we implemented and evaluated (when was required). The following sections discuss these steps in more details.

1.3.1. Literature Survey and Critical Analysis

In order to develop a deep and varied understanding in the fields related to our research, we performed a comprehensive survey on existing DIS systems, multi-agent systems, as well as planning and re-planning algorithms with the DIS field in mind, which we documented in chapter 2.

1.3.2. Problem Analysis and Design

Based on the critical analysis of our literature review, we specified the requirements and specifications of a scalable, abstract, interactive, and decoupled multi-agent DIS framework which includes dynamic story generation and narration, as well as different points of view.

To achieve these requirements and specifications we designed DIEGESIS, a multi-agent DIS framework using planning and re-planning techniques. DIEGESIS consists of several different components, each responsible for one or more features of the framework, such as the planner which includes a planning and a re-planning algorithm.

Although we used an incremental and iterative approach during the design or our framework and its components, for the sake of simplicity and clarity of the thesis we are documenting everything showing DIEGESIS as a final product. The requirements and specifications of our framework are documented in chapter 3, and its design in chapter 4.

1.3.3. Framework Implementation

While implementing our designed multi-agent DIS framework, including our planning and re-planning algorithms, we took an iterative prototyping approach. As soon as a part of a design was complete (e.g. a new component) we implemented it and then, when it was required, we designed and created an evaluation case to test and evaluate it. As soon as this process was complete, changes and refinements were made to the design (and therefore to the implementation) of our framework before starting this iterative process again. The implementation of our framework is documented in chapter 5.

1.3.4. Evaluation Assessment

As we mentioned in the previous section, during the course of our design and implementation phases, we performed a number of evaluations to evaluate several aspects of our framework, expose any limitations that it has, and ensure we were in course with the requirements and specifications we had set.

To aid us in these evaluations, we either used parts or the whole of the large-scale evaluation scenario we modelled, which includes several characters with rich relations between them, and a high number of possible actions and choices, that can provide different outcomes. This scenario along with other related information and our evaluations are documented in chapter 6.

1.4. Contributions to Knowledge

This thesis makes the following contributions to knowledge, in relation to the field of Digital Interactive Storytelling (DIS):

1.4.1. Critical Analysis Consolidation

Consolidating the knowledge of previous related work in the field of DIS, can help to identify techniques that provide good results and also to identify areas in the field which have not been thoroughly explored yet and in which we could provide a novel perspective.

Planning and re-planning techniques used in DIS systems are such an area, therefore we are presenting the results of our critical analysis and evaluation mostly focused on that, identifying (among other data) the types of planning and the planning algorithms the state of the art of related DIS systems use, as well as if they consider re-planning.

Parts of our literature survey have been published in the following papers: (Duarte et al., 2013, El Rhalibi et al., 2012, Goudoulakis et al., 2011, Goudoulakis et al., 2012b, Goudoulakis et al., 2013, Goudoulakis et al., 2012a, Goudoulakis et al., 2014)

1.4.2. Design of a Novel DIS Framework

DIEGESIS is a scalable, interactive, and modular DIS multi-agent framework, which includes dynamic story generation and narration, as well as different points of view. Most DIS systems use generic Artificial Intelligence (AI) planning algorithms which were not created specifically with DIS requirements in mind, and very few of them consider re-planning as part of their planners. DIEGESIS includes a new planner which consists of a planning and a re-planning algorithm created with the needs of DIS systems in mind.

Most DIS systems perform either centralised or decentralised planning; DIEGESIS follows a hybrid approach. On the plan generation level, it performs decentralised planning in which each character in a story is represented by an autonomous agent able to opportunistically generate plans based on its own goals. In the same manner, each agent tries to execute its own plans autonomously. We believe that this provides

a more flexible and realistic approach to the generation of a story, since each agent acts as a self-sufficient agent, generating an autonomous plan considering its own needs.

In the case of plans execution though, our approach borrows the control and coordination concepts from the centralised planning approach. Although the plans are individual, we want DIEGESIS to dictate the execution phase of the agent's plans (therefore the generation of the story) so the system can have a better control and understanding of what happens during the generation/execution of the story, and to be able to interfere if needed.

DIEGESIS' agent architecture, follows a hybrid approach; it includes elements of reactive agents (the agent receives input, processes it, and produces an output), elements of deliberative agents (the agent keeps an internal view of its environment), and elements of BDI agents (Beliefs – the agent's view of the world, Desires – the agent's goals, Intentions – the agent's plans).

In terms of interactivity, while many DIS systems allow the end-user to control only one character in the story (i.e. the protagonist), in DIEGESIS, there is not a main character (i.e. agent) that the player controls/observes; instead, the player can make choices (defined by the person who creates the story, i.e. the storyteller) for actions that can affect every character in the active story.

Apart from choices which can have a huge impact on the outcome of the generated narrative, DIEGESIS implements other mechanics which can have an impact on the story as well: A goal injection mechanism can inject new goals to the agents based on situations that occur during the generation of the story; a battle mechanism is able to calculate the outcome of both duels between agents and/or non-player characters (NPCs), and large-scale battles between large armies of NPCs; a futile goals module is able to assign goals to agents which are idle; and the concept of uncertain actions (actions which have a chance of succeeding or failing) have the potential to delay agents' plans from being successfully executed, or even invalidate them.

Traditionally, related DIS systems use either a first-person or a third-person perspective to present their stories to the player. In its default mode, DIEGESIS

presents the generated story as a whole, allowing the player to observe and interact (when is required) with any of the agents present in the story. These abilities constitute a third-person perspective, but since we want to provide the player with a first-person perspective as well, we created the concept of vantage points. If the player selects to view the story from the vantage point of an agent she will view only the story outcome which is related to the chosen agent, and will be available to interact with the story (i.e. make choices) only when an action is related to the story agent. The rest of the story (which is unrelated to the selected agent) will continue normally in the background. The player is able to choose between different vantage points or return to a full story view freely during run-time. All these mechanics are allowing linear storyline with differing endings, interleaved storylines, and even flashbacks.

Finally, DIEGESIS uses a hybrid story modelling approach, combining both plot-based and character-based elements. The game world is organised in multiple relatively abstract levels which can represent possible parts of a story. DIEGESIS is able to transfer knowledge between levels (acquired by previously executed levels), and judiciously choose which level needs to be executed next to form a valid and interesting story, based on a level transitioning system. Using this plot-based approach, DIEGESIS always has a high-level control over the overall structure of the story, being able to transition the story between levels which make sense, producing a coherent narrative.

But, when a level is loaded to be executed, we move closely to a character-based model; each agent may have some initial intentions/desires, but is able to operate autonomously and opportunistically to achieve its goals. The framework won't interfere with the decisions of an agent even if they mean that the story cannot progress any further. The authoring process in DIEGESIS provides enough freedom to the storyteller to operate whichever way she wants; either to create a relatively rigid storyline without much room for highly diverse narratives, or to model a story in a way that everything is fluid; a lot of player choices, several potential goal injections based on actions that may occur, and several uncertain actions; all of these features can contribute to unexpected situations and more emergent narratives.

Parts of the DIEGESIS' architecture have been described in the following published papers: (Duarte et al., 2013, Goudoulakis et al., 2012b, Goudoulakis et al., 2013, Goudoulakis et al., 2012a, Goudoulakis et al., 2014)

1.4.3. Implementation of a Novel DIS Framework

The aforementioned design of DIEGESIS is fully implemented into a full working multi-agent DIS framework. This implementation provides us with a framework for the creation and evaluation of our new planning and re-planning algorithms, as well as enabling us to provide an accurate evaluation of them. That also gives us the opportunity to use the framework in conjunction with other systems in the future, creating new expanded DIS solutions.

1.4.4. A New Planning Solution for DIS Frameworks

Most DIS systems use generic AI planning algorithms which were not created specifically with DIS requirements in mind. A few DIS systems have created adaptations of planning algorithms for their needs, but they don't provide many details about their mechanics.

DIEGESIS includes a new planning solution created based on the needs of the DIS field, able to generate plans of actions based on each agent's state and context, considering both the current world state and the available resources.

The planning algorithm is based on Graphplan (expanded to include support for several language requirements that we consider valuable for DIS) for solutions expansion, and a backtracking heuristic search for plan extraction, enriched with constraints satisfaction and dynamic opportunistic restart when required. The planning algorithm is also aware of the available time (duration) an agent/character has for a plan when it is asked to generate one.

The expansion stage allows the generation of all the sub-goals compatible with the current constraints, while the plan extraction involves a search technique using appropriate heuristics to link the goal(s) to the initial state and generate a valid plan.

Details about our planning solution have been published in the following papers: (Goudoulakis et al., 2012b, Goudoulakis et al., 2012a, Goudoulakis et al., 2014)

1.4.5. A New Re-planning Solution for DIS

When we consider classical AI planning for DIS, one of the premises is that the environment is static, which means that the planner is the only agent that can make changes in the story environment. However, a more realistic proposal is that the environment is dynamic; that is, there are other agents in the story and the actions generated by the planner may fail due to the actions of these agents.

We believe that a key aspect in the use of planning formalisms in DIS consists in their ability to support re-planning and to offer representations embedding the potential for failure. However, research for re-planning in DIS is sporadic. There are some DIS systems that claim to use re-planning approaches, but the information they provide is scarce. Most planners solve each planning task from scratch, which is time consuming.

DIEGESIS deals with the execution of the agents' plans in a higher level, and when a part of a plan fails, instructs the agent to re-plan based on its current knowledge of the state of the world. Considering that we modelled each agent to act as a real person in the way they generate and try to execute a plan, it does not make sense (in our context) to predict and prevent plan failures as some related DIS systems do, since a plan can fail either due to user intervention (which cannot be predicted), or intervention by other characters, or –in some cases– pure chance. In any case, failed plans due to "unpredicted" reasons are realistic and have the potential to enrich a generated narrative.

In our re-planning solution, as we interleave plan generation and plan execution, when a plan fails, we discard the already completed actions and we only re-plan for the failed (and some of the pending) actions of the plan, merging the new partial plan with the unexecuted portion of the original plan.

Details about our re-planning solution have been published in the following papers: (Goudoulakis et al., 2013, Goudoulakis et al., 2014)

1.4.6. Evaluation of DIS Systems

DIEGESIS has to be evaluated with experiments that provide evidence in support of our thesis and emphasise either the proof-of-concept (i.e. demonstrating the validity of a technique) or efficiency (i.e. demonstrating that a technique provides better performance than those that exist), depending on the evaluated component's role in the overall framework.

There are no widely accepted metrics to evaluate DIS systems that we could use, so we had to specify some of them, based on what we consider valuable for the DIS field. This collection of requirements and evaluations could be used in the future from other DIS systems as a unified test-bed for analysis and evaluation of such systems.

The outcome of our work is evaluated for the following requirements that we consider important for DIS frameworks:

- **Performance of planning and re-planning solutions**: Many related DIS systems reported that their planning and re-planning solutions suffered from performance issues, making the planning and re-planning expensive in any sizable domain. Our planner needs to have a good performance in order to generate (and regenerate) plans in real-time. To this end, we designed and performed a number of evaluations to identify potential bottlenecks of our planning and re-planning solutions, and explore their suitability for our DIS needs.
- **Suitability of planning algorithms' features for DIS**: Apart from performance, a planning algorithm should possess a number of features that we consider valuable for the DIS field. We performed an evaluation to identify which of the existing planning algorithms are suitable to be used in the DIS field, to be used as a base algorithm of our planning solution.
- **Performance-based interactivity of the framework**: Any DIS system and framework should support some kind of interactivity. To this end, the framework's performance should be adequate enough so the player will not suffer from delays causing the framework to be potentially unusable and possibly frustrating to use. To evaluate the performance-based interactivity of

DIEGESIS, we designed and performed an evaluation measuring the framework's level loading and turn execution times during the generation and execution of a large-scale scenario.

- **Summarisation metrics**: Although their types can vary in different DIS systems and frameworks, in most there should be some data which can quantify the complexity of a generated story. We performed an evaluation measuring several metrics of some storylines generated by DIEGESIS, such as the volume of levels, characters, turns, actions, potential nodes, etc.

To perform the aforementioned evaluations, we created a large scale evaluation scenario which models the story of Troy, derived from Homer's epic poem, "Iliad", which will be presented in section 6.1.

We have published evaluations of our framework and of relevant planning algorithms in the following papers: (El Rhalibi et al., 2012, Goudoulakis et al., 2011, Goudoulakis et al., 2012b, Goudoulakis et al., 2013, Goudoulakis et al., 2012a, Goudoulakis et al., 2014).

1.4.7. Proposal of Possible Applications

Apart of operating on its own, we are proposing two different possible applications which use DIEGESIS as a relying framework: an application which uses a 3D engine that will enable us to visualise the generated stories and improve the interactivity with the end-user; and a virtual storyteller application which interfaces our framework with a 3D character animation framework which will act as a narrator for the stories which our framework produces, using a natural language generation system as an intermediate, an application which we proposed in (Duarte et al., 2013).

1.4.8. Dissemination of Our Findings

The outcomes of our research have been disseminated via publishing a number of papers (Duarte et al., 2013, El Rhalibi et al., 2012, Goudoulakis et al., 2011, Goudoulakis et al., 2012b, Goudoulakis et al., 2013, Goudoulakis et al., 2012a, Goudoulakis et al., 2014), and our work has been presented at international conferences.

1.5. Structure of Thesis

The rest of the thesis is structured in the following way:

- In *Chapter 2 (Background & Related work)*, we present the background of our research area. More specifically, we discuss about the field of Digital Interactive Storytelling, about multi-agent systems presenting some agent architectures, and about DIS-related as well as multi-agent-related planning and re-planning. We also present some of the planning algorithms which are typically used in DIS systems along with some of the representation languages used by them, we present some examples of re-planning outside of the DIS field, and we survey and critically assess a number of DIS systems, stating their relation to our own work.
- In *Chapter 3 (DIEGESIS DIS Framework)*, we document the requirements and specifications of our multi-agent DIS framework.
- In *Chapter 4 (Design of the Framework)*, we discuss in detail the design aspect of every component of our framework.
- In *Chapter 5 (Implementation)*, we document all the details about the implementation of the multi-agent DIS framework we discussed in the previous chapter.
- In *Chapter 6 (Evaluation)*, we provide detailed information about the evaluation scenario that we modelled, showing its potential storylines, we discuss some of the mechanics that can have an impact on the generated story, and we specify the metrics used in our evaluations. We are also documenting a number of evaluations for the different components of our framework, using the evaluation scenario we presented earlier in the chapter.
- Finally, in *Chapter 7 (Conclusion & Future Work)*, we conclude this thesis and we document some future work ideas for our framework. We also describe some potential routes for our framework, utilising its capabilities via connecting it to other components and engines to allow us to create new DIS applications.

2 BACKGROUND & RELATED WORK

In this chapter we present the background of our research areas, which are the fields of Digital Interactive Storytelling (DIS), planning, re-planning, and multi-agent systems (MAS). Afterwards, we present some of the planning algorithms which are typically used in Digital Interactive Storytelling (DIS) systems along with some of the representation languages used by them, then some examples of re-planning outside the DIS field, and finally we survey and critically assess a number of DIS systems, stating their relation to our own work.

2.1. DIGITAL INTERACTIVE STORYTELLING

Video games for computers and consoles are established as the leading form of interactive digital entertainment (Barros and Musse, 2007a), are becoming more complex, and so their use as a storytelling medium is growing in importance and popularity. The unique interactive nature of games means that stories and characters can become more personal and involving.

Until now, stories in contemporary games are typically implemented using one or more standardised methods such as linear, branched or layered narrative (Paul et al., 2009). DIS is a relatively new field of interactive computer entertainment (Barros and Musse, 2005) that aims to create interactive applications capable to generate consistent narratives.

Traditionally, a story is considered to be a sequence of actions that leads to a sequence of events (Spierling, 2009). As defined in (Thue et al., 2007), DIS is “a story-based experience in which the sequence of events that unfolds is determined while the player plays”. A storytelling system can either actually create stories or enable the user just to tell different stories based on previously computed sequences of actions (Karlsson et al., 2007). As mentioned in (Thue et al., 2007), “deferring storytelling decisions to run-time can greatly improve the flexibility and replay value of a storytelling game”.

In fact, computer game stories can be implemented in different ways (Merabti et al., 2008): either linear, branching, parallel, or threaded. Most games typically follow a linear storyline, where the events of the story are presented in a predefined sequence. It can be argued that making a player follow a defined story can diminish the interactivity level of a game; the player is, after all, following a pre-set path already laid out for him/her by the author. In order to still convey a story and allow the player to feel a high degree of interactivity, the concept of interactive or non-linear storytelling has to be introduced. Simply put, interactive storytelling presents the opportunity for players to have an input on what is happening in the game world in which they are placed, to be the ones who dictate how certain events may come to pass within the constraints set by the story author.

Similar to other entertainment media, stories in games play a big role in increasing immersion, building tension and adding interest to the player. However, one main difference from the games to those other media is that games are interactive; they expect participation from the player and in turn, players expect to participate and get involved in the events the game is presenting and the outcomes of those events.

As thoroughly described in (Karlsson et al., 2007), a story model can be focused either on characters or on plots:

- In a character-based model, the storyline results from the real-time interaction among virtual autonomous agents. The main advantage of this model is the ability of anytime user intervention, meaning that the user may alter the plot as it unfolds by interfering with any character in the story. On the other hand, such an extreme interference level may lead the plot to unexpected situations or even to miss essential predefined events. Also, there is no guarantee that the narratives that emerge from the interaction of the above mentioned autonomous agents will be complex enough to create an interesting drama.
- In a plot-based model, characters should follow more rigid rules, specifying the intended plot structures. In a pure plot-based model, user intervention is more limited than in a character-based model but it is usually easier to guarantee coherence and a measure of dramatic power.

Another consideration is whether stories should be told using a first-person or a third-person perspective. As discussed in (Karlsson et al., 2007), a first person perspective tends to be particularly suitable for applications closer to digital games, whereas a third-person perspective is more appropriate for those involving film making.

Apart from its application in computer gaming, DIS has applications in several other areas like military training and interactive drama (Paul et al., 2009).

As discussed in (Charles et al., 2003), with the exception of emergent storytelling, DIS systems rely on various Artificial Intelligence (AI) techniques to support their behaviour including Assumption-based Truth Maintenance Systems (ATMS), Reasoning Maintenance Systems (RMS), logic programming and planning systems.

2.2. PLANNING

Planning is a combination of search and logic, two major areas of AI (Russell and Norvig, 2010). Planning involves knowing the state that you're in, the state you want to be in and then finding the sequence of operators that will get you from the current state to the final state. According to (Russell and Norvig, 2010), a planner can be considered as either a program that performs a search for a solution or as one that proves the existence of a solution.

To generate a storyline in DIS, planning systems are the most widely used techniques. They are considered extremely appropriate for DIS applications since plans are composed of discrete operations and stories can be easily converted to computer graphics-based output (Barros and Musse, 2007a). Apart from DIS systems, even AAA game titles such as the 2005 first-person shooter F.E.A.R. (Orkin, 2006) have employed successfully planning methods.

As stated in (Barros and Musse, 2005), the use of planning algorithms in DIS has two advantages:

1. Plans are a sequence of actions that can be used to achieve a given goal. They have an inherent notion of cause and effect that maps naturally to the concept of story.
2. Plans consist of discrete actions that can be individually assigned to and executed by characters.

However, there are fundamental differences between the goals of AI and DIS that should not be ignored (Barros and Musse, 2007a, Barros and Musse, 2005). In one hand AI algorithms are typically concerned with “hard” and precise goals such as optimality (e.g. finding the shortest path to a given place) whilst, on the other hand, the narrative goals in DIS are more subtle and not easily defined formally. That can be improved by using languages that use predicate logic, such as PDDL (Planning Domain Definition Language; discussed later in this section). Therefore, when applying AI algorithms in DIS problems these differences must be taken into account so narrative consistency of the generated stories will not be compromised.

Some of the problems with the current research in DIS, as discussed in (Spierling, 2009), are:

- AI engines appear obscure for authors from non-computer-science areas, and approaches in automatic planning are hard to grasp.
- Due to a lack of available playable prototypes, practical experience is missing.
- Naïve authoring approaches are generally too linear to suffice for highly interactive storytelling, which means granting end-users participation in the story.

There are many different description languages for representing planning problems. The most widely used is called PDDL (Planning Domain Definition Language) (Fox and Long, 2003). PDDL was derived from the original Stanford Research Institute Problem Solver (STRIPS) planning language which is slightly more restricted than PDDL since for example, STRIPS preconditions and goals cannot contain negative literals. STRIPS uses first-order predicate logic, and a world state is represented as a conjunction of predicates. There have been several versions of PDDL, consecutively extending the language expressiveness and features. Its first version was released on 1998 and the last version (i.e. 3.1) in 2008. Another planning language is ADL (Action Description Language) which is included as a PDDL extension (Fox and Long, 2003). ADL relaxed some of the STRIPS restrictions and made it possible to encode more realistic problems. Another major difference between these planning languages is that, in contrast to STRIPS, which use a closed-world model, the open world assumption applies to ADL. (Russell and Norvig, 2010)

Although planning systems have been used intensively in DIS systems, there have not been much novel solutions for DIS research with respect to planning algorithms. No DIS dedicated planning algorithm has been proposed as yet, and the justification of the choice of a planning algorithm for a DIS prototype is usually inadequate. In particular, a discussion of the specific requirements necessary for planning is often missing, and authors just propose comparisons of alternative existing planning algorithms in order to find the most appropriate one for a specific ad-hoc DIS problem domain.

2.3. Planning Using Constraints

Interest in using constraint techniques in planning problems has grown in recent years and has proven successful for many domains (Nareyek et al., 2005). As described in (Nareyek et al., 2005), the basic units of constraint-based problems are the constraints and the variables, where the constraints are entities that restrict the values that can be assigned to the variables. As further explained in (Barták et al., 2010), constraints are just relations while a Constraint Satisfaction Problem (CSP) indicates which relations (constrains) should hold among the given decision variables.

An interesting application of CSPs is in scheduling which shares some similarity with planning, but focuses essentially on actions, resources and time optimisation techniques. As explained in (Barták et al., 2010), scheduling concerns with the allocation of resources (such as time, machines etc.) to activities (actions) with the objective of optimising some performance measures. For example in time scheduling, the duration of a number of actions can be modelled as a CSP so they will not overlap while selected, or the final plan will not exceed the available time.

2.4. Multi-agent Systems

A definition of an agent in our context is that an agent is an entity which is part of an environment, perceives it with the help of sensors, and is able to act intelligently on it via a set of action mechanisms available to it (Vlachavas et al., 2005). Extending the above definition, we can add that an agent should be able to operate autonomously, persist over a prolonged time period, adapt to change, and create and pursue goals (Russell and Norvig, 2010).

A multi-agent system (MAS) is a system designed and implemented as a group of agents interacting with each other (i.e. communicating, competing, cooperating, coordinating, negotiating, and so forth). In such systems, the agents either work individually exchanging information and/or services with other agents trying to succeed to their individual goals or work together solving sub-problems so the combination of their solutions become the final solution. (Vlachavas et al., 2005)

According to (Vlachavas et al., 2005), there are two basic categories of interconnection models, i.e. ways for the agents to communicate with each other or with other systems; the blackboard systems, and the message passing systems.

In blackboard systems, there is a common working space (i.e. the blackboard) to be used by all of the system's agents, whereby they either share results, or they share tasks. When something is shared in this common area is accessible by all of the agents participating in the system. A blackboard system architecture is illustrated in Figure 1, adapted from (Vlachavas et al., 2005).

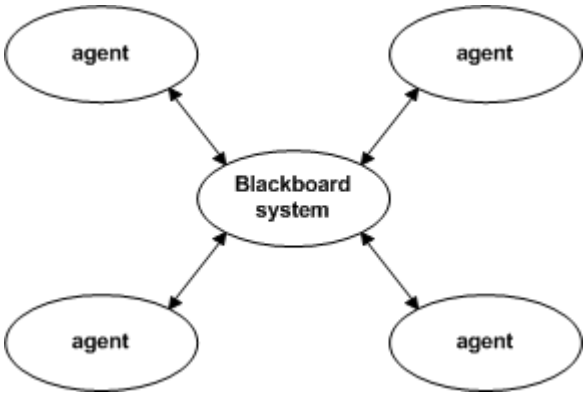

Figure 1: Blackboard architecture

On the other hand, in message passing systems the agents communicate directly with each other, sharing information via messages written in a communication language commonly accepted by all the system's agents. A message passing system architecture is illustrated in Figure 2, adapted from (Vlachavas et al., 2005).

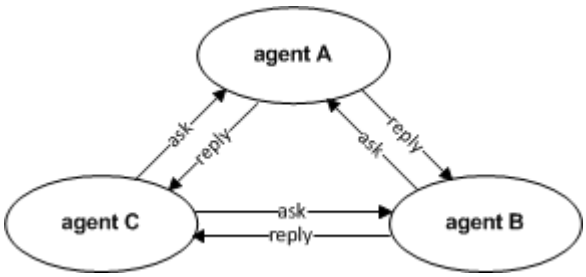

Figure 2: Message passing architecture

In any of the aforementioned interconnection models, there can be two types of communication: Either synchronous, meaning that an agent which asks a question to the system or to another agent inhibits its operation until an answer has been received, or asynchronous, meaning that the answer can be received at any point without a disruption in the agent's operation (Vlachavas et al., 2005).

An intelligent agent may implement some the following abilities (Vlachavas et al., 2005):

- *Autonomy*: The agents can operate without a direct intervention by a user or other agents, and they have (total or partial) control over their internal state, meaning that the agents are able to pursue their goals without constantly receiving user input.
- *Social ability*: The agents can communicate with other agents (or the user) using any kind of language that all of them can understand and "agreed" to use for the purpose of communication. Therefore, they are able for cooperation, coordination, and negotiation between them.
- *Reactiveness*: The agents are able to perceive the environment they exist in, and react to it based on the changes that are happening in it.
- *Pro-activeness*: The agents are not only able to react to the environment changes, but to act pro-actively as well, meaning that they can have goals and create plans to be able to achieve them.
- *Mobility*: The agents are not only static, but are also able to move in the environment they exist in.
- *Adaptivity*: The agents can constantly adjust to the environment or the choices of a user, meaning that they have an ability to learn.
- *Veracity*: The agents do not send wrong information on purpose.

- *Benevolence*: The agents are always trying to achieve their given goals.
- *Rationality*: The agents always act to achieve their goals, meaning that they don't do futile actions without being told to, and they don't act against achieving their goals.

In any definition of an agent, it is part of an environment, which can be categorised based on the characteristics they possess, as follows (Russell and Norvig, 2010, Vlachavas et al., 2005):

- *Fully observable* vs. *partially observable*: Whether an agent's sensors give it access to the complete state of the environment at each point in time or not.
- *Deterministic* vs. *stochastic*: If the next state of the environment is completely determined by the current state and the effects of the action executed by the agent, then the environment is deterministic; otherwise, it is stochastic.
- *Episodic* vs. *sequential*: In an episodic task environment, the agent's experience is divided into atomic episodes. In each of these episodes, the agent receives a percept and then performs a single action. Crucially, the effects of the actions taken in previous episodes do not affect at all the next episode.
- *Static* vs. *dynamic*: If the environment can change while an agent is deliberating, then the environment is considered dynamic for that agent, otherwise is considered static.
- *Discrete* vs. *continuous*: The discrete/continuous distinction applies to the state of the environment, to the way time is handled, and to the percepts and actions of the agent.

2.5. Multi-agent Planning

A common characteristic of the agents that work together in a multi-agent system is the capability of coordination via a communication language so they can communicate agreements and solve possible conflicts. A definition of coordination is that it is the attribute of a multi-agent system to solve problems in a common environment. Agents may coordinate their actions either to succeed a common goal (cooperation) or to succeed their individual goals (negotiation) (Vlachavas et al., 2005).

As explained in (Russell and Norvig, 2010), when there are multiple agents in the environment, each agent faces a multi-agent planning problem in which it tries to achieve its own goals with the help (or not) of the others. As discussed in (Vlachavas et al., 2005), in multi-agent planning, agents are generating a plan of actions and they will solve the problem based on that plan. During the execution, the plan is revised based on the new details and results.

Based on (Vlachavas et al., 2005), there are two types of multi-agent planning:

- Centralised multi-agent planning, in which a central agent is responsible to collect the partial or local plans of the other agents, to combine them in one plan and solve any conflicts that may occur.
- Distributed (a.k.a. decentralised) multi-agent planning, in which all the agents communicate with each other to generate their plans and to negotiate any possible conflicts.

2.6. Agent Architectures

There are several intelligent agents' architectures that are used in multi-agent systems, such as the reactive agents, the deliberative agents, and the belief-desire-intention (BDI) agents (Vlachavas et al., 2005). But, depending on the needs of the system, it is very common to see hybrid agent architectures, which combine elements from several architectures (Russell and Norvig, 2010).

The *deliberative agents* (Figure 3, adapted from (Vlachavas et al., 2005)) include an internal representation of the environment they exist in, and have knowledge of the set of rules that they must obey to, as well as of the set of actions they are able to execute. Therefore, they store a state which represents the evolution of their environment, as well as the current action they are executing, so they can decide for their next action (Vlachavas et al., 2005).

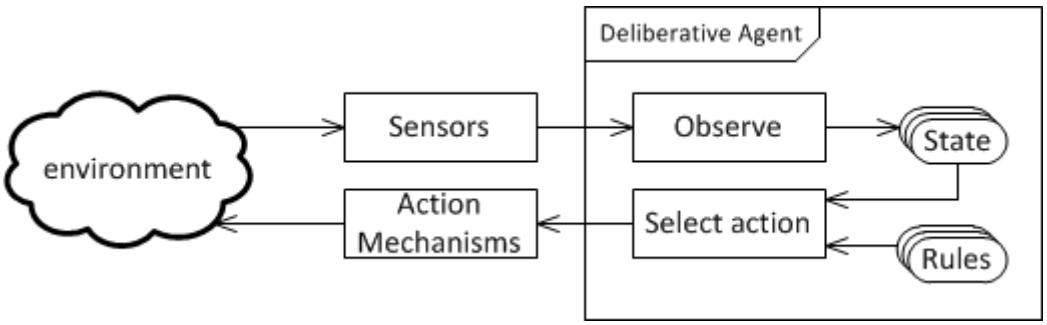

Figure 3: Deliberative agent architecture

The *reactive agents* (Figure 4, adapted from (Vlachavas et al., 2005)) on the other hand, do not store a representation of the environment that they base their reasoning on, and they implement a stimulus/response type behaviour based on the current state of the environment they exist in. These agents are receiving data information from their environment (perception) and, based on the rules they operate by, they decide on the action they will choose as a reaction to their perception. Finally, these agents do not have an internal memory, meaning that they do not calculate their next actions based on previous states of the world (Vlachavas et al., 2005).

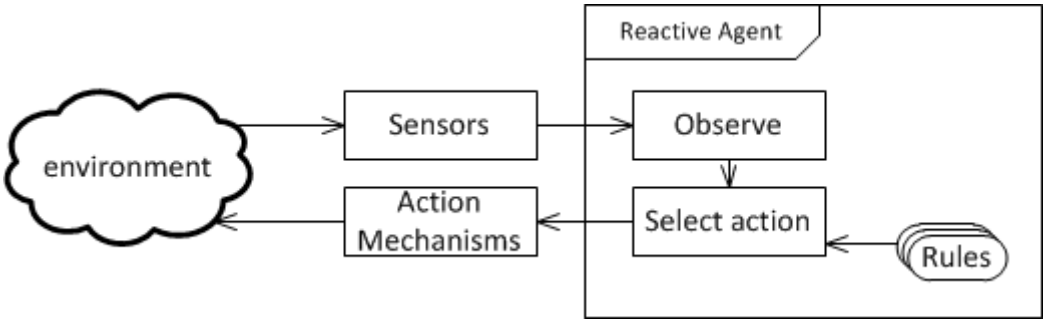

Figure 4: Reactive agent architecture

The *belief-desire-intention (BDI)* (Figure 5, adapted from (Vlachavas et al., 2005)) *agents* have a more complicated representation of their environment and they plan to achieve their goals. Their internal state consists of beliefs, desires, and intentions (i.e. the agent's plans) (Vlachavas et al., 2005).

- *Beliefs* are the agent's view and knowledge of the environment that exists in.
- *Desires* are related with the judgment that an agent will make for the future states of its environment, for example if a future state is desirable or not. In the desire level, the agent doesn't examine if a desired state is possible, and there

is also the possibility that some of the desired states are in conflict with each other.

- *Goals* are a subset of desires, and this is what the agent acts for. They should be achievable, and not in conflict with each other.
- *Intentions* are a subset of goals, which an agent tries to achieve at a given moment in time. In most cases, it is not possible to achieve all goals at once, therefore the agent selects a subset of them, which forms the intentions set, based on some hierarchy criteria.
- *Plans* are the set of actions that the agent can execute to achieve its intentions.

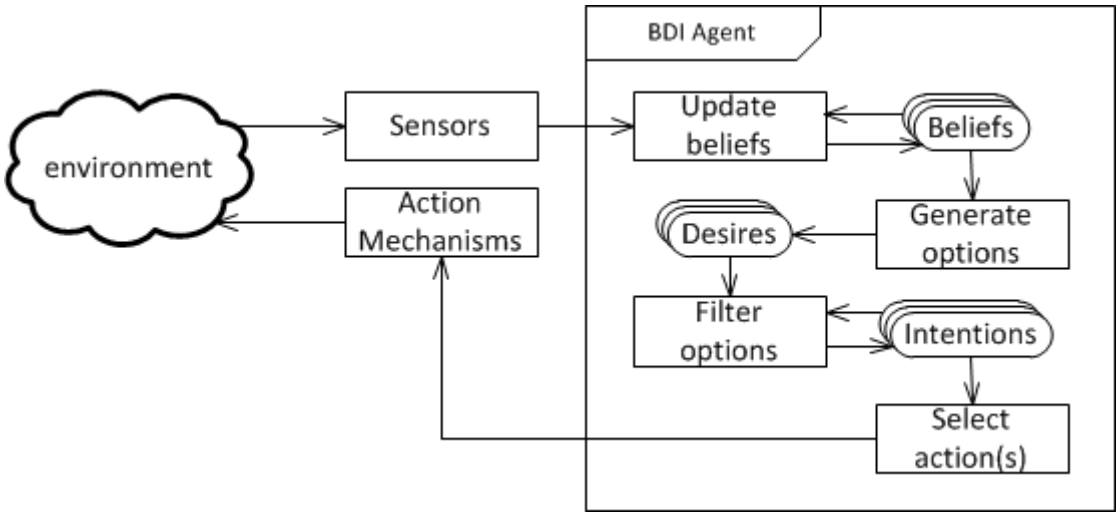

Figure 5: BDI agent architecture

2.7. RE-PLANNING

As discussed in (Doyle, 1996), planning is necessary for the organisation of large-scale activities since decisions about actions to be taken in the future have direct impact on what should be done in the shorter term. But even if a plan is thoroughly tested and well-constructed, its value decays as changing circumstances, resources, information, or objectives render the original course of action inappropriate. When changes occur before or during the execution of a plan, it may be necessary for a new plan to be constructed by either starting from scratch or by revising a previously generated plan.

Agents acting in complex and dynamic environments must often adjust their plans at runtime to avoid potential conflicts with other agents or using resources that are not available anymore. According to (Bartold and Durfee, 2003), such conflicts can be detected by selectively exchanging and comparing portions of agents' individual plans,

identifying inconsistent expectations, and adding synchronisation actions and/or blocking some action choices to ensure conflicts cannot arise.

Most planners solve each planning task from scratch by solving a series of similar planning tasks. Planning is time-consuming and severely limits the responsiveness and/or the number of what-if analyses that the planners can perform. To enhance their performance, (Koenig et al, 2002) states that re-planning methods that reuse information from previous planning episodes to solve a series of similar planning tasks are much faster than the approach of solving each planning task from scratch. (Doyle, 1996) states that "to re-plan effectively in demanding situations, re-planning must be incremental, so that it modifies only the portions of the plan actually affected by the changes".

As discussed in (Charles et al., 2003), a key aspect in the use of planning formalisms in storytelling consists in their ability to support re-planning and to offer representations embedding the potential for failure, however no solution have been proposed since for re-planning in DIS. There is an important gap in the use of re-planning methods in existing DIS systems and the proposed research will attempt to fill it.

2.8. Planning Algorithms Used in DIS

The following are some of the planning algorithms that have been used in DIS systems, listed alphabetically.

2.8.1. FF (Fast-Forward)

FF (Hoffmann, 2001) is a forward state-space searcher that uses the ignore-delete-lists heuristic, estimating the heuristic with the help of a planning graph. It then uses enforced hill-climbing search (modified to keep track of the plan) with the heuristic to find a solution. When it hits a plateau or local maximum – i.e. when no action leads to a state with better heuristic score- then FF uses iterative deepening search until it finds a state which is better, or it gives up and restarts hill-climbing (Russell and Norvig, 2010). FF was created by mixing some novel ideas with features of Graphplan and HSP (discussed in sections 2.8.2 and 2.8.3 accordingly) among others (Barros and Musse, 2007a).

Facing a search state S, a relaxed (ignoring delete lists) version of Graphplan is used to generate output for heuristic evaluation (the length of the solution plan) and the generation of helpful actions. Then, an enforced version of hill-climbing method considering only the helpful actions are used to find a solution plan. That is, all the direct successors of a state S are evaluated. If none of them has a better heuristic value than S, the successors' successors are evaluated, and so on, until a state S' with better heuristic value than S is found. When such a state is found, the path to it is added to the current plan and the search continues with S' as a starting point.

In summary, each iteration performs a complete breadth-first search for a state with strictly better evaluation. If enforced hill-climbing with helpful actions fails, then a best-first search considering all the applicable actions is performed to find a solution.

2.8.2. GRAPHPLAN

Graphplan (Blum and Furst, 1997) was the first planning algorithm that converted the planning problem into an intermediary data structure called a planning graph. Graphplan have moved the field of planning forward by obtaining impressive gains in performance compared to previous planning approaches, based on the experimental results documented in (Blum and Furst, 1997). Graphplan's main drawback is that although it is an optimal partial-order planner, its input language is quite limited (Russell and Norvig, 2010).

In Graphplan a plan is extracted from a graph. The graph consists of levels of literals which could be either true or false, and levels of actions of which the preconditions could be also either true or false. The graph is constructed starting at level zero (0) where all literals that are currently true are represented; these are true or false depending on the initial state and there are no other possibilities. Then, a level of actions for which the preconditions hold in the first level is added. This is followed by another level of literals that could hold if an action makes it true. Each level of literals gives the literals that could possibly be made true at that level depending on choices made earlier. Each level of actions gives all actions that could be used at that level depending on earlier choices.

The Graphplan algorithm creates the graph in steps; if at the current level of literals all literals from the goal are present without mutex relations between them, a solution plan may exist in the current graph. Otherwise, the graph is expanded by adding a new level of actions and a resulting literals level. If the graph possibly contains a solution, the algorithm tries to find it.

2.8.3. Heuristic Search Planner (HSP)

Heuristic Search Planner (HSP) (Bonet and Geffner, 2001) uses a STRIPS-based representation for problem description and searches the space of states from the initial state, using a traditional heuristic search algorithm and a heuristic automatically extracted from the STRIPS formulation (Charles et al., 2003).

HSP is a state space planning approach that can run either forward or backward and is much like path-finding. A state space search planner searches for a path along world states to the goals state. A world state can be reached by using an action. A forward searching planner starts with the initial state of the world and constructs a list of all reachable world states. These possible world states are nodes in the search tree. It will then choose one and repeat the process until it reaches a goal state. It will usually have a heuristic that gives rules for which node to expand, which world state to try first. A good heuristic function is important to make the planning fast.

The search can also start at the goal state. This is backward or regression planning. Regression planning may have a smaller space to search through. A state space planner will return a single plan. Actions in the plan are sometimes motivated by the next action in the plan but we cannot be sure of this. And sometimes actions are motivated by actions that are further along the plan. This is because actions that are in the plan are placed in a sequence that will make the preconditions of the actions be satisfied at the time they are executed.

According to (Russell and Norvig, 2010), HSP was the first state-space searcher that made state-space search practical for large planning problems.

2.8.4. Hierarchical Task Network (HTN) Planning

Hierarchical Task Network (HTN) based planning (Cavazza et al., 2002), which is also known as task-decomposition planning, is among the oldest approaches for providing domain-specific knowledge to a planning system.

An HTN planner solves problems by decomposition. The initial problem statement, the initial state and goal are viewed as a single action that must be decomposed into lower level actions. On the lower levels, actions are decomposed further until only primitive actions remain. There will often be choices available to the planner when choosing decomposition for an action. Action decomposition specifies a way to turn an action into a plan.

HTN is based on forward search, and thus can be searched to extract a task decomposition corresponding to a solution plan. It is also goal-directed at the same time, since the top-level task is the main goal. This brings the unique property that during planning itself the state of the world is known at all times (Charles et al., 2003).

2.9. Representation Languages

There are different description languages for representing planning problems. The following sections contain overviews of some of the representation languages which are typically used in planning algorithms.

2.9.1. Stanford Research Institute Problem Solver (STRIPS)

According to (Russell and Norvig, 2010), the Stanford Research Institute Problem Solver (STRIPS) (Nilsson and Files, 1971) was the first major planning system. The representation language used by STRIPS was way more influential than its algorithmic approach. What we today call the “classical” representation language, is close to what STRIPS used. STRIPS use first-order predicate logic, and a world state is represented as a conjunction of predicates.

To describe a planning problem in STRIPS we need an initial state of the world, a set of goals that should be achieved, and a set of actions that can be executed to achieve any

goals. According to (Vlachavas et al., 2005), the STRIPS model makes the following admissions:

- *Indivisible actions*: The actions of the planning problems are indivisible, meaning that the state of the world during the execution of an action is irrelevant; it is relevant only at the beginning and at the end of the action. Also, the execution of an action cannot be interrupted.
- *Deterministic effects*: There is no uncertainty for the effects of an action, since they are known beforehand.
- *Omniscience*: The planning system has complete knowledge of the current state of the world, as well as its options (based on the available actions).
- *Closed world assumption*: There is no possibility to include new or remove existing objects from the system's world.
- *Static world*: The world is modified only as a result of the actions executed by the planning system, and not by itself or by the actions of another entity.

2.9.2. Action Description Language (ADL)

The Action Description Language or ADL (Pednault, 1989) relaxed some of the STRIPS restrictions and made it possible to represent more realistic problems. Another major difference between these planning languages is that, in contrast to STRIPS which uses a closed-world model, the open world assumption applies to ADL. ADL also allows negative literals, as well as disjunctions. (Russell and Norvig, 2010)

2.9.3. Planning Domain Definition Language (PDDL)

The Planning Domain Definition Language (PDDL) (Ghallab et al., 1998) is the most widely used among planning algorithms. PDDL is an action-centred modular language and was derived from the original STRIPS planning language which is more restrictive than PDDL since, for example, STRIPS preconditions and goals cannot contain negative literals (Russell and Norvig, 2010).

Apart from its relation to STRIPS, PDDL is descended from several forebears (Ghallab et al., 1998): ADL (which is included as a PDDL extension), the SIPE-2 formalism, the

Prodigy-4.0 formalism, the UMCP formalism, the Unpop formalism, and, most directly, the UCPOP formalism.

As discussed in (Ghallab et al., 1998), PDDL is intended to express the "physics" of the domain, i.e. which predicates exist and which actions are possible along with the action's structure and effects. PDDL is neutral in a way that it doesn't provide any kind of "advice" (e.g. which actions to choose to achieve a goal) to the planners using it, and as a result of this neutrality, almost all planners will require extending the notation in different way. To this end, the language is factored into subsets of features (modules) called requirements, so each planner can choose to implement a subset of them.

There have been several versions of PDDL, consecutively extending the language expressiveness and features, for example expressing temporal planning domains in PDDL 2.1 (Fox and Long, 2003). Its first version was released in 1998 and the latest version (i.e. 3.1) in 2008.

2.9.4. Hierarchical Task Network (HTN)

Apart from the planners which are using STRIPS-like languages, there are also Hierarchical Task Network (HTN) based planners, which –according to (Lekavy and Navrat, 2007)– are based on hand-made hierarchical decomposition of the problem domain. The planner is provided with domain knowledge, expressed as the possible decompositions of tasks into subtasks. Tasks are categorised to primitive (i.e. directly executable) and non-primitive, which have to be decomposed into other tasks.

Each non-primitive task includes one or more lists of tasks it can be decomposed into, and these lists of tasks along with any other restrictions (e.g. precedence of tasks, variable binding, mutual exclusions, etc.) comprise a task network. (Lekavy and Navrat, 2007)

According to (Lekavy and Navrat, 2007), although the theoretical model of HTN is strictly more expressive than STRIPS, both approaches are –in practice– identically expressive and can solve all domains solvable by a Turing machine with finite tape.

2.10. Review and Critical Analysis of Existing DIS Systems

The following are most of the existing DIS systems that we researched, listed alphabetically.

2.10.1. Fabulator

In Fabulator (Barros and Musse, 2005), a planning algorithm is used to generate a sequence of actions (an actual story) performed by characters, that is capable to transform the system's world.

The player controls one character (the protagonist) and every other character is a Non-Player Character (NPC). All NPC's actions are determined by the system.

In DIEGESIS, there is not a main character that the player controls/observes; instead, the player can make choices (defined by the storyteller) for actions that can affect every character in the active story. Also, the player is allowed to select and view the story from the perspective of any of the characters (in the default view mode, the story is presented as a whole), and to be able to switch between them without any limitations, during the generation of the narrative.

Fabulator uses a "re-planning from scratch" approach; if an action of the player renders the current plan invalid, the system uses the planning algorithm to create a new plan. This way, the story is adapted to the player's actions.

In DIEGESIS, we have designed and implemented a new re-planning approach, aiming to make a minimal disruption to the original plan. We evaluated this approach against the approach of planning from scratch, concluding that the new approach has a better performance and has no difference in the outcome of the generated story compared to the other approach.

Fabulator's implementation "treats the planning problem as a state space search problem and uses the A* algorithm to solve it". The creators of Fabulator state that there are several planning algorithms specific for STRIPS-like domains that can achieve better performance than A*, but for small storyworlds (the result of authoring process in DIS) like the one the current implementation uses, performance is not an issue. They

also state that the most important shortcoming of their work was its reliance on predicate logic to represent the world state.

In a latter implementation of the system, the Metric-FF planning algorithm is being used (Barros and Musse, 2007a). In this implementation, the notion of tension arc is being introduced ("the level of tension in a story in function of time results in a curve called tension arc") along with a mechanism that makes the generated stories follow an author-defined tension arc.

2.10.2. FAÇADE

Façade (Mateas and Stern, 2003) is a 20 minute interactive drama which can be played multiple times, where the player has to interact with a couple of NPCs that are experiencing marriage issues. According to (Karlsson et al., 2007), it integrates characteristics of both plot-based and character-based approaches.

Façade consists of a 3D world, believable agents, a broad and shallow natural language processing system, and a drama manager. Façade implements a reactive behaviour planner that selects, orders, and executes fine-grain plot elements called beats that describe action/reaction behaviours that story world characters will perform. The drama manager uses this planner to manage the story resulting from the simulation.

A beat is the smallest unit of dramatic action that moves a story forward. Beats are authored by a human author and are given preconditions and effects. The preconditions specify when the beat can be applied and the effects specify what the result will be in the story state. The set of beats together implicitly defines a narrative graph.

According to (Arinbjarnar et al., 2009), the beats are "explicitly pre-authored, with all actions within the beat being fully defined, and the actions of all roles being assigned to allow for multi-agent coordination".

By traversing the beats in some sequence, which depends on the interaction of the human player, the story is moved forward by the drama manager. Because the number of different ways in which beats can be sequenced is large the player can experience a lot of freedom in what story is experienced. The way the drama manager changes the

simulation is by modifying the behaviour of the characters; it adds and removes behaviours while the simulation runs. In Façade the number of beats is approximately 200 and they are used in the rate of once every minute.

As discussed in (Roberts and Isbell, 2008), due to the level of granularity required to author beats and their interactions, a beat-based drama manager seems ideally suited to small-world variety dramas (like the one used in Façade). “However, the freedom of replayability and authorial control may come at the price of ease of authoring, at least for large systems.”

The characters are programmed using A Behaviour Language (ABL). “ABL is a reactive planning language, based on the Oz Project language Hap, designed specifically for authoring believable agents - characters which express rich personality, and which, in [this] case, play roles in an interactive, dramatic story world”. (Cooper, 2011)

Façade is the first complete DIS system published. According to (TeessideUniversity, 2010), its global agency is limited and user's actions (which mainly consist of typed text) have little explicit consequence on future developments of the story.

2.10.3. GADIN

The Generator of Adaptive Dilemma-based Interactive Narratives (GADIN) (Barber and Kudenko, 2009) dynamically generates interactive narratives which are focused on dilemmas to create dramatic tension. Its authors claim that the system addresses two open challenges: maintaining the dramatic interest of the narrative over a longer period and (story) domain independence.

As described in (Roberts and Isbell, 2008), to construct the narrative, GADIN selects among the set of available dilemmas based on an appropriateness estimate, as well as based on the frequency with which each particular type of dilemma has been employed already.

Its planner (which is based on the Graphplan algorithm) creates sequences of actions that all lead to a dilemma for a character (who can be the user). “The user interacts with the storyworld by making decisions on relevant dilemmas and by freely choosing

their own actions. Using this input, the system chooses and adapts future storylines according to the user's past behaviour."

DIEGESIS includes a concept similar to dilemmas, in the form of "choices". The storyteller (i.e. the person who models the story which is generated and executed by DIEGESIS) can mark any kind of action as a choice. When such an action is about to occur, DIEGESIS either makes a choice itself, or asks the player to make a choice whether the action will happen or not. The idea behind choices in DIEGESIS is that important decisions throughout the story should be marked as choices so they can potentially alter the outcome of the generated narrative.

According to (TeessideUniversity, 2010), GADIN "continuously presents the user with dilemmas to keep the narrative going". While in GADIN the generation of dilemmas is necessary to keep the narrative interesting and going, DIEGESIS' stories can be generated even without any choices, although –as we already mentioned- the storyteller is encouraged to use them since they can potentially have a significant impact on the outcome of the generated narrative.

Although the authors consider other application domains –in (Barber and Kudenko, 2009) they implemented a finite children's short story–, as discussed in (Barber and Kudenko, 2007), GADIN is best suited for genres which places a particular emphasis on stereotypes and clichés, such as soap operas which is the domain the system was evaluated on. Comparing the above to our framework, we believe that DIEGESIS is both suited for movie-like experiences including relatively long-length finite stories, as well as shorter stories, since it provides to the storyteller the flexibility required to experiment with multiple genres and lengths of stories.

(Roberts and Isbell, 2008) argues that authoring a story in GADIN is not easy, since it requires STRIPS-like specification of the domain and character specific information, which necessitates AI competence. DIEGESIS uses a combination of modelling approaches: The storyteller needs to model the game world both in PDDL and in XML. To make the authoring process easier, we are using a PDDL editor created by (Cooper, 2011), and although the authoring process in XML is quite easier compared to PDDL, we have designed an XML editor as an extension to the PDDL one.

GADIN's authors have identified that the main problem of GADIN's planner is that as more characters and actions are included, the time spent planning becomes unreasonably long; the time increases exponentially with the number of characters and the number of actions. On the other hand, neither the number of locations nor the number of dilemmas has an impact on the speed. According to (Arinbjarnar et al., 2009), "with an increased number of actions, dilemmas and characters, the planning becomes too slow for a real-time experience of the narrative"; (Roberts and Isbell, 2008) agrees, stating that "online planning approach can be slow in any sizable domain" The authors claim that a potential solution would be the use of a form of hierarchical planning.

During the implementation phase of our research, we have identified that the bottleneck of DIEGESIS' planner is its pre-processing process, i.e. the time needed by the planner to pre-generate nodes that are later used in the actual planning and re-planning process. Our planner only needs to pre-process the information of a level once and then the pre-processed information can be reused in any planning and re-planning episode of that level. Therefore, any delay due to the pre-processing will only affect the loading time of the level, and not the experience of the player while executing the level.

The results of the evaluation we document in section 6.5.5 show that DIEGESIS is capable of generating and executing a large and complex story containing several characters in a very short amount of time, making the framework suitable to be used for the purpose of DIS.

As it is mentioned by its authors, GADIN is performing re-planning, but there are not a lot of details for its mechanics. According to (Paul et al., 2010, Paul et al., 2011), if the player's actions move too far away from the goal state or they make the story goal improbable, GADIN randomly selects a new story goal and reveals it to the player. This new story goal does not involve any further player actions; all further actions are carried out by NPCs.

The system performs continuous planning in a thread using a global planning graph, so the re-planning can be faster, and performs a centralised planning for all the agents operating in the system.

In contrast, DIEGESIS performs a decentralised planning; each agent is generating a plan based on its own goals, and then tries to execute it. The framework deals with the execution of the agents in a higher level, and when a part of the plan fails, instructs the agent to re-plan based on its current knowledge of the state of the world. We believe that this provides a more realistic approach to the generation of a story, since each agent acts as a real person, generating an autonomous plan considering its own needs.

2.10.4. I-STORYTELLING

The Interactive Storytelling approach described in (Cavazza et al., 2002) and (Charles et al., 2003) is character-based supporting user interventions at any time. The graphic environment of the system was based on the Unreal™ game engine and the scenario used is inspired from the popular sitcom Friends™.

The first prototype of the system (Cavazza et al., 2002) includes four autonomous agents/characters, and is able to generate short stories up to three minutes in duration, with approximately one "beat" (Mateas and Stern, 2003) per minute. According to (Barber and Kudenko, 2009), a longer narrative is not easy to accomplish due to the large amount of content and ordering predefinition required. As further discussed in (Paul et al., 2009), each character has a number of context-specific subtasks, therefore careful authoring of each character's task network is needed to ensure that an interesting narrative will occur. The character roles are designed a priori for the story, therefore the actions that a character can take are scripted for a particular role; character's roles are pre-selected at design time for a particular story.

As discussed in (Karlsson et al., 2007), "the main doubt about pure character-based approached is to what extent dramatic and engaging narratives may actually result. The task seems to be easier with genres like sitcoms, wherein the climax of a story is not so clearly distinguishable."

The user of the system can wander in the 3D world as an invisible avatar and interact with key objects, and can make suggestions (using a speech recognition interface) to NPCs, which may or may not be followed (Arinbjarnar et al., 2009). According to (Paul et al., 2010), "plot coherence is ensured by allowing only NPC actions related to the on-going story".

Hierarchical Task Networks (HTNs) are being used since the characters' roles can be represented in a consistent fashion as such. A single HTN corresponds to several possible decompositions for the main task therefore an HTN can be seen as an implicit representation for the set of possible solutions.

The search algorithm that produces a suitable plan from the HTN searches the HTN depth-first left-to-right and executes (or at least attempts to execute) any primitive action that is generated. Backtracking is allowed when these actions fail. In addition, heuristic values (which are used to represent narrative concepts as well) are attached to the various sub-tasks, so forward search can make use of these values for selecting a sub-task decomposition.

Any agent's action will in turn trigger re-planning. This is implemented using the search mechanism of the HTN planner by back-propagating the failure of the action to the corresponding sub-goal, so search will backtrack and produce an alternative solution.

Another planning formalism that was used in a second implementation (Charles et al., 2003) of the same scenario (for comparison reasons) was HSP. Compared to the HTN implementation, HSP offers greater flexibility in the definition of action and more variability in the stories generated while HTN offers clear authoring principles and a global vision of the baseline plot.

According to (TeessideUniversity, 2010), the system's strong points are the general nature of the HTN-based planning system, the dramatization of narrative situations and the user interactions' influences on the unfolding of the narrative in real-time, while its limitation is the lack of control over the quality of the narrative generated.

In (Charles et al., 2003), the authors mention that HTN is not a good solution when it comes to re-planning, and they switched to the HSP algorithm, claiming that it provides a better re-planning solution. The actual mechanics of re-planning are not described though. Based on the provided examples, the factors which can trigger re-planning are usually user interference or the availability of a resource.

In our framework, user interference is just one of the options which can trigger re-planning. Since each agent operates as an individual, generating its own plans as we explained before, it is very common for an agent's plan to interfere with a plan of

another agent, causing the latter to re-plan. The interference can be something simple such as the availability of a resource, or even something more complicated, like the death of a character (represented by an agent), which can potentially have a huge impact to the whole story, altering significantly the agent's goals, therefore their plans.

2.10.5. LOGTELL

LOGTELL (Karlsson et al., 2007) is a storytelling system based on modelling and simulation. Its model includes typical events and goal-inference rules and tries to conciliate both plot-based and character-based modelling. Successive cycles of goal-inference, planning, plan recognition and user intervention are used to generate plots.

Typical events are described by parameterised operations with pre-conditions and post-conditions so that planning algorithms can be used for plot generation while, on the other hand, the goal-inference rules model the behaviour of the various actors providing some character-based features. The rules specify how situations can bring about new goals for each character.

In LOGTELL, the stories are told with a third-person view-point, and user intervention is always indirect. That means that during the simulation the user can either let the partially-generated plots that seem interesting to be continued, or try to enforce the occurrence of situations and events. But, these interventions might be rejected by the system whenever it finds no valid way to change the story to accommodate the intervention. According to (Barber and Kudenko, 2007), the resulting story is graphically presented at a lower level, without any possible user interaction.

In DIEGESIS, as we already mentioned, the storyteller (i.e. the person who models the story) can mark any kind of action as a choice, which is the way the player can interfere with the story. User interferences are always accepted by DIEGESIS and they affect the generation and execution of the story in real time so the user will be able to form the story in the way she wants, no matter how much impact they have on the generated narrative.

The planning tool used is a non-linear planner implemented in Prolog, adapted from others' work with extensions. The use of a non-linear planner is justified as it seems more suitable because it uses a least-commitment strategy.

The generation of a plot starts by inferring goals of characters from their initial configuration. Then, the system uses the planner that inserts events in the plot in order to allow the characters to try and fulfil their goals. When the planner detects that all goals have been either achieved or abandoned, the partial plot is generated and presented to the user and can be optionally dramatized. If the user does not like the partial plot, an alternative can be generated. If the plot is accepted, the process continues by inferring new goals from the generated situations. If new goals are inferred, the planner is activated again to fulfil them. The process alternates goal-inference, plan generation/recognition and user interference until the moment the user decides to stop, or no new goal is inferred. In the goal-inference phase, forward reasoning is being used, where in the planning phase, an event inserted in the plot for the achievement of a goal might have unsatisfied preconditions, so they are checked via backward reasoning.

The authors argue that combining goal inference, plan generation/recognition and user participation constitutes a promising strategy towards the production of entertaining and coherent plots, but on the negative side, plan generation is limited by computational complexity considerations. They also mention that "modern and post-modern genres with their emphasis on a more radical transgression of any conventions should not be so easy to formalise in a systematic way".

2.10.6. MIMESIS

The Mimesis (Riedl et al., 2003, Young and Riedl, 2003) system defines an architecture for building and coordinating interactive adaptive narratives. According to (Arinbjarnar et al., 2009), it is designed as a general architecture, therefore it should work with any game engine.

Mimesis uses two planners; the narrative planner, which is responsible both for determining the actions that will occur within the virtual environment as the story unfolds and for modifying the plan during the story's execution when the player's

actions deviate substantially from the story's intended structure; and the discourse planner, which is responsible for selecting the communicative techniques that will be used to convey the unfolding action to the player. Both planners use the Longbow planning system, a hierarchical partial-order causal link planner that can produce plans both for physical actions as well as communicative ones.

The narrative planner takes as input a declarative representation of all the actions that are applicable in the virtual world as well as a specification of the goals for the end of the story. The narrative planner searches for a story plan, which is a sequence of actions which will be carried out by the characters in the story (including the character controlled by the player) and will both satisfy the goals of the story and provide an engaging narrative arc.

The discourse planner takes as input the story plan generated by the narrative planner and a library of communicative actions that can be used by the game engine to convey the unfolding action of the story. Then, the discourse planner creates an action sequence containing directives to be carried out not by characters in the story world but by the game engine's interface resources and intended to be executed concurrently with the story plan itself.

Mimesis deals with re-planning in the following way (Mateas and Stern, 2003): It monitors the story world for potential player actions that might threaten causal links in the current story plan. When threat is detected, the system either generates a new plan which accommodates the player's potential action while still accomplishing the story objectives, or intervenes by causing the player action to fail and thus protect the threatened causal link. According to (Paul et al., 2011), if Mimesis fails a player's action, she will be given a pre-authored reason for the failure (e.g. a gun jamming preventing the player from killing an important character).

(Roberts and Isbell, 2008) claim that re-planning in Mimesis is expensive in any sizable domain. Because of that, Mimesis builds re-planning policies in an opportunistic fashion; when processing demands are low, the system pro-actively computes policies for plans other than the one that is currently executing.

In contrast, DIEGESIS interleaves plan generation and plan execution, therefore re-planning is happening in real time during execution for each individual agent when is required. Furthermore, based on our evaluations, the re-planning solution we created does not suffer from performance issues.

2.10.7. MIST

Multiplayer Interactive StoryTelling (MIST) (Paul et al., 2009, Paul et al., 2010, Paul et al., 2011) is "a system for interactive storytelling in a dynamic virtual world where NPCs can perform tasks autonomously to satisfy their internal motivations, as well as interacting with each other in various ways". It uses AI planning methods for story creation and revision and character role selection. The system's proposed architecture has two main components: a game engine and a drama manager.

The game engine handles the display and update of game world objects and also interacts with characters and the drama manager. Each character in the game operates under a Belief-Desire-Intention (BDI) framework and has its own HTN planner in order to facilitate the creation of a dynamic game world where characters can interact with each other in a non-deterministic way.

Generally, characters use their planners to decide how to perform tasks or achieve goals assigned to them by either the drama manager or the game engine. Characters also convert their local knowledge (acquired by sensors) into a partial game state representation for use by its planner. The planner uses this information to guide the decomposition of an assigned task into primitive tasks whose preconditions are known to be satisfied. All possible plans generated by the planner are ranked in order of decreasing plan cost; in case there is more than one possible plan, the planner returns one that minimises the total cost of all primitive tasks in the plan.

The drama manager has a hierarchical network of story elements, which can be pieced together in different ways to form a story. The current state of the game world is passed to the drama manager periodically from the game engine. Then, the drama manager attempts to create a story that fits the current state of the world via its HTN planner, and the network of story elements. The authors argue that using an HTN planner in this way (i.e. as part of the drama management subsystem) could

potentially enable the creation of many story variants based on the state of the world at a particular time.

In MIST, NPCs that have been assigned roles in a story plan are prevented from disrupting the story by being prevented to attempt to achieve their internal desires while the story is in progress. When it detects an invalid plan step resulting from the actions of non-story characters in the story plan, the drama manager attempts to repair the on-going story. The repaired plan is required to be consistent with the steps that have already been completed in the original plan.

The authors have considered two different approaches to detecting invalid plan steps while a story is in progress: The first is to look one step ahead to check that the preconditions of the next plan step are satisfied. An important limitation of this approach is that because of commitments made by characters close to the point of (potential) failure, a consistent plan repair may not be possible.

The second approach is that the drama manager continuously checking the preconditions of all future plan steps. This kind of detection increases the chance of finding a consistent plan repair because it enables the drama manager to avoid commitments being made by story characters close to the point of failure; therefore it is more likely to find a consistent plan repair that bypasses the invalid step. This approach applies though only to situations where a plan step is made invalid by the deletion of a precondition that was true in the initial state from which the story was generated; it does not apply to situations where a plan step is made invalid by the deletion of a precondition achieved by an earlier plan step.

The approach that they ended up using removes the unsatisfied precondition from the initial state and uses the HTN planner to search for an alternative story plan that begins with the same steps as the original plan, up to (and including) the most recent step that has already been completed. The new story plan is both consistent with the original plan and generated from the same (correctly authored) HTN, thus ensuring that plot coherence is maintained.

In DIEGESIS, we have modelled our agents (representing characters in the story) to be autonomous and opportunistic, generating and trying to execute plans considering

only their own needs, as we believe that this provides a more realistic approach to the generation of a story, since each agent acts as a real person. Therefore, in our context it does not make sense to predict and prevent plan failures since a plan can fail either due to user intervention (which cannot be predicted), or intervention by other characters, or –in some cases– pure chance (discussed in section 4.10). In any case, failed plans due to "unpredicted" reasons are realistic and have the potential to enrich a generated narrative.

Another difference between MIST and DIEGESIS is in the way we deal with plan repair. In our re-planning solution, as we interleave plan generation and plan execution, when a plan fails, we discard the already completed actions and we only re-plan for the failed (and some of the pending – discussed in detail in section 4.14) actions of the plan, merging the new partial plan with the unexecuted portion of the original plan. Finally, situations like these described by MIST in which "a plan step is made invalid by the deletion of a precondition achieved by an earlier plan step" are not applicable in DIEGESIS, since it is not possible to generate a valid plan where the effects of a previous action renders a future action (in the same plan) invalid.

Although the system was designed to use a set of HTN planners, in the initial implementation of the system (Paul et al., 2009) the authors used JPlan, a Java implementation of Graphplan, as the planning component of the system. As they explain, they needed a Java implementation of an algorithm for their first prototype, and JSHOP2 which is the most popular Java-based HTN planner had limitations when it comes to real-time planning that was needed in the system.

Although that the algorithm is efficient and optimal, it has been identified by the creators of the system to have limited features for the purposes of their research. It was also stated that "the lack of expressivity in the operator input language restricted scalability". The creators decided that, given the limitations of the graph planning algorithm, the most flexible solution would be the creation of a HTN planner in a subsequent implementation of the system, which they did in (Paul et al., 2010), implementing an HTN planner written in Prolog.

2.10.8. OTHELLO

Othello (Chang and Soo, 2009, Chang and Soo, 2008) is a multi-agent simulation game environment where narratives arise on the fly from spontaneous interactions among characters during the game.

An agent-based and plan-based storytelling approach has been used and assumes that plans serve as a proper representation of narratives and that a narrative is the result of plan execution by individual AI characters. A simulation session is considered to contain multiple autonomous planning agents who are given mental states, personality traits and social relations. Narratives are expected to be the total sequence of actions in the plans that the agents make and execute.

Othello's character plans embody the sociality of narratives, and are called social plans. A social plan realises a common narrative idiom that a character works to bring change to another character. Although a persistent game universe can develop intertwining narrative units where multiple characters exist (with all of them having their own social plans), Othello limits the focus on generating separate narrative units, each of which have a main character who is the builder of the social plan. A narrative unit is considered to be the result of the execution of a social plan.

As an example of the size of a social plan, the authors mention that the simplified plot of Shakespeare's Othello (i.e. the manipulative scheme of Iago against Othello) that they used in their simulations can be viewed as one social plan.

To generate these social plans and allow NPC agents to engage in story-like activities by influencing others during a game session, Othello uses HSSP (the authors mention that in a previous version of the system they were using the Optop planner), a planning tool which interleaves social reasoning with state-space forward-search planning, guided by an adapted version of the HSP heuristic. As the authors explain, "apart from the heuristics part, the planning process itself is a normal forward search into the state space".

Finally, the authors discuss that although the scalability of the total narrative length is not within the scope of their research, their findings suggest a negative

correspondence between the number of actions and the social plan length using a classical planning approach like HSSP.

2.10.9. PaSSAGE

PaSSAGE (Player-Specific Stories via Automatically Generated Events) (Thue et al., 2007) is an interactive storytelling system that "uses player modelling to automatically learn a model of the player's preferred style of play and then uses that model to dynamically select the content of an interactive story". PaSSAGE uses a plot-based approach, including personalisation of the narrative experiences in the form of selection of events which matches the player preferences.

According to (Roberts and Isbell, 2008), the system uses a three level hierarchy for defining a narrative similar to the idea of Façade's narrative sequencing: "the event sequence level where the components of the story are selected; the structure level where the details concerning the time and place of story events are determined; and lastly, the behaviour level where the actions of individual characters are determined".

PaSSAGE uses some pre-defined player types (Fighter, Method Actor, Power Gamer, Storyteller, and Tactician) and during gameplay, it learns a player model expressed as weights for each of the above player types. PaSSAGE generates its stories using a library of possible events, called encounters, each of which has been pre-filled by an author with a number of possible events that would be suitable for each player type. Each encounter has one or more branches (i.e. potential courses of action for the player to take in that situation). The encounters follow a particular order depending on their type (Arinbjarnar et al., 2009). As it is mentioned in (Roberts and Isbell, 2008), this approach makes the stories hard to author, since it requires exhaustive and rich annotations of many sub-plots.

While searching for an encounter to run, the system examines each encounter's set of branches, and chooses the encounter whose branch fits the player model the best, via an inner-product calculation. Also, to help maintain a strong sense of story, encounters are grouped into sets.

The system is independent of time, place, and actor identity since the encounters are scripted generically and their details (e.g. when and where an encounter should occur) are determined at runtime.

As an experiment, the authors modelled a modified version of the fairy tale “little red riding hood” and visualised using the toolset provided by the role playing game Neverwinter Nights. Their finalised model consists of 20 possible lines of gameplay called paths, with five different endings.

According to (Barber and Kudenko, 2009), the player model used in PaSSAGE is less likely to be applicable in less computer game-oriented domains, since it is based specifically on computer game players.

2.11. DIS Systems Comparison

The following table (1) provides an overview of the features of the related DIS systems discussed in the previous section. The presented features are the following:

- *Story Model*: The story model of the system, which (as discussed in section 2.1) can be either character-based, or plot-based, or a combination of both.
- *Type of Planning*: Either centralised, meaning that the system’s planner generates a combined plan for all of the involved agents, either decentralised, meaning that each of the agents generate its own plan.
- *Re-planning*: Whether the system performs re-planning or not.
- *Planning Algorithm*: The planning algorithm used to generate plans of actions.
- *Representation Language*: The representation language used to model the story world.
- *Perspective*: Whether the stories are presented via a first-person perspective, i.e. the player experiences parts of the story which are related to one character, or via a third-person perspective, i.e. the player can experience the story irrelevant of a main character.
- *Interactivity*: How the end-user can interact with the system.
- *Extendibility*: If the system provides tools to connect it with other systems, or generate new stories.
- *Audience*: The types of audience the system is designed for.

DIS System	Story Model	Type of Planning	Planning Algorithm	Representation Language	Re-planning	Perspective	Interactivity	Extendibility	Audience
Fabulator	Plot-based	Centralised	Initially A*, then Metric-FF	Initially STRIPS & ADL, then PDDL	Yes, from scratch	First-person	Player controls only protagonist; the rest are NPCs	No authoring tools; the PDDL files can be modified	General
Façade	Plot-based with some character-based elements	No information	Reactive behaviour planner	ABL	No information	First-person	Player controls only protagonist; the rest are NPCs	No tools for extendibility	Adults
GADIN	Character-based	Continuous centralised	Adaptation of Graphplan	STRIPS-like	Yes, without information about its mechanics	Third-person	Player input to resolve dilemmas	No tools, new stories are possible but they need to be hard coded	Soap opera fans and children
I-Storytelling	Character-based	Initially decentralised, then no information	Initially HTN (depth-first left-to-right search with heuristics), then HSP	Initially HTN, then STRIPS-based	Yes, initially by backtracking in the HTN, then without information about its mechanics	Third-person	Player can wander in the 3D world as an invisible avatar and interact with objects; speech input to provide advice to NPCs	No information	Sitcom fans

LOGTELL	Plot-based with some character-based elements	Centralised	Non-linear planner in Prolog	Prolog	No information	Third-person	Indirect/passive interaction	No tools available	Anyone (depending on story)
Mimesis	Plot-based	Centralised	Longbow planning system	No information	Yes, monitors game world for threats and builds solutions pro-actively	First-person	Player controls only protagonist; the rest are NPCs	No information	No information
MIST	Character-based	Decentralised	Initially Graphplan, then HTN written in Prolog	Initially STRIPS, then Prolog	Yes, by trying to predict and repair in advance a potentially invalid plan	Third-person	No information	No information	Computer game players
Othello	Character-based	Decentralised	HSSP (state-space forward-search planning, guided by an adaptation of HSP)	PDDL	No	No information	No information	No information	No information

PaSSAGE	Plot-based	No information	No information	No information	No information	First-person	Player controls only protagonist; the rest are NPCs	No information	Computer game players
DIEGESIS	Hybrid	Hybrid	New planner based on Graphplan	PDDL and XML	Yes, interleaving plan generation and plan execution	Both first and third-person, including vantage points	Player can make choices for any character or event	PDDL and XML editors	Anyone (depending on story)

Table 1: Feature sets of Digital Interactive Storytelling systems

In the previous section we reviewed and critically analysed 9 state-of-the-art DIS systems and in Table 1 we combined and presented a set of their features (analysed at the beginning of this section) for comparison purposes.

Regarding *story models*, 4 of the systems (GADIN, I-Storytelling, MIST, and Othello) are using a pure character-based approach and 3 of them (Fabulator, Mimesis, and PaSSAGE) a pure plot-based approach. The remaining 2 systems (Façade and LOGTELL) claim that they are using a plot-based approach combined with some character-based elements.

DIEGESIS uses a hybrid approach, combining both plot-based and character-based elements. More specifically, as we will discuss in section 4.1, the game world (created by a storyteller) is organised in multiple levels which can represent possible parts of a story. Typically, a level represents a broad area where a number of events in a story may occur. The levels are organised in a hierarchical manner; each level may include potential successor levels which have a logical connection with it. As soon as a level is complete, the framework makes an informed decision and based on what happened previously during the generation and execution of the story, either loads a new level or ends the story (the detailed process is discussed in section 4.7). Using this plot-based approach, DIEGESIS always has a high-level control over the overall structure of the story, being able to transition the story between levels which make sense, producing a coherent narrative.

The authors of (Carmichael and Mould, 2014) designed a framework focusing on deciding which scenes (a concept similar to DIEGESIS' levels) to offer to players next. They use a similar plot-based approach and their framework "uses simple calculations to prioritise scene nodes". Their scenes (which are designed beforehand by a storyteller) are loosely connected to each other and they include values that can be modified during runtime to prioritise them over others, as well as preconditions that need to be met so the scene can be applicable.

The main difference between that framework and DIEGESIS' transitioning component is that in (Carmichael and Mould, 2014)'s framework, when a scene is complete the player is presented with the potential scenes and is asked to select which one she wants to execute next, knowing beforehand the content of each scene, a concept

similar to the quests included in role playing games. In contrast, DIEGESIS' transitioning component makes the choice of which level to execute next itself based only on what happened previously in the story and the preconditions set by the storyteller for each level, something that we believe it adds both to the generation of a coherent narrative and to the emotion of unexpected of the player since she does not know what will happen next based on choices she made during the execution of the story.

Continuing with the story model discussion, in DIEGESIS, when a level is loaded to be executed, we move closely to a character-based model; each agent may have some initial intentions, but is able to operate autonomously and opportunistically to achieve its goals. The framework won't interfere with the decisions of an agent (even if they are imposed by the player or the Oracle – discussed in section 4.10) even if they mean that the story cannot progress any further, although –in the bottom line– that is based on the story modelling performed by the storyteller. The authoring process in DIEGESIS provides enough freedom to the storyteller to operate whichever way she wants; either to create a relatively rigid storyline without much room for highly diverse narratives, or to model a story in a way that everything is fluid; a lot of player/oracle choices, several potential goal injections based on actions that may occur, and several uncertain actions; all of these features can contribute to unexpected situations and more emergent narratives.

Moving to *types of planning*, 4 systems (Fabulator, GADIN, LOGTELL, and Mimesis) are performing a centralised planning, 3 systems (I-Storytelling, MIST, and Othello) a decentralised planning, and the rest 2 systems (Façade and PaSSAGE) do not provide any information about it.

DIEGESIS follows a hybrid approach. On the plan generation level, it performs a decentralised planning; each agent (represents a character in the story) is modelled to be autonomous, opportunistically generating and executing plans based on its own goals. We believe that this provides a more realistic approach to the generation of a story, since each agent acts as a real person, generating an autonomous plan considering its own needs.

In the case of plans execution though, our approach borrows the control and coordination concepts from the centralised planning approach. Although the plans are

individual, we want DIEGESIS to dictate the execution phase of the agent's plans (therefore the generation of the story) so the system can have a better control and understanding of what happens during the generation/execution of the story, and to be able to interfere if needed.

The systems use a variety of *planning algorithms*. Fabulator's initial implementation was using A*, but then moved to Metric-FF. GADIN uses an adaptation of Graphplan. MIST's initial implementation, although it was designed for HTN, used also a Java implementation of Graphplan, but afterwards its authors created an HTN planner written in Prolog. LOGTELL uses a non-linear planner written in Prolog as well, and Mimesis the Longbow planning system. I-Storytelling's initial implementation was using an HTN planner (using depth-first left-to-right search with heuristics), but since moved to HSP. Othello uses HSSP, which is a state-space forward-search planning system, guided by an adaptation of HSP. Façade uses a reactive behaviour planner, and finally, PaSSAGE does not provide information about its algorithm.

For DIEGESIS, we have created a planner which consists of a planning and a re-planning algorithm, able to generate plans of actions based on each agent's state and context, considering both the current world state and the available resources. The planner is aware of the available time (duration) an agent/character has for a plan when it is asked to generate one. Our planning algorithm is based on Graphplan for solutions expansion, and backtracking heuristic search for plan extraction, enriched with constraints satisfaction and dynamic opportunistic restart when required. The Planner is discussed in detail in section 4.14.

Regarding *representation languages*, most of the systems (Fabulator, GADIN, I-Storytelling, MIST, and Othello) use –or used in some of their versions– either the STRIPS language or adaptations of it, or languages derived from it like ADL and PDDL. A couple of systems (LOGTELL and the second version of MIST) represent their storyworlds in Prolog, the same language their planner is implemented with. Finally, the first version of I-Storytelling was modelled using an HTN representation, Façade uses ABL, and a couple of systems (Mimesis and PaSSAGE) do not provide such information.

DIEGESIS uses a combination of modelling approaches: The basic information for every level of a story is modelled using PDDL and further information such as information for each character, goal injection rules, choices and their fall-backs, etc. in XML. All the representation information including examples is documented in chapter 4.

In terms of *re-planning*, 3 systems (Façade, LOGTELL, and PaSSAGE) provide no information whether they support re-planning or not, Othello doesn't support re-planning, and GADIN mentions re-planning without giving much information about its mechanics. The rest implement different re-planning approaches: Fabulator re-plans from scratch; I-Storytelling by backtracking in the HTN; and both Mimesis and MIST pro-actively, trying to predict and repair faulty plans in advance.

DIEGESIS deals with the execution of the agents' plans in a higher level, and when a part of a plan fails, instructs the agent to re-plan based on its current knowledge of the state of the world. Considering that we modelled each agent to act as a real person in the way they generate and try to execute plans, it does not make sense (in our context) to predict and prevent plan failures, since a plan can fail either due to user intervention (which cannot be predicted), or intervention by other characters, or –in some cases– pure chance (discussed in section 4.10). In any case, failed plans due to "unpredicted" reasons are realistic and have the potential to enrich a generated narrative.

In our re-planning solution, as we interleave plan generation and plan execution, when a plan fails, we discard the already completed actions and we only re-plan for the failed (and some of the pending – discussed in detail in section 4.14) actions of the plan, merging the new partial plan with the unexecuted portion of the original plan.

Regarding *perspectives*, Othello does not provide information about it, half of the remaining systems (Fabulator, Façade, Mimesis, and PaSSAGE) use a first-person perspective to present their stories to the player, while the other half (GADIN, I-Storytelling, LOGTELL, and MIST) use a third-person perspective.

In its default mode, DIEGESIS presents the generated story as a whole. At any point during the generation of the story the player is able to view any action that a character

is executing, make choices related to any character, as well as view details about them (i.e. their current goals and plan). These abilities constitute a third-person perspective.

But, apart from the default mode, we want to provide the player with a first-person perspective as well, that's why we created the concept of vantage points (discussed in detail in section 4.12). If the player selects to view the story from the vantage point of a character she will view only the story outcome which is related to the chosen character, and will be available to interact with the story (i.e. make choices) only when an action is related to the story character. The generation of the rest of the story (which is unrelated to the selected character) will continue normally in the background (with the exception that any choices concerning other characters supposed to be made by the player will be made by the Oracle instead), yet invisible to the player. The player is able to choose between different vantage points or return to a full story view freely during run-time, allowing linear storyline with differing endings, interleaved storylines, and even flashbacks.

In terms of *interactivity*, in most of the systems (Fabulator, Façade, Mimesis, and PaSSAGE) the player is able to control only the protagonist; the rest of the characters are NPCs. In GADIN, the only player input is to resolve dilemmas. In LOGTELL there is only indirect/passive interaction during the generation of a narrative; in the dramatization phase there is no user interaction. In I-Storytelling, the player is able to wander in the 3D world as an invisible avatar and interact with objects, as well as to provide advice to NPCs via speech input. Finally, MIST and Othello does not provide any information on interactivity.

In DIEGESIS, there is not a main character that the player controls/observes; instead, the player can make choices (defined by the storyteller) for actions that can affect every character in the active story. Also, as we already explained before, the player is allowed to select and view the story from the perspective of any of the characters (in the default view mode, the story is presented as a whole), and to be able to switch between them without any limitations, during the generation of the narrative.

As we just mentioned, DIEGESIS includes a concept similar to GADIN's dilemmas, in the form of "choices". The storyteller can mark any kind of action as a choice. When such an action is about to occur, DIEGESIS either makes a choice itself, or asks the player to

make a choice whether the action will happen or not. The idea behind choices in DIEGESIS is that important decisions throughout the story should be marked as choices so they can potentially alter the outcome of the generated narrative. User interferences are always accepted by DIEGESIS and they affect the generation and execution of the story in real time so the player will be able to form the story in the way she wants, no matter how much impact they have on the generated narrative.

Regarding the *extendibility* of the systems, the information provided by the systems themselves is scarce. According to (Cooper, 2011), Fabulator has source files for the planner which can be modified but no editors or source code distribution, Façade is not designed to be modified therefore there are no tools available, GADIN provides no tools but new stories are possible if hard coded, and LOGTELL has no tools available. The rest of the systems (I-Storytelling, Mimesis, MIST, Othello, and PaSSAGE) do not provide such information.

To make the authoring process easier for DIEGESIS, we are using a PDDL editor created by (Cooper, 2011), and although the authoring process in XML is quite easier compared to PDDL, we have designed an XML editor as an extension to the PDDL one.

Finally, regarding *audience*: Fabulator has a general audience; Façade's audience is adults; GADIN fits best soap opera fans (and possible children based on a children story they modelled); I-Storytelling's audience is sitcom fans (since they modelled situations based on the famous sitcom Friends™); LOGTELL's audience can be anyone depending on the story; MIST's and PaSSAGE's audience is computer game players; Mimesis and Othello does not provide enough information to categorise them.

We believe that DIEGESIS is both suited for movie-like experiences including relatively long-length finite stories, as well as shorter stories, since it provides to the storyteller the flexibility required to experiment with multiple genres and lengths of stories. Therefore, DIEGESIS' audience could be anyone, depending on the story.

2.12. Re-planning outside of the DIS field

Moving away from the DIS field, there is research dealing with re-planning in several different fields, using multiple approaches.

For example, in (Zhang et al., 2007) a distributed graph planning algorithm is used by the agents to generate a plan collectively in a distributed manner, and re-plan accordingly. As we previously mentioned, DIEGESIS instructs each agent to generate and execute a plan individually. If at any point during execution the plan fails, re-planning occurs only for an individual agent.

A hybrid FastForward and HTN re-planning approach is explained in (Klusch et al., 2005, Klusch and Renner, 2006), in which the re-planning is being performed off-line. In (Van Der Krogt and De Weerdt, 2005), the re-planning approach is to generate a number of sub-plans (by removing actions from the initial plan), and then calculate heuristic values for each one of them to decide which is the best candidate to expand, so a new valid plan can be constructed.

In (Fox et al., 2006), the authors use a solution based on LPG algorithm and investigate the efficiency of repairing a plan versus re-planning from scratch. The approach considers plans which have their initial state and goals modified, and do not focus on re-planning during the execution of a plan.

As we already mentioned, our solution is focused on re-planning during the execution of a plan in real time. The re-planning is being performed using the planner we have created and is based on Graphplan for solutions expansion, and backtracking heuristic search enriched with constraints satisfaction and dynamic opportunistic restart when required.

In this chapter we presented the background and the related work of our research area. More specifically, we discussed about the field of DIS, about multi-agent systems and presented some of the relevant agent architectures, and about DIS-related as well as multi-agent-related planning and re-planning. We also presented some of the planning algorithms which are typically used in DIS systems, along with some of the representation languages used by them. Finally, we presented some examples of re-planning outside of the DIS field, and we surveyed and critically assessed a number of DIS systems, stating their relation to our own work. In the next chapter, we will discuss the requirements and specifications of our DIS framework.

3 DIEGESIS DIS FRAMEWORK

In this chapter, we document the requirements and specifications of our multi-agent Digital Interactive Storytelling (DIS) framework, called DIEGESIS. The functionality of the framework's components will be described in the next chapter.

In the three chapters where we describe our framework in detail (i.e. chapters 3, 4, and 5), we used a number of UML diagrams, using the notation and recommendations made by (Fowler, 2003). More information about the use of UML in this thesis can be found in Appendix A.

To properly design our framework, we need to think about who will use it and what would be helpful to them, who will create the story, which are the needs of the stories that our system will be able to manage, and which are going to be the key requirements of our framework.

3.1. USER TYPES AND CHARACTERISTICS

There will be two types of users associated with DIEGESIS. Firstly, the person who creates the structure of a story to be used by our framework, and secondly, the person who is going to use our framework to interact with the already created story structure and view the outcome of it.

For the rest of this thesis, we'll call the first person the "*storyteller*", and the second one the "*player*". A storyteller, to be able to design and model the structure of a story that will be used in our framework, needs to have knowledge of the PDDL and XML languages. As we will discuss in section 4.18, to make it easier for the storyteller to generate the story data we will design and use a PDDL and an XML editor.

On the other hand, the characteristics of the player are more relaxed, since the only requirement is the ability to use a computer so he can interact with DIEGESIS via a Graphical User Interface (GUI).

As it is illustrated in Figure 6, a use case for a storyteller is to use any available editors to create a story to be used in DIEGESIS, and also play the story she created in DIEGESIS, usually for testing purposes, a use case which they share with the player.

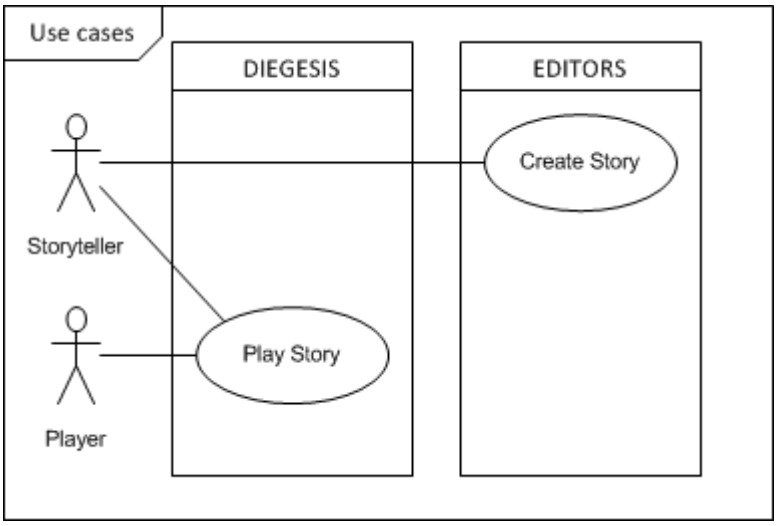

Figure 6: User types use case diagram

3.2. General Specifications

We want to build a scalable, abstract, DIS framework, which includes dynamic narration and story generation.

- **Scalable**: The framework needs to be able to accommodate multiple characters and levels. Therefore, during the implementation of our framework, we will have to constantly evaluate its performance, to ensure that the framework stays responsive and usable even when using large stories.
- **Abstract**: We intent to design the framework in the most abstract way we can, to be able to be used with any kind of story, instead of being highly coupled with one. That will enable the framework to be used in the future as a testing framework for planning and re-planning algorithms used in DIS.
- **Dynamic story generation and narration:** The storyteller has to model the elements of the story. Such elements can include characters, locations, items, actions, goals of the characters, etc. Our framework should generate the outcome of the story in a dynamic way (i.e. not predefined). To this end, we will create and use a planning and re-planning solution, which fits the needs of such a framework.

- **Interactive**: Since we are creating a Digital Interactive Storytelling framework, the resulting framework needs to be interactive. We intent to include a way for the player to be able to interact with the framework, altering the outcome of the story based on any choices made, as well as a way to allow the player to view and interact with the story from different vantage points.
- **Different points of view**: A different point of view (or vantage point) can dramatically alter the experience for a spectator or a participant, since it can change the context of a story. We want the player to be able to experience the generated story in different ways: both viewing and participating in the story as a whole, and viewing (and interacting with) the story via the "eyes" of a specific character, while the story progresses as usual. The framework should be able to alter these vantage points during runtime.
- **Decoupled**: The components of our framework should be created in a decoupled way when possible, to allow it to be embeddable to other systems. We need to be able to replace some of our components with others. For example, we want our framework to be able to be connected to a 3D virtual world representation that will deal with a visual representation of the generated story.

3.3. Choice of Base Representation Language

As we already discussed in chapter 2, there are many different description languages for representing planning problems. We decided to use PDDL (Planning Domain Definition Language) (Ghallab et al., 1998), which belongs to the STRIPS family, which is extensively used among planning algorithms.

To model a story into a planning task for PDDL, the following components are required as a minimum: a domain consisting of language requirements, types, predicates, and actions; and a problem consisting of objects, and initial state, and a set of goals. Figure 7 contains a simple example of a domain and Figure 8 an example of a problem definition.

```
(:requirements
    :typing :conditional-effects :equality :disjunctive-preconditions
)

(:types
    character location item - object
)

(:predicates
    (at ?x - (either character item) ?y - location)
    (has ?x - character ?y - item)
)

(:action walk-to
    :parameters     (?who - character ?from - location ?to - location)
    :precondition   (and
        (at ?who ?from)
        (not (= ?from ?to))
    )
    :effect  (and
        (at ?who ?to)
        (not (at ?who ?from))
    )
)

(:action pick-up
    :parameters (?who - character ?what - item ?where - location)
    :precondition   (and
        (at ?who ?where)
        (at ?what ?where)
    )
    :effect  (and
        (has ?who ?what)
        (not (at ?what ?where))
    )
)

(:action drop
    :parameters (?who - character ?what - item ?where - location)
    :precondition   (and
        (at ?who ?where)
        (has ?who ?what)
    )
    :effect  (and
        (at ?what ?where)
        (not (has ?who ?what))
    )
)
```

Figure 7: PDDL domain definition example

PDDL is a modular language. Each set of features are packed in a module, and can be included and used in a domain if they are declared in the requirements declaration. If a domain does not contain any requirements declaration, then the basic set of STRIPS requirements is assumed. The version 3.0 of PDDL (which we will use in our framework) includes the following requirements (Gerevini and Long, 2005):

- :strips – Basic STRIPS-style adds and deletes.
- :typing – To allow type names in declaration of variables.
- :negative-preconditions – To allow “not” in precondition and goal descriptions.
- :disjunctive-preconditions – To allow “or” in goal descriptions.
- :equality – To support “=” as built-in predicate.
- :existential-preconditions – To allow “exists” in goal descriptions..
- :universal-preconditions – To allow “forall” in goal descriptions.
- :quantified-preconditions – Combined declaration of existential and universal preconditions.
- :conditional-effects – To allow “when” in action effects.
- :fluents – To allow function definitions and use of effects using assignment operators and arithmetic preconditions.
- :adl – Combined declaration of strips, typing, negative preconditions, disjunctive preconditions, equality, quantified preconditions, and conditional effects.
- :durative-actions – To allow durative actions.
- :derived-predicates – To allow predicates whose truth value is defined by a formula.
- :timed-initial-literals – To allow the initial state to specify literals that will become true at a specified time point (implies durative-actions).
- :preferences – To allow the use of preferences in action preconditions and goals.
- :constraints – To allow the use of constraints fields in domain and problem files. These may contain modal operators supporting trajectory constraints.

Based on our needs, we have specified three base types of objects (i.e. characters, items, and locations) that can exist in a domain, which can be extended if required; for

example, there can be different kinds of items. Predicates are expressions that describe simple or complex states of the world in relation to the types we specified, which can be either true or false. In our example, a character or an item can be located at a specific location, and a character may have an item.

Actions are usually made up of three parts: parameters, preconditions, and effects. Parameters are variables which define the objects which need to exist for an action to be executed, as well as their types. The preconditions are the predicates related to the parameters which need to be either true or false for an action to be executed, and finally the effects are the predicates which are going to be true or false after an action is executed successfully.

```
(:objects
    tom - character
    mary - character
    living-room - location
    kitchen - location
    glass-of-water - item
    tv-remote-control - item
)

(:init
    (at tom living-room)
    (at mary living-room)
    (at glass-of-water kitchen)
    (at tv-remote-control living-room)
)

(:goal
    (and
        (has mary glass-of-water)
    )
)
```

Figure 8: PDDL problem definition example

In the problem file, we define the actual objects (based on the types we defined before) that exist in the story that we are modelling. We also define an initial state for all of the objects present, in the form of predicates. Goals are also predicates of a desired outcome for our story, and the job of the planner is to find a valid plan using the available actions to reach this outcome.

In the example we are using, the goal is that Mary has the glass of water. Since Mary is located in the living room, and the glass of water is located in the kitchen, the most likely outcome that the planner will produce will be that Mary will have to execute the "walk-to" action to move to the same location as the item she wants to acquire, and – as soon as this happens- the "pick-up" action, to have the glass of water in her possession.

3.4. Choice of Base Planning Algorithm

To aid us to decide which planning algorithm to use as a base for our solution, we performed an evaluation of planning algorithms with a DIS perspective in mind. In section 2.8 we documented the planning solutions some of the relevant DIS systems utilise.

When we had to make the choice of a base planning algorithm, there was only one paper available in the literature that investigated the suitability of general-purpose planning algorithms for DIS systems (Barros and Musse, 2007b), describing an approach to perform such an evaluation, so we decided to use this approach as well.

The approach was to benchmark different planning algorithms testing their performance to solve a specific problem in a specific domain and to compare their feature sets with DIS applications in mind. The feature sets considered valuable to DIS applications are the following: Support for extra language requirements; capability to generate partial-order plans; optimality; support for actions with costs; support for numeric variables.

Support for extra language requirements: As we have already mentioned, most of the planning algorithms have adopted PDDL as their input language and it is our choice as well. PDDL is a modular language, therefore planning algorithms are only required to implement a very basic set of its features. Every extra feature (requirement) supported by a planning algorithm adds expressive power to its input language (and enables the creation of more interesting actions from a storytelling point of view) or just eases the task of describing certain actions.

The five language constructs which are considered important are the following: Type hierarchies (:typing requirement); Built-in equality operator (:equality requirement),

Negative preconditions (:negative-preconditions requirement), Conditional effects (:conditional-effects requirement), and Existential preconditions (:existential-preconditions requirement.

Capability to generate partial-order plans: Total-order plans are sequence of actions without any sort of parallelism. In a DIS context, these actions represent story events. To be able to have actions occurring simultaneously in a story, partial ordered plans are needed.

Optimality: Optimal planning algorithms are guaranteed to produce the best possible plan in a given problem. We must keep in mind however that optimality can be misleading (e.g. a partial-order plan including unnecessary actions will be considered optimal if the metric of "parallel steps" is used).

Support for actions with costs: Many planners have a fixed metric that can be used to evaluate the value of the plan generated: the number of actions executed.

Support for numeric variables: Classic planning systems represent the world state as a conjunction of Boolean predicates which can be a limiting factor in the Interactive Storytelling (IS) field since almost nothing is (rigidly) black or white in real-life stories that an IS system is trying to generate. The use of numeric variables (in addition to Boolean variables) can be used in IS to go beyond this limitation.

The details of this evaluation are discussed in section 6.5.1. There, we discovered that there is no planning algorithm that combines all the characteristics described before. Therefore, we concluded that no planning algorithm can be considered ideal for DIS applications, and based on the available planning algorithms and considering that each DIS system has its own goals, the final choice of algorithm must be done based on the unique requirements of each DIS system.

We believe that a new planning algorithm (combining some features from existing algorithms with novel ideas) needs to be created specifically with DIS systems in mind. Extra attention to the expressiveness of its language must be given since it will help authors and researchers easily create better stories, the fundamental principle of every DIS system. Also, support for numeric variables, actions with costs and, possibly, capability to create partial-order plans would be desirable.

We published the evaluation (Goudoulakis et al., 2011) with the idea that the family of the FF planners (FF, Marvin, and Metric-FF) seem to possess a number of these capabilities (especially the latter) along with a good performance (they had some of the quickest times in solving the test problem) so they could be used as a starting point to our planner.

After we continued the design of our system though, we finally decided that our base planning algorithm would be Graphplan, since we wanted to be able to have more flexibility in the design of our planner and Graphplan provided that (several of Graphplan's features are used by the FF family algorithms anyway). Since its major lack of features comparing to the other solutions was the lack of support for the extra language requirements, we decided to extend the algorithm and include any requirements that we need while progressing with the implementation of our system.

3.5. Multi-agent Needs

The stories that DIEGESIS will generate based on the storyteller's modelling, will most likely include multiple characters. Each of these characters should be able to act as a real person, even if they play a very small part in the whole story. To elaborate on that, a character should have its own will (i.e. try to achieve his own goals), be able to generate plans to achieve his goals and act independently from another -if required- to do so, have knowledge of the world that he exists in, and be able to take decisions if needed.

All the above makes it clear that each character should be represented by an agent, which will make DIEGESIS a multi-agent system.

Each agent in the game world will use an instance of the Planner (i.e. the planning and re-planning algorithms of our framework; discussed in the next chapter) to be able to generate plans of actions and regenerate them if needed. The framework should be able to dictate the execution of the agents' plans, therefore the generation of the story, and should be able to coordinate them during the execution phase. Finally, to allow the framework to be as flexible as possible, there is not going to be a main character that the player controls/observes; instead, the player will be able to make

choices for actions that can affect every character in the active story, and -in extend- the outcome of the story.

As we already discussed in section 2.5, there are two types of multi-agent planning: centralised planning, in which a central agent is responsible to collect the partial or local plans of the other agents, to combine them in one plan and solve any conflicts that may occur, and distributed (a.k.a. decentralised), in which all the agents communicate with each other to generate their plans and to negotiate any possible conflicts.

In DIEGESIS, as we already mentioned at the beginning of this section, we want each agent (i.e. character) to operate as a real person. Relating that to the planning process, we want each agent to be able to generate its own plans based on each own goals and try to execute them individually and opportunistically. We believe that this provides a more realistic approach to the generation of a story, since by this way each agent can act as a real person, generating an autonomous plan considering only its own needs. This approach is similar to the description of decentralised planning.

Decentralised planning involved that agents communicate only with each other to negotiate conflicts, etc. We don't use that approach. Instead, although the plans are individual, we want DIEGESIS to dictate the execution phase of the agent's plans (therefore the generation of the story) so the system can have a better control and understanding of what happens during the generation/execution of the story, and to be able to interfere if needed. Therefore, in the case of plans execution, our approach borrows the control and coordination concepts from the centralised planning approach.

In this chapter, we documented the requirements and specifications of our multi-agent DIS framework. In the next chapter, we will document and discuss in detail the design aspect of every component of our framework.

4 DESIGN OF THE FRAMEWORK

In this chapter, we discuss in detail the design aspect of every component of our multi-agent Digital Interactive Storytelling (DIS) framework. As described in section 1.3, while designing and implementing the framework we used an incremental and iterative process. The work reported in the design and implementation chapters is the result of the aforementioned process.

To achieve our needs, we designed DIEGESIS as a multi-agent Digital Interactive Storytelling (DIS) framework using planning and re-planning techniques. DIEGESIS consists of several different components, each responsible for one or more features of the framework. The design of the framework and its components evolved while progressing with the implementation and the evaluation of the system, to keep up and comply with the evolving nature of a research project's requirements and specifications.

Figure 9 depicts DIEGESIS' high level architecture that we used in some of our publications, and illustrates the framework's main components at the time. There have been some changes since then since some of the sub-components of the main components grew and became main components themselves, as well as new components were added, but most the processes of the system remain the same, so we will briefly discuss how the system initially operated.

As we discussed in section 3.1, there are two types of users; the storyteller and the player. The Storyteller models the story in a set of XML & PDDL files, and the Parser component is responsible of translating them into a representation the framework understands and feed them to the World Manager (WM), which is the main component of the system and coordinates the rest. The WM stores this information to the Knowledge Base component, and uses it to update the environment which is perceived by the multiple instances of the Agent component (each Agent represents a character in the story).

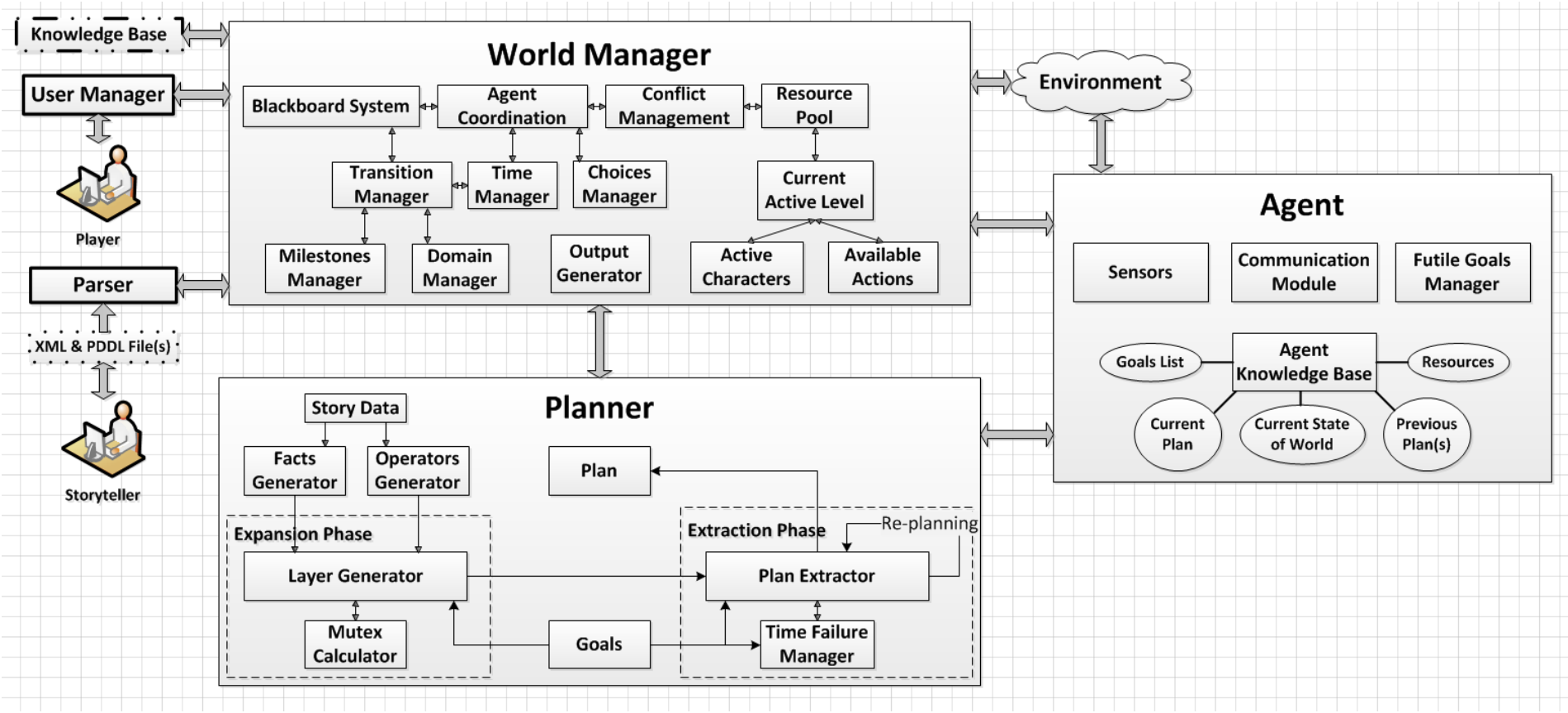

Figure 9: DIEGESIS' initial architecture

The planner consists of a planning and a re-planning algorithm able to generate plans of actions based on each agent's goals considering the current state of the environment as the agent perceives it.

The User manager is responsible of communicating with the player to either receive the player's interaction, or to show him the outcome of the generated story. As we already mentioned and is illustrated in **Figure 9** as well, the WM component included at the time several other sub-components. Eventually, as the framework grew and extra functionality was designed and implemented, most of these sub-components grew enough to become components by themselves, something which also promotes the modularity of the framework.

Figure 10 illustrates the final architecture of the DIEGESIS DIS framework. The finalised components are the following:

- *Parser*: It is responsible for parsing and processing the storyteller-created files.
- *Knowledge Base*: A centralised repository of information, including a relational database and information stored in memory. The Knowledge Base component stores information about the currently active story.
- *Level Manager*: It is responsible of keeping track of most of the information about each possible level (i.e. scene) of the story, and distributing this information to other components when required.
- *World Manager*: It is the main component which coordinates the whole system so the stories can be generated and executed. Its sub-components include the Agents' Manager, the Blackboard System (to communicate with the agents), the Time Manager, and the Output Generator. It also keeps an up-to-date representation of the world and is responsible for distributing it to the agents when required.
- *Choices Manager*: Based on the modelling of the story by the storyteller, the player may be able to make choices about important circumstances occurring while the story is being generated and executed. This component is responsible for dealing with them.

- *Transitioning Manager*: The component is responsible for performing a transition from a level which was just concluded to a new one which makes sense in the context of the story.
- *Goal Injection Manager*: It is responsible for injecting goals to the agents based on specific conditions specified by the storyteller.
- *Futile Goals Manager*: A component responsible of providing futile goals to the agents which are idle.
- *Oracle*: In certain situations during the generation and execution of a story, a relatively random outcome needs to be calculated. This component is responsible for doing that.
- *Uncertain Actions Manager*: There are some actions that make sense that they should have a percentage that will succeed (or fail) due to pure chance. This component deals with them.
- *Vantage Point Manager*: During the execution/generation of a story, the player is able to choose freely between different characters' vantage points (i.e. to view the story from the perspective of a specific character) and a full story view, and this component deals with these vantage points.
- *User Manager*: It contains a graphical user interface to communicate the story outcome and other relevant information to the player, and receive user input when is required.
- *Planner*: As we already mentioned, it consists of a planning and a re-planning algorithm able to generate plans of actions based on each agent's goals considering the current state of the environment as an agent perceives it.
- *Agent*: Every character in a story is represented by an agent. The component's architecture follows a hybrid approach including elements of reactive, deliberative, and BDI agent architectures.
- *Battle Manager*: There are cases in the evaluation scenario that we built (discussed in chapter 6), in which we need large-scale battles to occur; therefore, we built a component which deals with them.

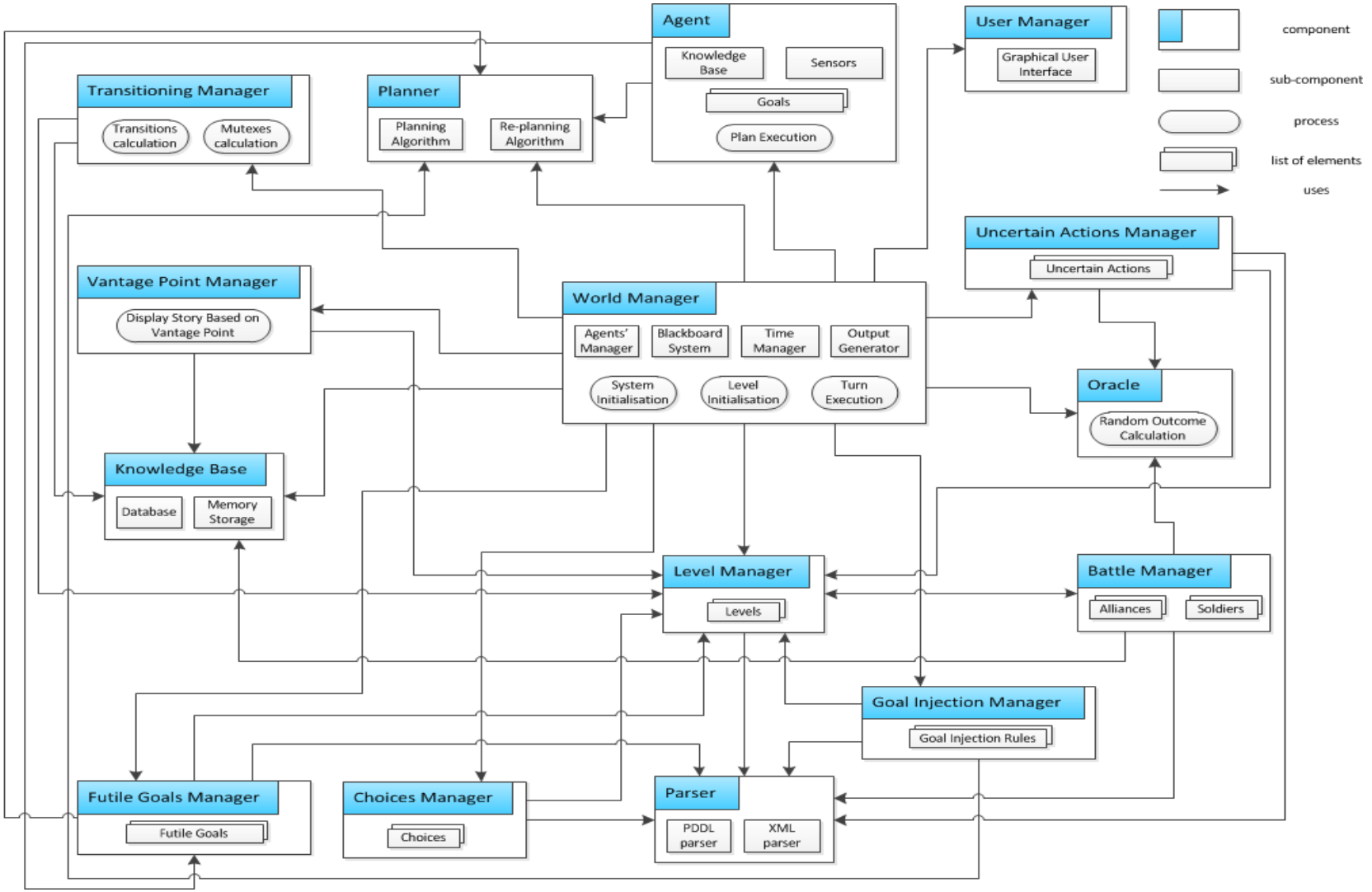

Figure 10: DIEGESIS architecture

In the following sections, we will discuss all the components of the, explaining in detail their processes and how they are operating together with other components.

4.1. Game World Architecture

The game world is created before the execution of a story, by a storyteller. The world is organised in multiple levels which can represent possible parts of a story. Typically, a level represents a broad area where a number of events in a story may occur. The levels are organised in a hierarchical manner; each level may have some potential successor levels which have a logical connection with it. An example of a game world is illustrated in Figure 11.

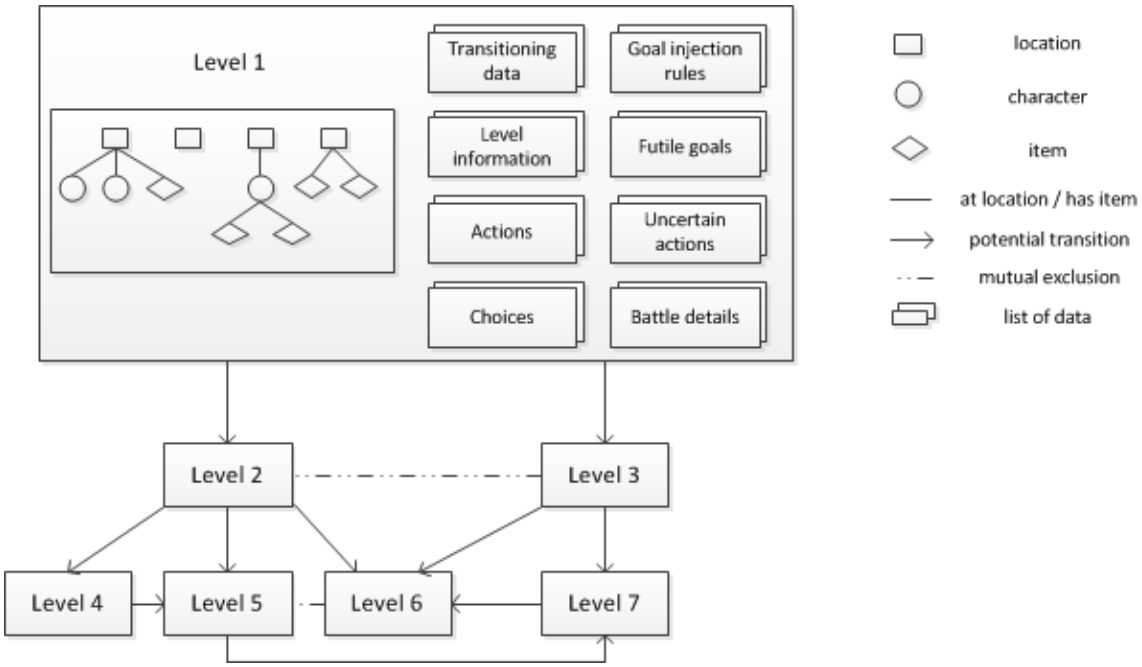

Figure 11: Game world architecture

A level –and in extend the game world which will produce a story– consists of the following elements:

- Locations which can be either small such as rooms or large such as whole countries.
- Characters along with their individual information which will be discussed in detail in section 4.15, such as their list of goals.
- Items which can be anything.
- Actions which are applicable in a level and can be executed by the characters based on certain conditions.

- Information about the level (such as its title, etc.), which will be discussed in section 4.4.
- A set of choices which are potential decision making moments for either the player or the framework and are based on rules specified by the storyteller, which will be discussed in section 4.6.
- A set of transitioning data such as potential successor levels, milestones, etc. (will be discussed in detail in section 4.7) which will be used to perform a transition to a new level as soon as a level comes to an end.
- Goal injection rules which will be discussed in section 4.8.
- A set of futile goals which can be assigned to a character if is idle, which will be discussed in section 4.9.
- A set of uncertain actions which will be discussed in section 4.11.
- Information about a large-scale battle which may occur in a level, which will be discussed in section 4.16.

The minimum mandatory elements that must exist in a level so DIEGESIS can process it consist of a set of locations, characters, and actions; everything else is optional.

As we mentioned in section 3.3, each level's main representation is modelled in PPDL. That includes the locations, characters who are present in a level (specifying in which location they are initially located), items present in a level (associated either with locations or with characters), and a set of applicable actions for a level.

An example of a PDDL representation of a part of the story we are using (discussed in detail in section 6.1) is displayed in Figure 12, where we omitted some information to ensure readability. An initial (default) state of all the characters and items in a level needs to be defined by the storyteller, but it can be dynamically altered based on events that occurred in previously executed domains.

```
(:objects
    helen - character
    menelaos - character
    paris - character
    hector - character
    throne-room - room
    private-room - room
    guest-room - room
```

```
    docks - room
    troy - location
    gift - item
    troy-ship - transportation-method
)

(:predicates
    (at ?x - (either character transportation-method) ?y - location)
    (has ?x - character ?y - item)
    (in-discussion ?x - character ?y - character)
    (emotion-loves ?who - character ?whom - character)
)

(:init
    (at menelaos throne-room)
    (at helen private-room)
    (at hector docks)
    (at paris docks)
    (at troy-ship docks)
    (has hector gift)
    (emotion-loves paris helen)
)

(:action talk-to
    :parameters (?x - character ?y - character ?z - room)
    :precondition (and
        (at ?x ?z)
        (at ?y ?z)
        (not (= ?x ?y))
    )
    :effect (and
        (in-discussion ?x ?y)
    )
)

(:action seduce
    :parameters (?who - character ?whom - character ?where - location)
    :precondition (and
        (at ?who ?where)
        (at ?whom ?where)
        (in-discussion ?who ?whom)
        (emotion-loves ?who ?whom
        (not (= ?who ?whom))
    )
    :effect (and
        (emotion-loves ?whom ?who)
    )
)
```

Figure 12: Example of a PDDL representation

Apart from the main representation of each level which is modelled in PDDL, the rest of the elements are further modelled in XML and we will discuss them in detail in the following sections.

4.2. PARSER

As we already mentioned, each story is written and modelled by the person who is creating the story, i.e. the storyteller. The modelling of the story world including levels, characters, locations, items, goals, milestones, available actions, etc. is stored in a number of files, available to the system. There are two types of files: PDDL and XML.

The Parser Component can be instructed by the World Manager (the component which coordinates the whole system and will be discussed later) or any other component to parse and analyse a set of files corresponding to a specific level, create a representation of them in the format needed, and communicate them back to the World Manager or the component which requested them to be used appropriately. This process is illustrated in Figure 13.

The files are parsed in an iterative manner. After each file is parsed, analysed, and the information it contains is passed to the component which requested them, the Parser checks if there are still files left in the queue to be parsed. If there are no files left, the process ends.

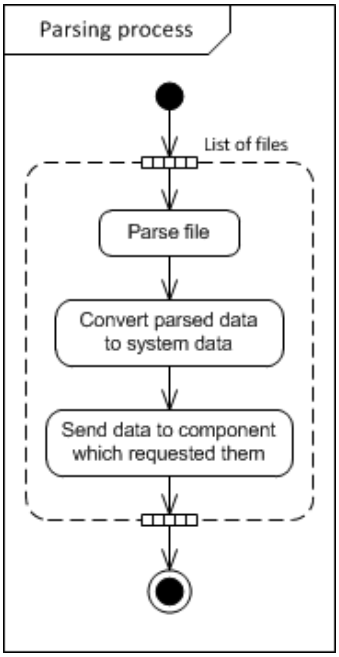

Figure 13: Parser activity diagram

4.3. KNOWLEDGE BASE

In our system's context, a Knowledge Base (KB) is a machine-readable centralised repository of information. DIEGESIS includes two types of KB; a relational database, and information stored in memory.

The memory-based part of the KB is responsible to keep information about the currently active level of the game world. The relational database includes tables about characters and their options, levels (and mutual exclusions between them), milestones, story actions, transitions, and information about the characters and any battle groups. A preliminary schema is illustrated in Figure 14.

The KB is populated during runtime by the framework, using data both from the information contained in the files created by the storyteller and parsed by the Parser as described in the previous section, as well as from information produced during the generation and execution of the story.

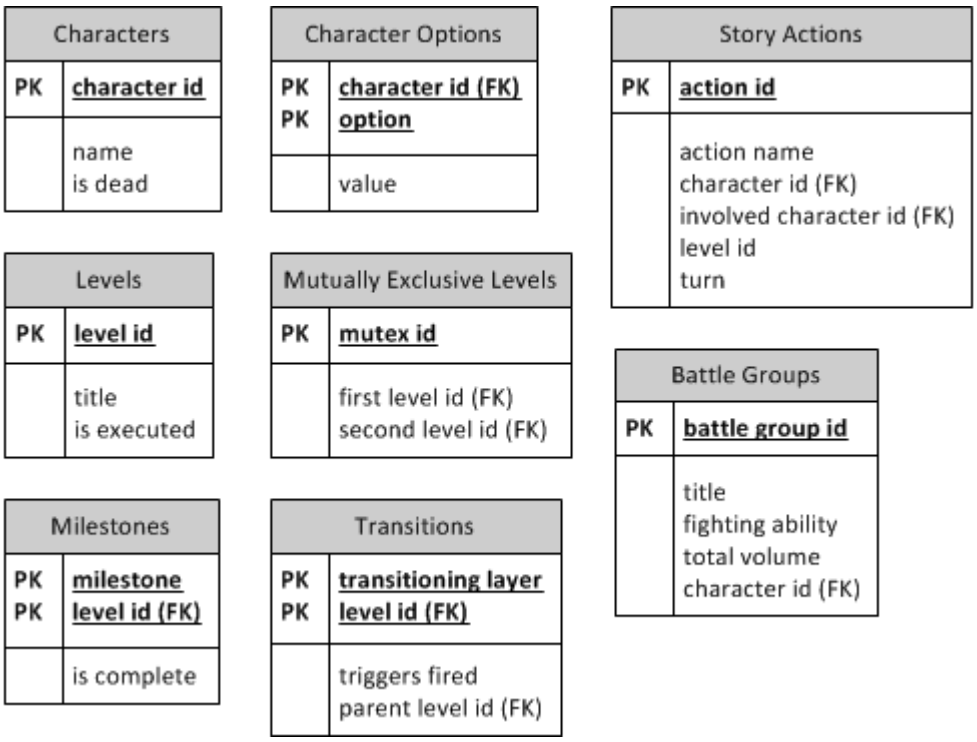

Figure 14: Preliminary database schema

Characters table include information about individual characters, such as a unique id, a name, and if the character is still alive and part of the story. Characters' options table is able to store characters' options. The table is designed in an abstract way, to allow the storyteller to represent any types of options. For example, a character based on his

previous actions, might need to be present at a certain point in the future of a story. The storyteller can create and store an option to keep this information.

Levels table contain information about the different possible levels which are present in the story, such as a unique id for each level, its title, and if it was executed. Any mutual exclusion between levels is stored in the mutually exclusive levels table.

The milestones table is related to the levels table, and is used to store the milestones of each level, and their state (i.e. are complete or not). Transitions table holds information about all the transitions, past and future, which occurred or will occur during the execution of the story, so the transitioning between levels, can be instructed.

The story actions table is used to store and keep track of all the actions that occurred during the execution of the story, along with information about them (i.e. the characters related to them; when and where the action occurred).

Finally, the battle groups table is used to store information about any battle groups present in the story, such as the group's title, leader (represented by a character), fighting ability, and total volume.

4.4. Level Manager

As we discussed in section 4.1, the game world is organised in multiple levels which can represent possible parts of a story. The Level Manager (LM) component is responsible of keeping track of most of the information about each possible level of the story.

Apart from the PDDL representation of each level which we explained in section 4.1, we intentionally omitted to explain in detail the list of centralised information about each level. It is a list of all the possible levels which may be executed during the generation of the story, containing important information about them. This information is modelled by the storyteller in an XML file, using the semantics presented in Figure 15.

```
<levels>
    <level>
        <title>level-title</title>
        <filename>level-base-filename </filename>
        <display_title>Level's Display Title </display_title>
        <info>A small description about the level.</info>
        <milestones>
            <milestone>(a-pddl-fact-may-be-a milestone)</milestone>
        </milestones>
        <is_battle_level>false</is_battle_level>
    </level>
</levels>
```

Figure 15: Semantics of XML level nodes

The mandatory information for each level includes the level's title, its base filename (so the rest of the level information mentioned in section 4.1 can be retrieved), a human-readable title and description of the level, and a flag informing the system if the level is a battle level or not. If it is a battle level, then further information about the battle is included which is discussed in detail in section 4.16.

Finally, each level includes three sets of triggers, a set of milestones, a set of potential successor levels, and a set of character options. All this information is optional and is used in the level transitioning phase. It is omitted in Figure 15 since it is further discussed in section 4.7.

When initialised by the World Manager (WM), LM uses an instance of the Parser to load all the information related to each level, translates them into a system-readable representation and stores whatever is needed to the Knowledge Base. It also has direct communication with the Battle Manager component to request any information needed which is related to a battle which may occur in a level. This process is illustrated in Figure 16.

Only one level of the whole story can be active at a time. LM is responsible to keep track of which level is active at a given moment in time, and keep it in memory so it can be easily accessible to the other components (such as the WM and the Transitioning Manager) when is required.

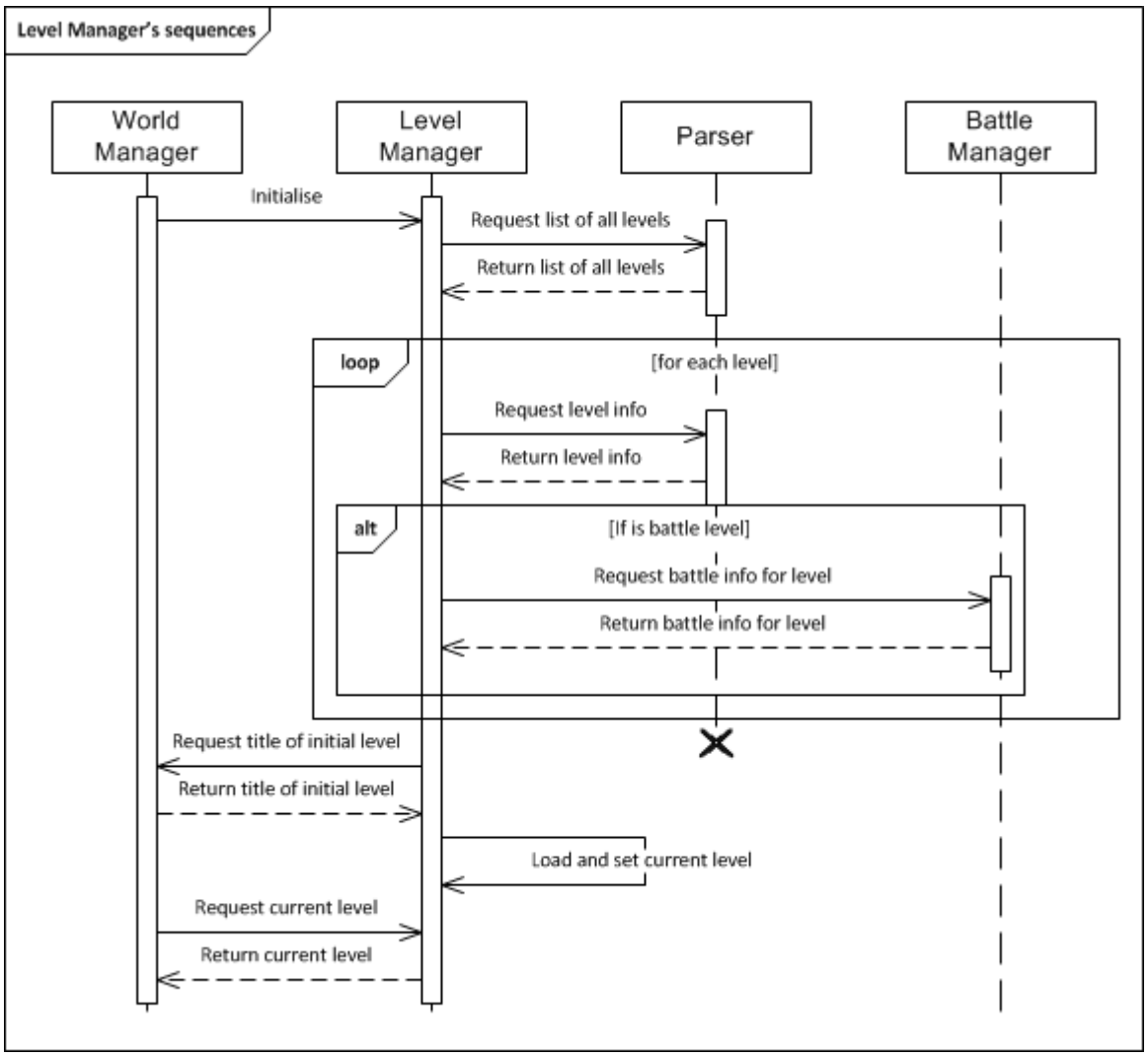

Figure 16: Level Manager sequence diagram

4.5. World Manager

The World Manager (WM) is the main component which coordinates the whole system. It has direct access to all the other components of the system, and (among other responsibilities) is responsible for keeping track of and updating the current state of the world, i.e. the environment the agents are aware of.

Figure 17 illustrates the high level functionality of the component. As soon as DIEGESIS launches, it initialises the system, by initialising most of the components which are going to be used during the generation and execution of the story, which are the following: Knowledge Base (KB), Battle Manager (BM), Level Manager (LM), Futile Goals Manager (FGM), Planner, Goal Injection Manager (GIM), Transitioning Manager (TM), Choices Manager (CM), Uncertain Actions Manager (UAM), User Manager (UM),

Vantage Point Manager (VPM), Time Manager (TiM), Parser, Output Generator (OG), and Agents' Manager (AM).

When the initialisation of each component includes further processes other than a simple enabling of the component, the explanation of each process is included in the sub-section in which each component is documented.

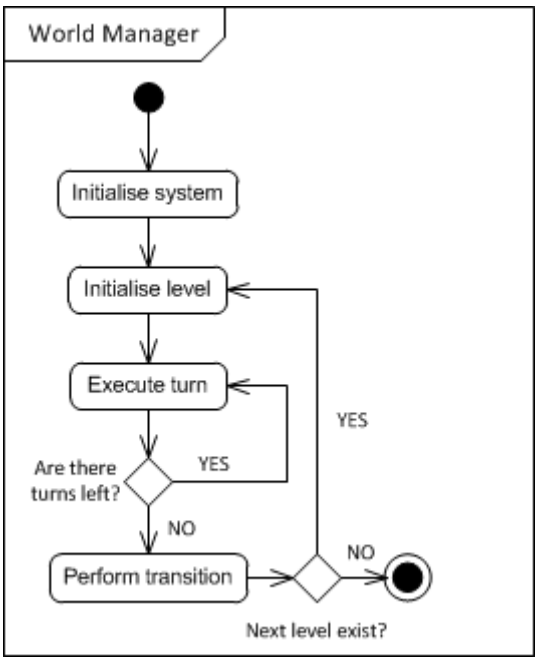

Figure 17: World Manager high-level activity diagram

The next step after the initialisation of the system is to initialise the currently active level. This process is depicted in Figure 18. Initially, the WM requests the information of the currently active level from the LM. Then, it sends the relevant information to the Planner, instructing it to perform an initialisation of the level based on the PDDL model of the level, and after this initialisation is complete, the WM requests the current state of the world, as well as the generated PDDL representation of the level which was constructed by the Planner.

Afterwards, the WM instructs a number of components to load a new set of information for the new level: the BM to load the battle details, the FGM to load the futile goals, the GIM to load the goal injection rules, the CM to load the choices, and the UAM to load the uncertain actions. Finally, the WM instructs the AM to initialise the agents of the new level, requests the lists of agents, and passes it to the UM.

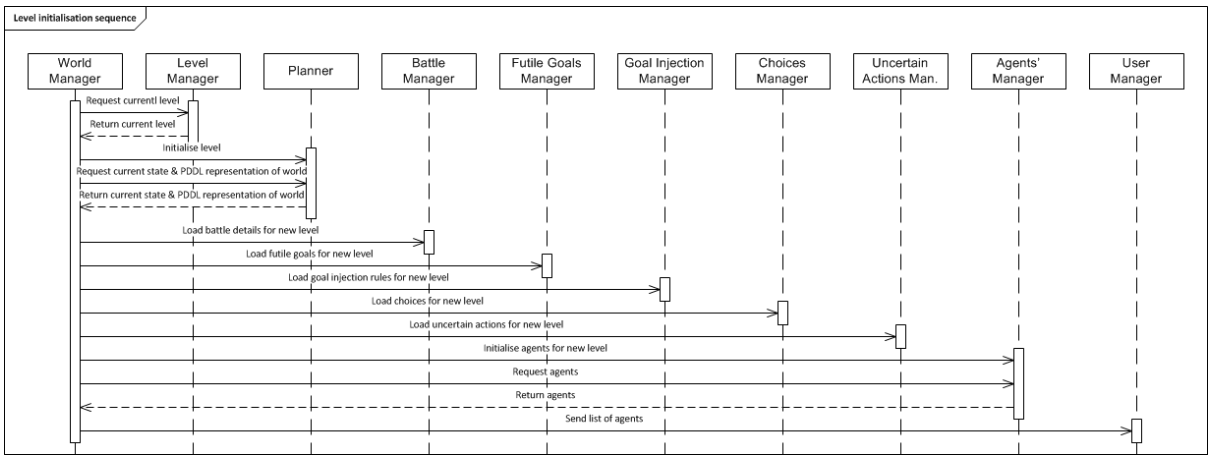

Figure 18: Level initialisation sequence diagram

The AM is a sub-component of the WM, which is responsible of managing the agents. It also keeps a list of the activated agents along with any information relevant to them, so they can be easily accessible when is required.

The initialisation of the agents that the WM requested is illustrated in Figure 19, and operates in the following way: Initially, the AM finds which characters are present in the currently active level by using the PDDL representation of the level which was previously created by the Parser. In the same manner, it identifies in which location each of the characters is initially located in.

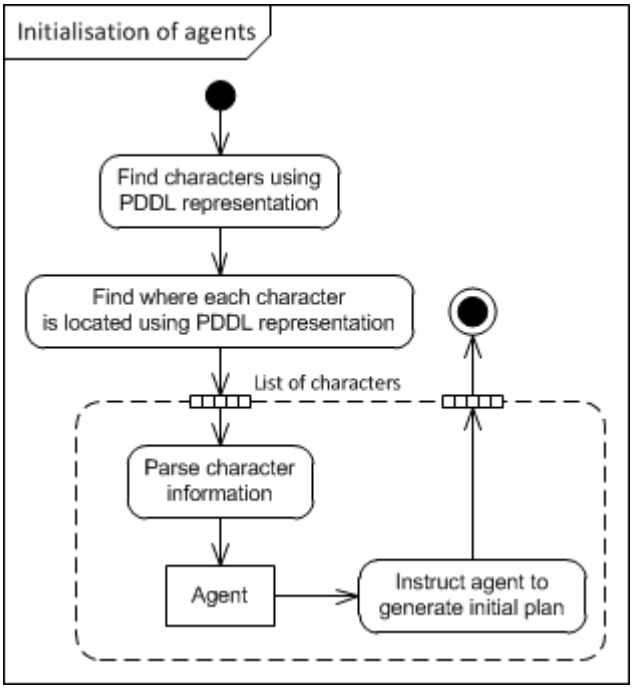

Figure 19: Initialisation of agents activity diagram

Afterwards, using the Parser, it parses all the character information for each one of the characters, and creates an instance of the Agent component for each individual character, feeding it with the parsed information. Finally, it instructs each created agent to generate an initial plan based on its current set of goals (if there are any).

The character information is created by the storyteller and is written in XML. A character node is required for every character present in each level. It is illustrated in Figure 20 and includes the PDDL name of the character that is related to, including the following information related to him/her:

- The available time to complete the specified goals and the PDDL goals list;
- If the character will be allocated with a futile goal when its goals' list is empty;
- The character's fighting ability which will be used if the character engages in battle during the execution of the level, along with the alliance in which the character belongs to;
- And a set of initial goals (which can change during runtime). Each goal node includes a name of a PDDL fact, (optionally) the importance value of the goal, and (optionally as well) one or more PDDL facts as preconditions.

```
<character>
  <name>paris</name>
  <futile_goals>disabled</futile_goals>
  <available_time>3600</available_time>
  <alliance>troy</alliance>
  <fighting_ability>70</fighting_ability>
  <goals>
    <goal>
      <name>(will-follow helen paris troy)</name>
      <importance>50</importance>
      <precondition></precondition>
    </goal>
    <goal>
      <name>(at paris troy)</name>
      <importance>100</importance>
      <precondition>(will-follow helen paris troy)</precondition>
    </goal>
  </goals>
</character>
```

Figure 20: An XML Character node

For the agents to communicate with the framework, DIEGESIS implements a blackboard system as an interconnection model. In our implementation of a

blackboard system, every agent communicates synchronously with the WM to access and update the shared knowledge base and coordination information, and not directly with each other. The WM continuously and in turns, updates each agent's knowledge of the current state of the world for any updates, and then "asks" them if there is an action to execute. Other communication (which will be discussed in detail in the remainder of the section) includes checking if an agent is busy, to instruct an agent to plan/re-plan, or wait, and to inject a new goal based on executed actions.

After the initialisation of the level, the WM begins the generation and execution of the story. The story is executed in turns. The execution process is illustrated in Figure 21. Initially, the WM informs the UM that the execution of a turn started so the UM can disable the next turn button. Afterwards, the WM checks if a battle is in progress with the help of the BM. If it is, then the next step is to check if an alliance needs to retreat. If the battle ended due to a retreat, then the current state of the world is updated with the retreat information. If not, a battle is performed. All of the battle-related processes are discussed in detail in section 4.16.

Then, for each individual agent the WM informs the agent of any changes in the current state of the world and checks if the agent is dead or busy (i.e. was either part of an action of another character or already involved in a battle). If it is, then the agent does nothing in this turn.

The WM then makes an inquiry to the agent, asking if the agent has a plan to execute. If the agent doesn't, then it is instructed to generate one before asked again if it has a plan. If it still doesn't, then the WM checks if the agent is cleared to fight (i.e. if there is a battle going on, and if the agent is in a battlefield location). If it does, then a battle versus a soldier of the opposite alliance is performed and the turn of the agent comes to an end.

If the agent has a plan to execute, then the WM requests the next set of actions from the agent. The generation and structure of a plan is discussed in detail in section 4.14. For each of the agent's actions, the WM first identifies all the agents who are involved in the action and checks if they are available (i.e. still alive and not busy) and if the action is interruptive. An interruptive action (set by the storyteller) will ignore the fact

that an involved agent (other than the one who executes the action) might be busy, and will be executed anyway.

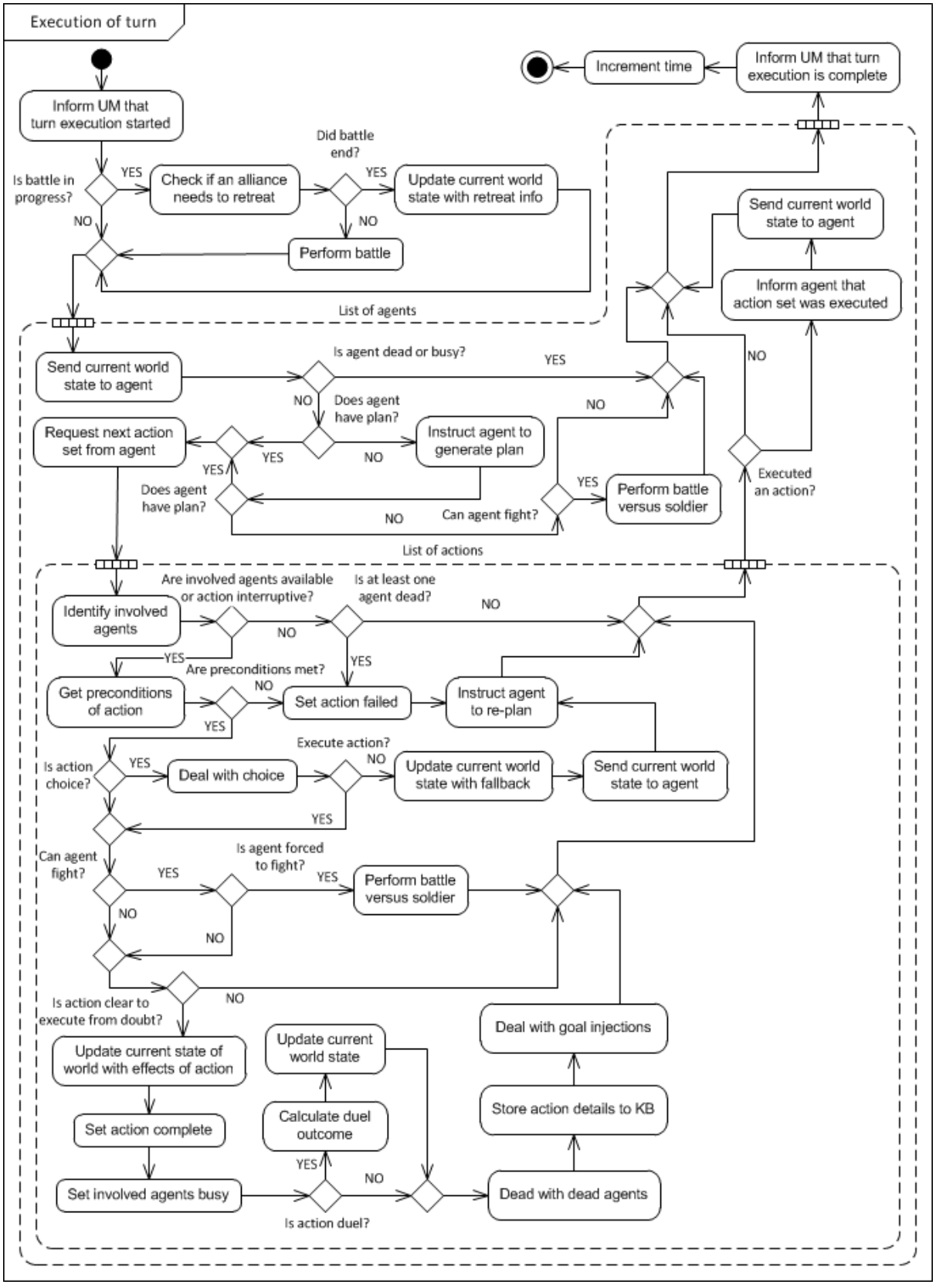

Figure 21: Activity diagram of the process of executing a turn

If any of the agents is not available (and the action is not interruptive), then the WM checks if at least one of them is dead. If not, then the action is not executed in this turn, but will be pending for execution at the next one. But, if an agent is dead, the action fails, and the agent is instructed to re-plan.

When the involved agents are available (or there are no involved agents in the action except from the agent who executes the action), the action's preconditions are checked by the WM against the current state of the world, to identify if they are met. If even one of them is not met, then the action fails, and the agent is instructed to re-plan.

In the case that the preconditions are all met, the next step for the WM is to contact the CM and identify if the action is marked as a choice. Again, all the choices processes are discussed in more detail in section 4.6. If the action is marked as a choice, the WM (with the help of CM) deal with the choice. When the outcome of the choice is the action not to be executed, the WM updates the current world state with the fallback(s) of the choice, sends it to the agent, and instructs the agent to re-plan.

On the other hand, when the outcome of the choice is that the action will be executed (or it wasn't marked as a choice in the first place), the WM checks with the help of BM if the agent can fight, i.e. if the agent is located in a battlefield location and there is a battle in progress. If these conditions are true, then there is a chance that the agent's action will be interrupted by a battle. The WM makes a decision with the help of the Oracle (discussed in section 4.10) whether the action will be interrupted or not (using an interruption percentage provided by the BM), and if it does, a battle is performed between the agent and a soldier of the enemy alliance.

If the action is still ok to be executed, the last check involves if an action is clear to be executed due to doubt. The WM makes the appropriate checks with the UAM (the details are discussed in section 4.11), and if the action cannot be executed in this turn it is ignored. But, if it doesn't then it is finally executed, and the following happen: The current state of the world is updated with the effects of the action, the action is marked as complete, and the involved characters are set to busy.

There is also a chance that the action is a duel. In the event that it is, the WM calculates the outcome of the duel (with the help of the Oracle) and updates the current state of the world with its outcome (i.e. who won and who lost the duel). The final step is to deal with any dead agents, to store the action's details to the KB, and to deal with any goal injections.

It is also important to mention that even if an agent is in a position to execute multiple actions in a single turn, it can only fight once, and if an action fails and the agent is instructed to re-plan, then the agent cannot try and execute any other actions during the same turn.

After the execution of actions is finished, the WM checks if at least an action was executed. If it did, then the WM informs the agent which of the actions were executed, and updates the current world state that the agent is aware of.

As soon as all of the agents finished their turn, the WM informs the UM that the turn execution is complete, it increments the time (with the help of the TiM), and completes the execution of the turn.

The TiM is a small sub-component of the WM, whose only responsibility is to keep track of the time steps (i.e. turns) in which the story is at any point, and feed this information to the WM when it's required.

After the end of a turn, the WM checks if at least one action was executed by an agent during that turn, if there is an active battle going on, and if any of the agents have a valid plan which is still pending completion and does not consist of futile goals. When none of the above conditions are met, the WM understands that the execution of a level is finished, and instructs the TM to calculate and perform a transition. If there isn't a suitable successor level, the story ends and the system shuts down. The details for the transitions are discussed in section 4.7.

The player who uses DIEGESIS to execute and interact with a story created by a storyteller needs to be able to view the outcome of the generated story, as well as other information relevant to the story. Therefore, we need a component which responsibility is to generate all these messages in a human-readable form. During all of the WM processes, the WM (as well as other components if it's required) uses the OG

to generate and display (with the help of the UM) appropriate messages to the player, as well as to the console for debugging purposes.

The OG includes some pre-defined templates to visually represent a different variety of messages. The templates include headers, sub-headers, alerts, and plain messages. An example of requests for printing messages in the console as well as displaying them to the player can be found in Figure 22.

There, the WM requests from the OG to print and display an alert message. The OG prepares the final message passing it through the alert template, and prints it to the console itself, as well as sending it to the UM so it can be displayed to the player. Afterwards, an Agent requests from the OG to print a plain message to the console, and display another message to the player. The OG prints the message in the console, and sends the other message to the UM requesting it to be displayed to the player.

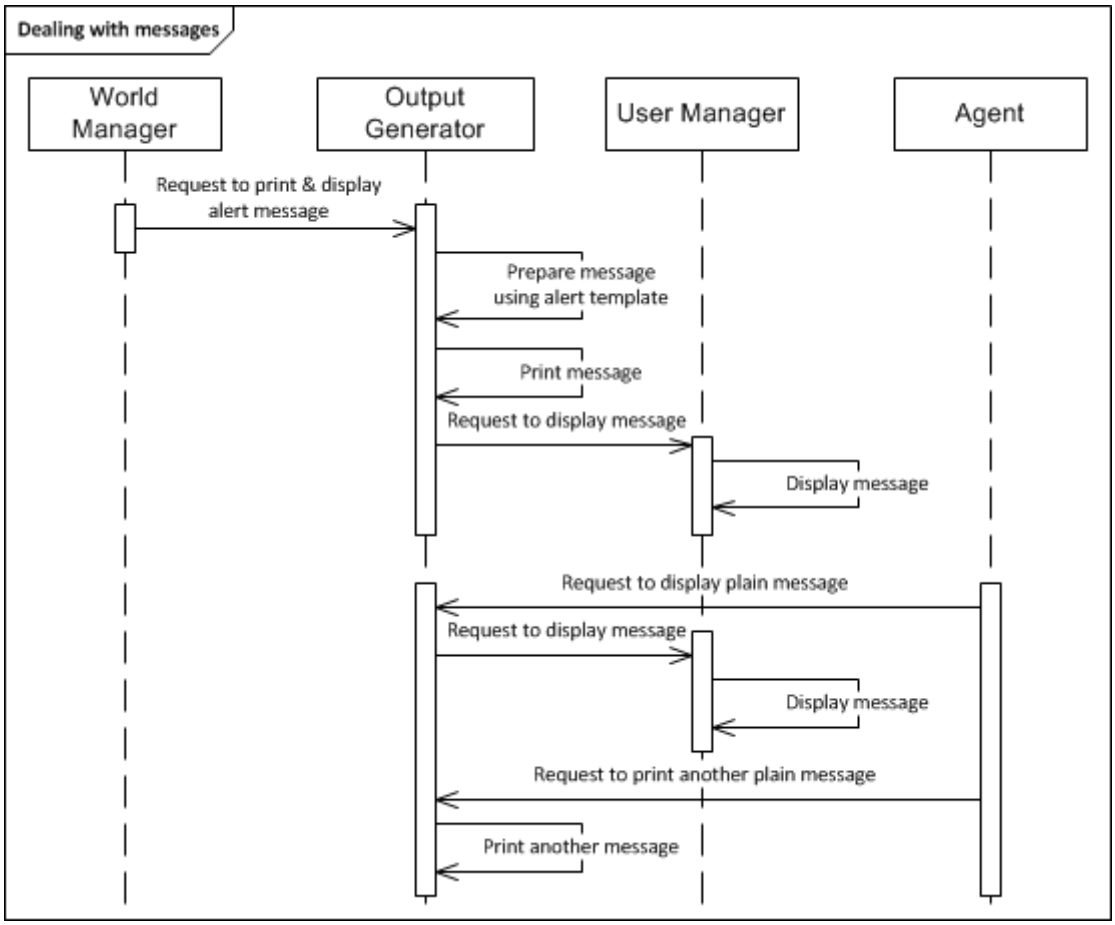

Figure 22: Sequence diagram of dealing with messages

Finally, before sending a request to display a message, the OG also checks if there is a vantage point enabled to determine if it's appropriate to display the message or not. This process is described in section 4.12.

4.6. CHOICES MANAGER

The storyteller has the ability to mark actions as choices, and the Choices Manager (CM) is responsible of keeping the relevant information. A choice node (illustrated in Figure 23) contains the name of a PPDL action that needs to be flagged as a choice. The choice can be made either by the player, or by the framework. If the action succeeds (chosen either by the framework or the player), then the normal effect of the PDDL action is executed.

```
<choices>
  <choice>
    <action_name> decide-if-will-plunder-temple</action_name>
    <who_decides>player</who_decides>
    <fallbacks>
      <fallback>
        <equals index="1">achilles</equals>
        <predicate_to_become_true>(decided-if-will-plunder-
temple)</predicate_to_become_true>
        <predicate_to_become_true>(will-not-plunder-
temple)</predicate_to_become_true>
      </fallback>
    </fallbacks>
  </choice>
  <choice>
    <action_name> decide-if-will-capture</action_name>
    <who_decides>player</who_decides>
    <fallbacks>
      <fallback>
        <equals index="1">achilles</equals>
        <equals index="2">briseis</equals>
        <predicate_to_become_true>(decided-if-will-capture
briseis)</predicate_to_become_true>
        <predicate_to_become_true>(will-not-capture
briseis)</predicate_to_become_true>
      </fallback>
    </fallbacks>
  </choice>
<choices>
```

Figure 23: A set of XML Choice nodes

But since PDDL doesn't offer the option of effects that are triggered if an action is about to be executed but failed (for any reason), we specify a set of fallback predicates (effects) which will be enabled if the choice is negative.

The fallback can be applied in any form of the selected action, or it can be applied only if there are specific conditions in an action, if for example the action is executed by a specific character. For example, in the second choice node of Figure 23, the fallback predicate will be enabled only if the first index (i.e. variable) of the "*decide-if-will-capture*" action is "*achilles*" and the second one is "*briseis*".

Every time that a new level is loaded, the World Manager (WM) instructs the CM to load the choices information for the new level (i.e. all the choice nodes which are relevant to the new level). The CM makes an inquiry to the Level Manager to receive the choices information about the currently active level, and using that it asks the Parser to parse and return the set of choices, which is stored in memory. This process is illustrated in Figure 24.

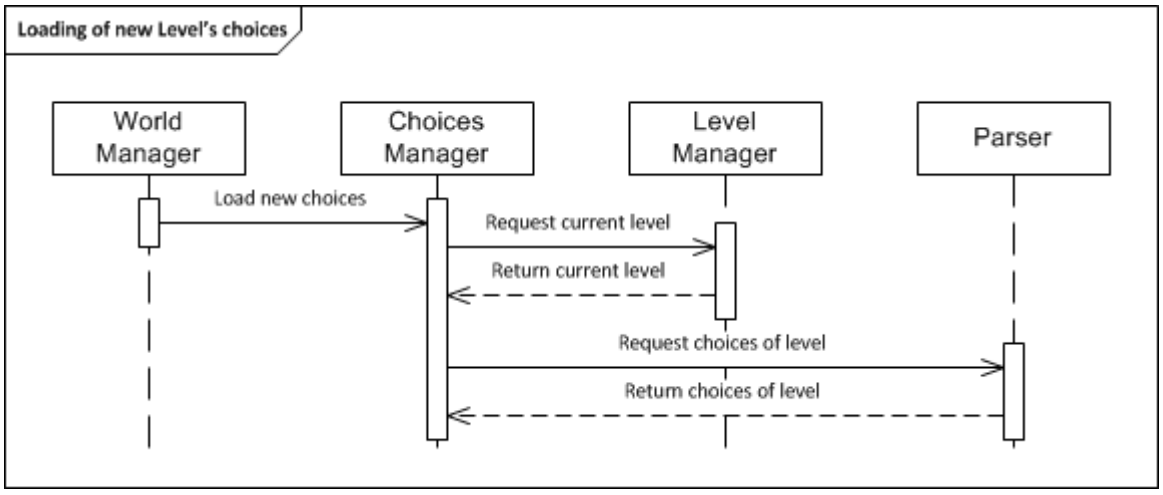

Figure 24: Sequence diagram of loading a new level's choices

While the story is generated and executed, every time that an action is about to be executed the WM checks with the CM to identify if the action is marked by the storyteller as a choice action.

When such an action is set to be executed, a decision needs to be made; either the action is going to be executed or not. Based on the storyteller's selection, either the framework will make a positive or negative decision, or the User Manager will be

instructed to stop the execution of the story and ask the player to make this decision. Based on the outcome of the decision, the action is either going to be executed, or not.

If the choice is negative, then the WM requests the action's fallback from the CM, and deals with it. This process is illustrated in Figure 25.

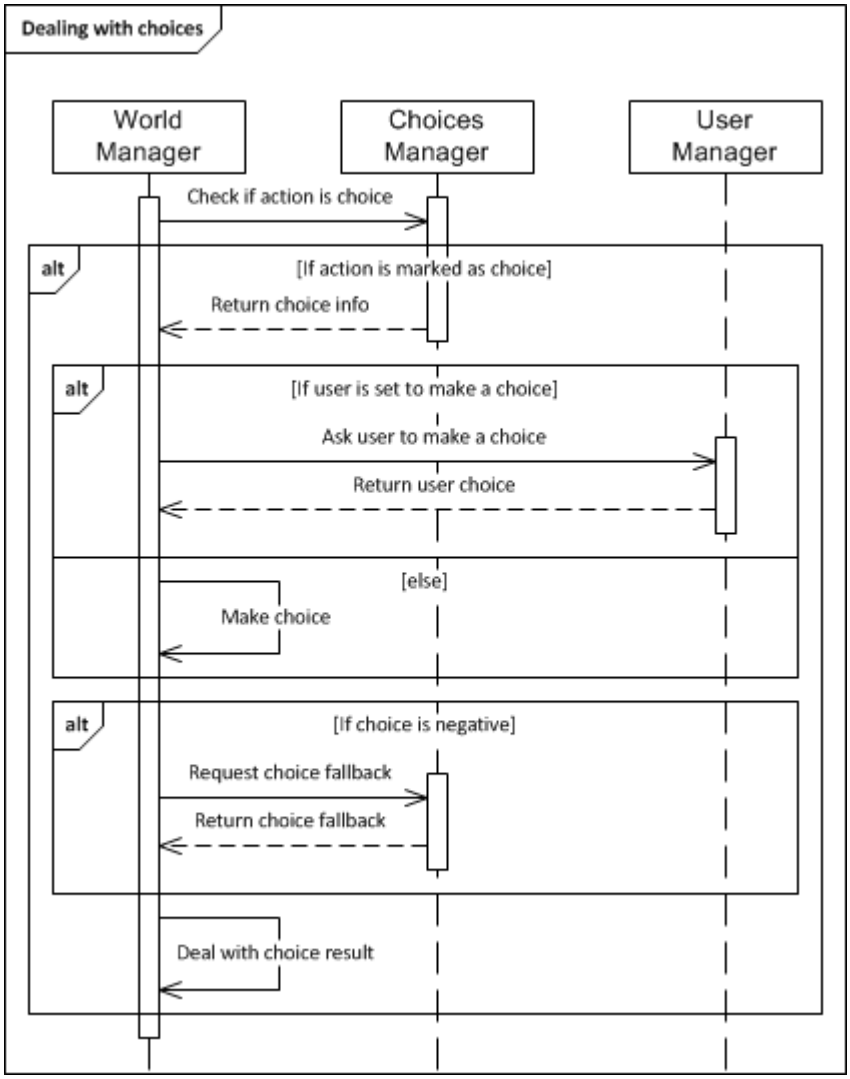

Figure 25: Sequence diagram of dealing with choices

If a decision is set to be made by the player but the player has chosen to view the story via the vantage point (discussed in section 4.12) of a character who is not involved in the specific decision, the framework will have to make the decision instead of the player.

4.7. TRANSITIONING MANAGER

As we discussed in section 4.5, every time a turn ends while the story is executed, the World Manager (WM) component constantly monitors the current state of the active level to identify if nothing significant is left (and is still able) to happen in the currently active level, or it can be terminated, and a next level can be loaded. When a level ends, it is a time for a new level to be selected and enabled, and that's what the Transitioning Manager (TM) component is responsible of.

As we already mentioned in sections 4.1 and 4.4, the story levels are organised in a hierarchical manner; each level may include some potential successor levels which have a logical connection with it. Each level may also include three sets of triggers, a set of milestones, and a set of character options. All this information is optional, is used in the transitioning phase, and an example of it is included in Figure 26.

A connection between two levels is being made by specifying a level as a "successor" of another one (using the specified title of a level). A level may include multiple potentially successor levels. "Milestones" are PDDL facts; when a level ends, their status (either true or false) is checked and the outcome is stored in the Knowledge Base (KB) component.

"Character options" are also checked as soon a level ends, but they are a little more complicated than a milestone. They include the option's title, the character who refers to (although it can be blank if they are general options), a set of options, a "then" value and an "else" value. Each option includes a PDDL fact and the condition the system will check if it exists. If all the facts' conditions are met, then the option is stored in the KB with the "then" value; if not, the option is stored with the "else" value.

In plain words, the "triggers" are used to identify if a level makes sense to be executed. They are preconditions, which need to be met for a level to be a successful candidate to be loaded into the system for execution. They can be milestones of another level, or character options, and they include an "expected" value to be checked against. They can also be marked as important.

The "triggers type" variable can take three different values, which relate to the main triggers: "all", "any", and "important". The default value is "any", meaning that the

level is a successful candidate if any of the triggers is fired. The “all” value means that all of the triggers must be fired, and the “important” value means that only the triggers marked as important need to be fired for a level to be executed.

```
<levels>
    <level>
        (…)
        <successors>
            <successor>title-of-a-potentially-successor-level</successor>
        </successors>
        <milestones>
            <milestone>(a-milestone-is-a-pddl-fact)</milestone>
        </milestones>
        <character_options>
            <option_group>
                <title>option-title</title>
                <character_name>mary</character_name>
                <then_value>true</then_value>
                <else_value>false</else_value>
                <options>
                    <option fact_condition="true" fact="(pddl fact)" />
                </options>
            </option_group>
        </character_options>
        <triggers_type>all<triggers_type>
        <triggers>
            <trigger expected="true" from_level="another-level-title">(another level’ s milestone)</trigger>
        </triggers>
        <knowledge_transfer>
            <character_triggers>
                <character_trigger char_name="mary" from_level="another-level-title" char_exists_if="true">
                    (another level’ s milestone)
                </character_trigger>
            </character_triggers>
            <fact_triggers>
                <fact_trigger fact_to_enable="(pddl fact)" from_level="another-level-title " enable_if="true">
                    (another level’ s milestone)
                </fact_trigger>
            </fact_triggers>
        </knowledge_transfer>
    </level>
</levels>
```

Figure 26: Transitioning information in XML

There are two more types of triggers, the “character triggers” and the “fact triggers”, which are checked when a level is selected as the next transition and is about to be executed. Character triggers work in a similar way to the main triggers, but they refer to characters. If they are fired, then a character exists in the level. Fact triggers (as their name suggests) refer to PDDL facts. If the milestone or the character option is met, then the trigger is fired and the “fact to enable” will be set to true when the level is loaded and ready for execution.

There are certain situations where a number of levels are mutually exclusive with others. The storyteller can specify them using the semantics presented in Figure 27. The mutual exclusions are grouped (each group needs a unique title), and each level has a priority value to identify which of the levels should be selected for execution in the event that a mutual exclusion situation appears.

```
<mutually_exclusive_groups>
    <group title="landing">
        <level priority="15">war-prevented</level>
        <level priority="10">troy-beach-landing</level>
        <level priority="5">troy-beach-landing-fallback</level>
    </group>
</mutually_exclusive_groups>
```

Figure 27: A group of mutually exclusive levels modelled in XML

As we already mentioned in section 4.5, as soon as the WM identifies that a level finished, it instructs the TM to calculate the next transition. This process is illustrated in Figure 28.

The first step of the process is to get the information of the current level from the Level Manager (LM) and the current state of the world from the WM. Then, for each of the milestones, the TM checks the state of the milestone against the current state of the world and stores the relevant information in the KB. The interaction between the different components is illustrated in Figure 29.

As soon as the checking of all the milestones is finished, the TM performs the same checks for the character options, and stores all the relevant information in the KB. Then, it creates a list of all the successor levels, getting the relevant information from the LM component.

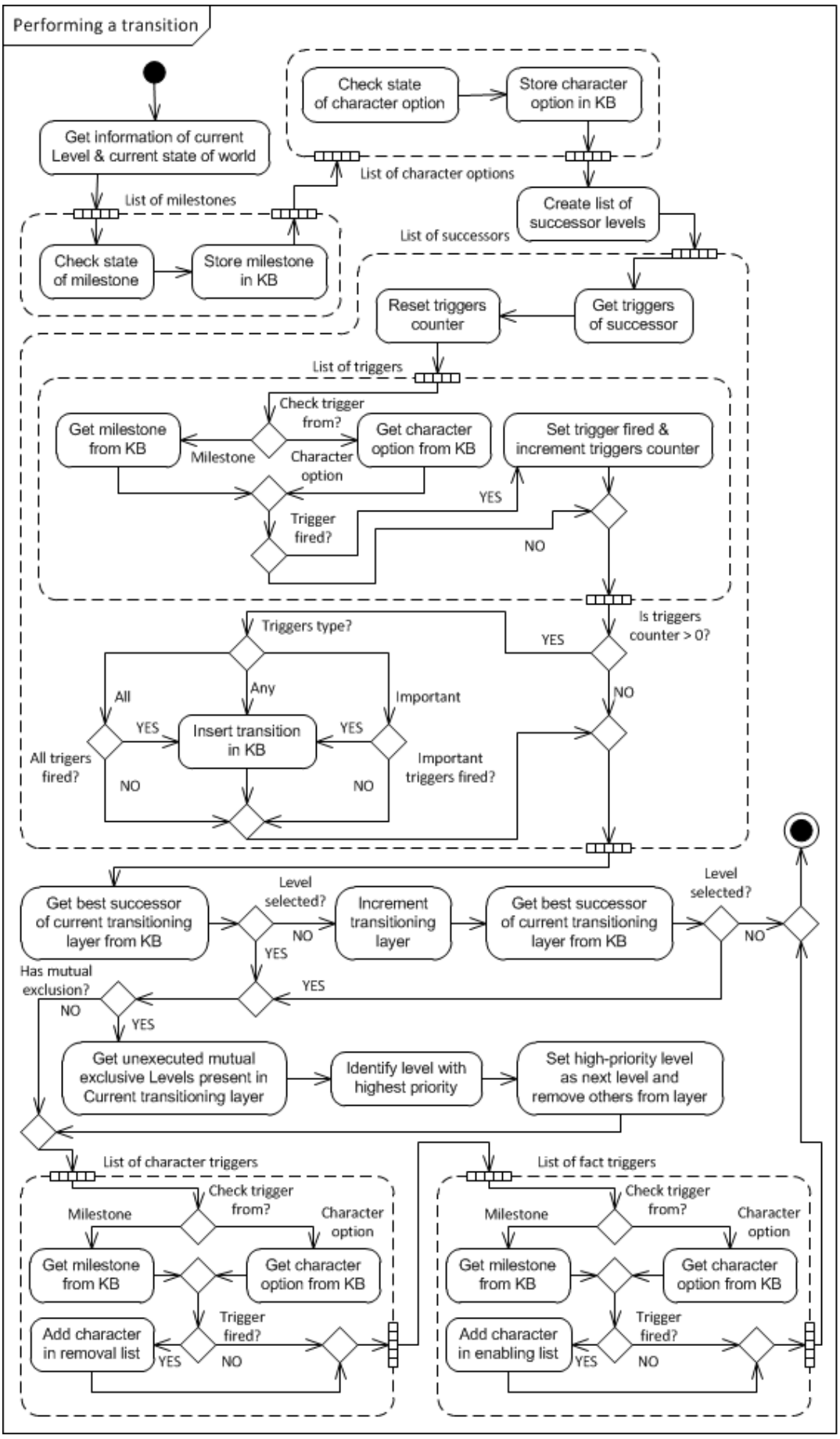

Figure 28: Activity diagram of the transitioning process

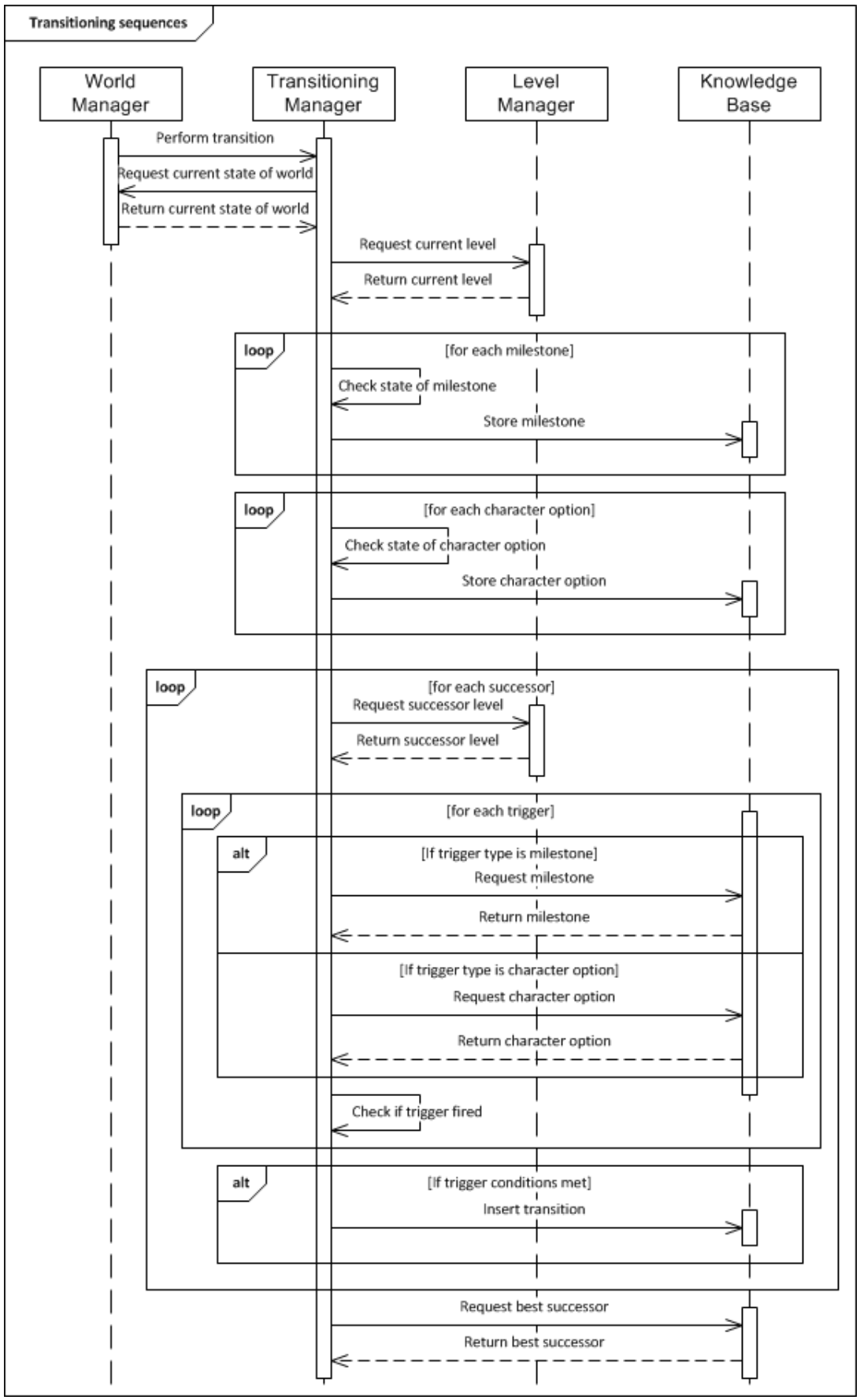

Figure 29: Transitioning sequence diagram

For each of the successor levels, the TM get a list of their successors and resets the triggers counter, which is responsible of counting the number of triggers which were fired. Then, it iterates through the triggers. If the trigger needs to be checked against a milestone, the TM requests the milestone from the KB, or if the trigger needs to be checked against a character option, the TM retrieves the character option's information from the KB instead. The value of either the milestone or the character option is checked against the precondition of the trigger, and if the trigger was fired is marked as fired and the triggers counter is incremented.

If at least one of the triggers was fired, the TM checks the triggers type of the level. If it is set to "any", then the TM creates a potential transition and stores the relevant information (i.e. the title of the level, the number of fired triggers, and the transitioning layer) in the KB. If the triggers type of the level is either set to "all" or to "important", then the TM checks if all of the triggers or the important triggers were fired respectively, and if they did, it adds the level as a potential transition. If not, then the level is ignored.

As soon as this process ends, then it is time for the TM to select the next level which will be executed. Before we describe how a level is selected, we need to explain the notion of transitioning layers.

An example of transitioning layers containing possible transitions is depicted in Figure 30. Layer 0 is the initial layer, which can only contain the initial level. As soon as this level is executed, the TM is requested to perform a transition. By performing the process we already described, the TM identifies that there are three levels which meet their preconditions and can be executed. All of them, are placed in the next transitioning layer (i.e. layer 1), since their content happens after the content of the already executed level. Every level which is in the same transitioning layer is supposed to happen in a simultaneous time as the others, although the player will "watch" and interact with them sequentially, just like in a movie.

For the sake of the example, we will assume that their execution order will be the following: First the Level 1, then the Level 2, and finally, the Level 3. So, as soon as Level 1 is executed and comes to an end, the TM identifies that there are two successor levels (i.e. Levels 4 and 5) which need to be added in the transitioning stack.

Since the current transitioning level is 1, they need to be placed in layer 2. The TM then continues executing Level 2 which does not produces any suitable successors after it's complete, and then it's the turn of Level 3, which produces four successor levels: Levels 5, 6, 7, and 8, which are also added in layer 2 (since Level 5 already exists in the layer is not added for a second time).

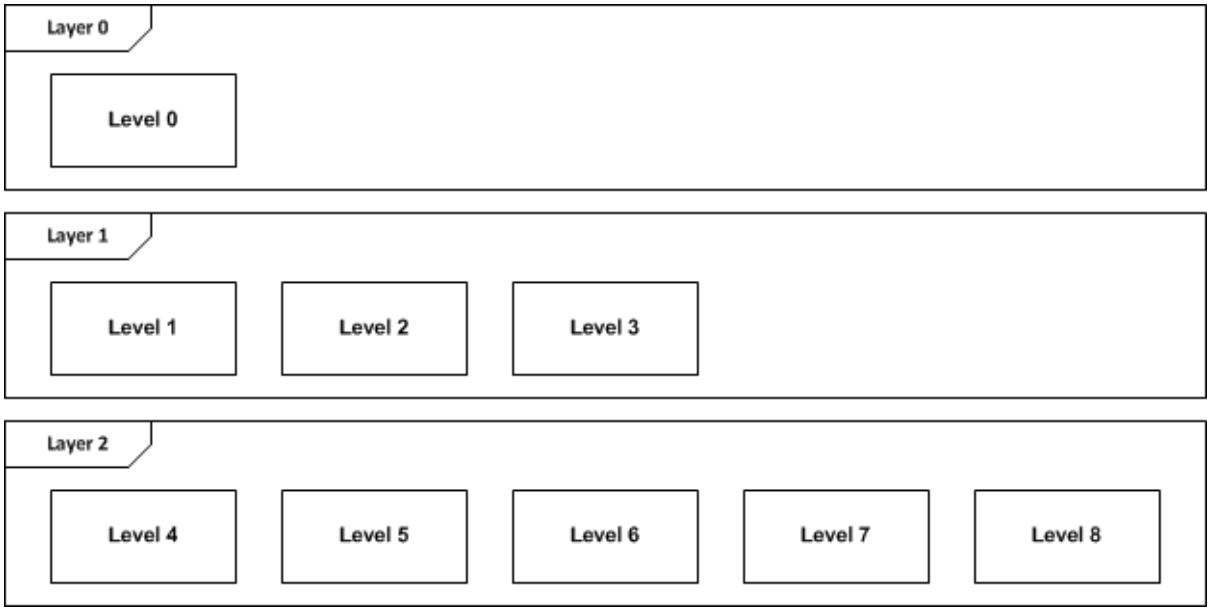

Figure 30: Example of transitioning layers

So, to summarise, all the levels which are present in the current transitioning layer need to be executed before we move to the next transitioning layer, and any successors of them are placed in the next layer.

Having explained that, we will return to the main transitioning process. The TM has already identified and added to the KB all the suitable successors of the completed level, and needs to select the next level which will be executed. To do that, the KB requests from the KB to return the title of the unexecuted level of the current transitioning layer which has the most fired triggers, although the execution order of levels present in the same level does not really matter since they are supposed to happen in the same time, therefore they should not be able to alter any information which is used in the levels of the same layer.

If the KB does not return a level it means that there are no levels left to execute in this transitioning layer, so the TM increments the transitioning layer and asks again for a

level from the KB. If there is no level in the next layer as well, that means that the story has come to an end.

As soon as the TM is in possession of the next level, it checks if there is a mutual exclusion specified for it. If there is one, then the TM requests from the KB all the levels which are present in the same transitioning layer as the selected level, and are still unexecuted. A mutual exclusion applies only for the current transitioning layer; therefore if a mutual exclusive level is present in a future layer will not be affected. As soon as the list of levels is constructed, the TM identifies which of the levels has the highest priority, sets it as the level which is going to be executed next, and removes the rest from the layer.

We extend the example of Figure 30, in Figure 31. There, the execution of Level 4 was just completed (all the levels with a striped background are complete) and produced two more levels (9 & 10), which are added in the next transitioning layer (i.e. layer 3). The TM identifies that the next level that should be executed is Level 5. But, it also identifies that this level belongs to a mutual exclusion group along with Levels 6, 7, and 10.

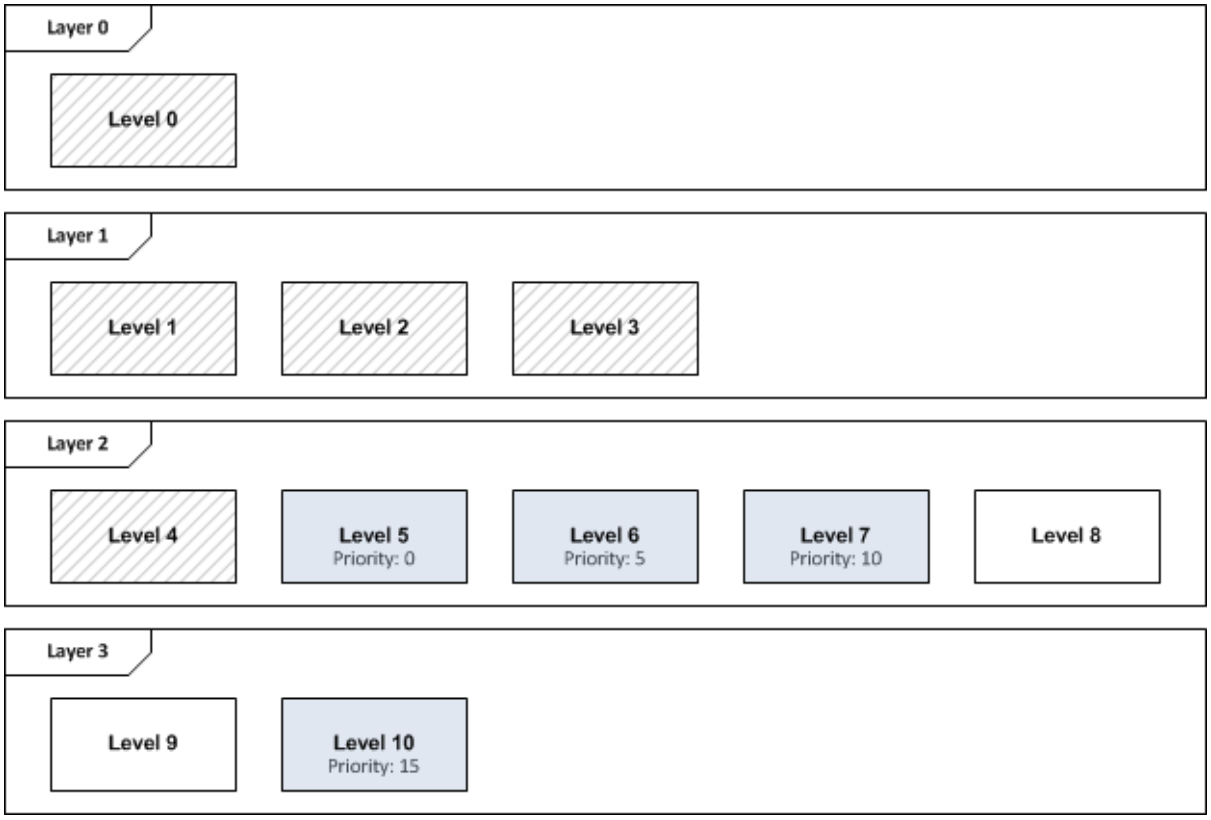

Figure 31: Transitioning layers containing mutually exclusive levels

As we discussed earlier, since level 10 doesn't belong to the same layer as Level 5, it is ignored. The priorities of rest of the levels (i.e. levels 5, 6, and 7) are compared, and the one with the highest priority (level 7) is selected for execution and the others are removed from the transitioning layer (Figure 32).

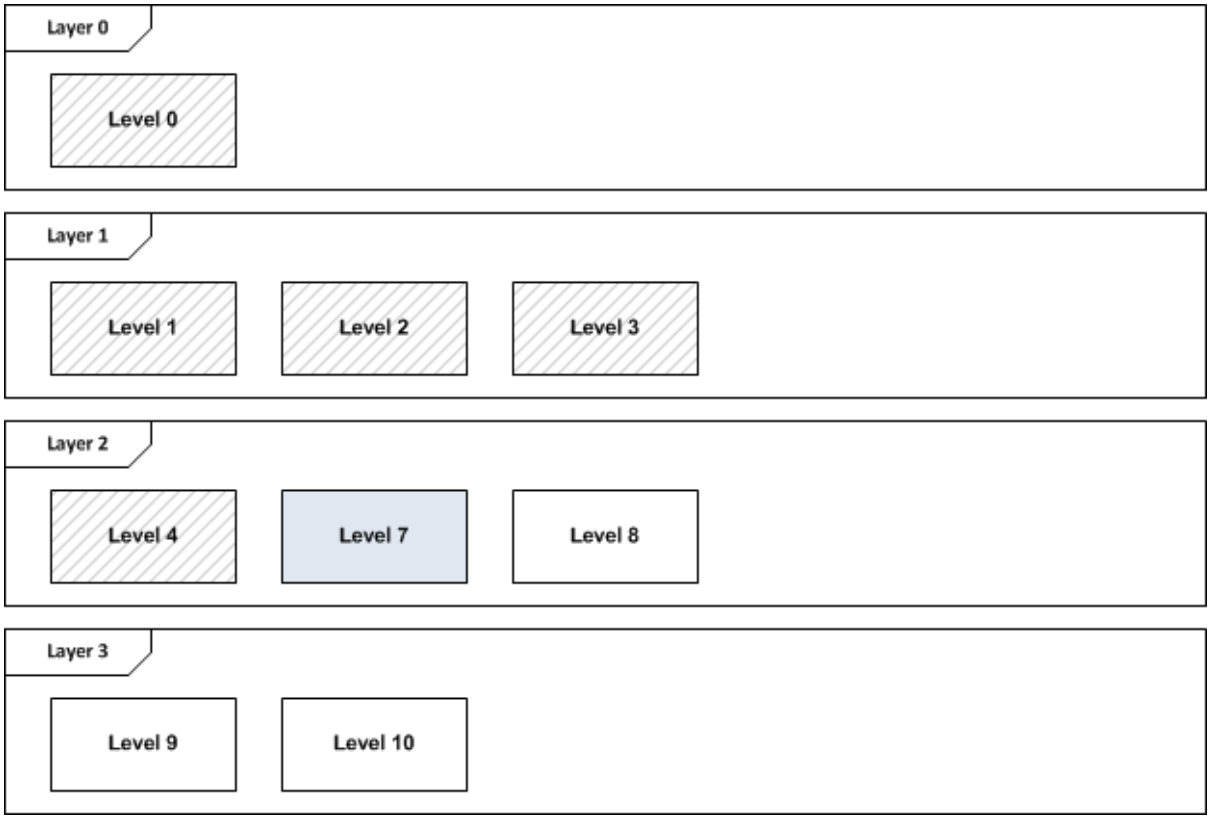

Figure 32: Transitioning layers after the mutually exclusive levels are removed

4.8. Goal Injection Manager

While the story is executed, the Goal Injection Manager (GIM) constantly monitors the current state of the active level with the help of the World Manager (WM), to identify if a goal needs to be injected to the goal list of an agent/character. The actions which can trigger a goal injection, along with any conditions, are created by the storyteller.

There are three types of goal injection rules: fixed, default, and conditional. In all of them, there is a PDDL fact that triggers the goal injection.

In the fixed type, the storyteller can specify a specific goal, which is going to be injected to a pre-defined character when a specific event occurs during the generation

of the story. In the default type, the goals to be injected are abstract and they are constructed during runtime using variables and indexes.

In the example of Figure 33, the fact name which triggers the goal injection is "*will-follow*". The complete abstract PDDL predicate is the following: "*will-follow ?who - character ?whom - character ?to – location*". The "*inject_to*" node takes as a value the index of the variable in the PDDL predicate in which the goal will be injected to. In the example, since the index is "*1*", then the goal will be injected in the character who corresponds to the "*?who*" variable.

The goal set to be injected can take two types of variables: the *"_agent_"* variable, and a set of *"_param_"* variables. The parameter nodes are specified as indexes corresponding to PDDL variables, exactly as it was explained before.

In the example, the goal which is set to be injected is *"(at _agent_ _param_)"*. The *"_agent_"* variable is going to be substituted by the agent name (explained before), and the *"_param_"* variable by the variable in the PDDL predicate corresponding to the 3rd index, i.e. the *"?to"* variable.

```
<goal_injection_rules>
  <rule>
    <fact_name>will-release-briseis</fact_name>
    <type>fixed</type>
    <inject_to>agamemnon</inject_to>
    <goals_to_inject>
        <goal>
            <goal_name>(informed-that-is-released briseis)</goal_name>
            <goal_importance>100</goal_importance>
            <precondition></precondition>
        </goal>
    </goals_to_inject>
  </rule>
  <rule>
    <fact_name>will-follow</fact_name>
    <type>default</type>
    <inject_to>1</inject_to>
    <goals_to_inject>
        <goal>
            <goal_name>(at _agent_ _param_)</goal_name>
            <goal_importance>100</goal_importance>
            <precondition></precondition>
        </goal>
    </goals_to_inject>
```

```
    <parameters>
      <parameter>3</parameter>
    </parameters>
  </rule>
  <rule>
    <fact_name>is-dead</fact_name>
    <type>conditional</type>
    <conditions>
      <condition>
        <equals index=" 1" >patroklus</equals_to>
        <inject_to>odysseus</inject_to>
        <goals_to_inject>
          <goal>
            <goal_name>(knows-important-person-is-dead                achilles
patroklus)</goal_name>
            <goal_importance>100</goal_importance>
            <precondition></precondition>
          </goal>
        </goals_to_inject>
      </condition>
      <condition>
        <equals index=" 1" >hector</equals_to>
        <inject_to>paris</inject_to>
        <goals_to_inject>
          <goal>
            <goal_name>(knows-important-person-is-dead                priam
hector)</goal_name>
            <goal_importance>80</goal_importance>
            <precondition></precondition>
          </goal>
        </goals_to_inject>
      </condition>
    </conditions>
  </rule>
</goal_injection_rules>
```

Figure 33: A set of XML Goal Injection rule nodes

The conditional type is an extension of the fixed type, where we specify a generic event that might occur during the generation of the story, and a set of conditions related to it, which can trigger different goal injections to different characters.

In the example of Figure 33, the PDDL predicate is *"is-dead ?who - character ?by - character"*. So, the system checks the *"?who"* variable against the conditions. When the *"?who"* variable is equal to *"patroklos"*, then the first condition is met. The new goal is going to be injected to *"odysseus"*. In the same manner, if the first variable is equal to *"hector"*, a goal will be injected to *"paris"*.

Every time that a new level is loaded, the WM instructs the GIM to load the new rules of the level. GIM requests the new level's information from the Level Manager, and using that asks the Parser to return all the parsed goal injection rules of this level. For each rule, the GIM communicates with the Planner to get the PDDL representation of the goals which are included to each rule, and creates the rule. This process is illustrated in Figure 34.

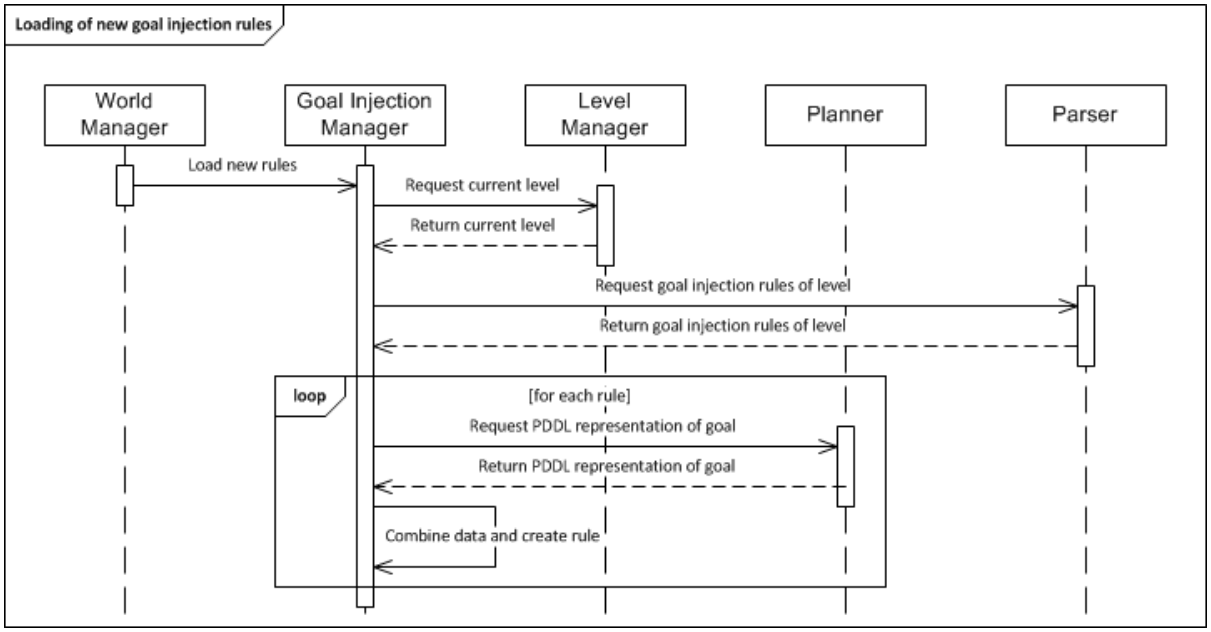

Figure 34: Loading of new goal injection rules sequence diagram

Every action has a set of positive or negative effects. Every time that an action is successfully executed during the execution/generation of the story, the WM sends its effects (which are PDDL facts) to the GIM, asking it to check if any of them match any of the goal injection rules of the currently active level.

Then, the GIM requests the PDDL name of each fact from the Planner, and checks them against the goal injection rules. If a fact matches a rule, it adds the goal injection rule to a list and at the end of this process returns the list to the WM, which deals with the injection of the goals to the appropriate characters based on the rules of each goal. This process is illustrated in **Figure 35**.

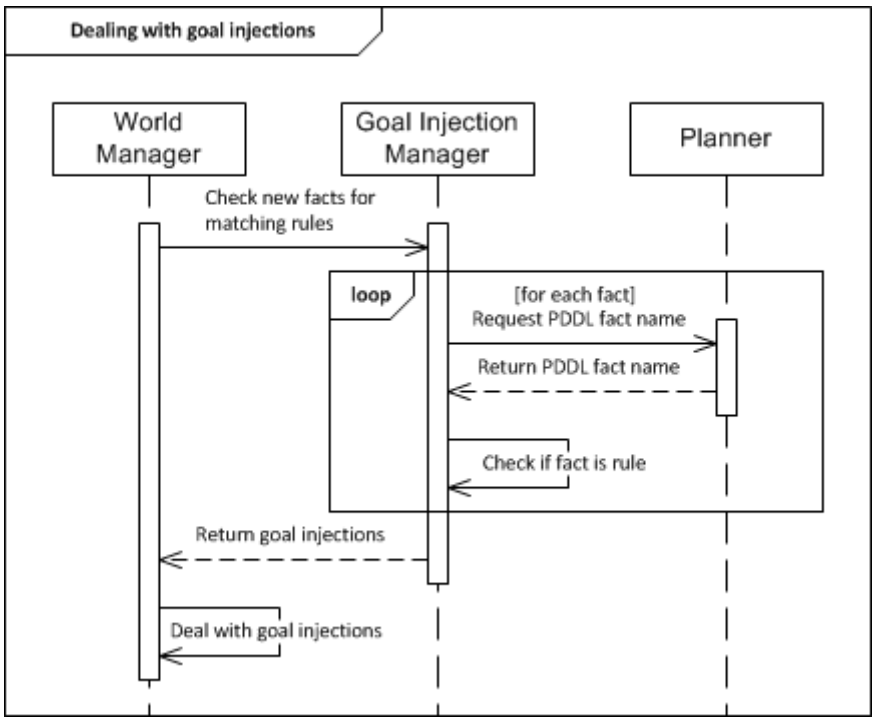

Figure 35: Sequence diagram of dealing with goal injections

4.9. FUTILE GOALS MANAGER

The idea behind the creation of "futile goals" is that sometimes (depending on the story) the characters present in a level may not have any important goals to achieve, so instead of having them staying idle, the storyteller can specify a set of futile goals which are applicable in a level, and they can be used to keep the characters busy.

A futile goal node (illustrated in Figure 36) contains a PDDL fact using a variable (i.e. ?agent) so the system can substitute it with the name of the character the futile goal is injected to, the importance value of the goal, as well as its preconditions (if there are any).

Apart from the futile goal nodes, the storyteller can also specify a set of illegal location nodes, which represent locations in which the futile goals are not applicable, meaning that even if a character is idle and located in one of these locations, she will not be assigned with a futile goal.

The Futile Goals Manager (FGM) is in direct communication with all the activated agents. If at any point any agent doesn't have any important goal to achieve, it may receive a futile goal, depending on the following conditions: if there are any futile

goals, if the agent is configured to accept futile goals, if the agent already waited for a turn doing nothing, and if the agent isn't in a location where the futile goals are forbidden. The component keeps track of the futile goals assigned to an agent, so they won't be assigned again to the same one, unless is required.

```
<futile_goals>
  <goal>
    <name>(not-thirsty ?agent)</name>
    <importance>0</importance>
    <precondition></precondition>
  </goal>
  <goal>
    <name>(not-hungry ?agent)</name>
    <importance>0</importance>
    <precondition></precondition>
  </goal>
</futile_goals>
<illegal_locations>
    <location>garden</location>
    <location>basement</location>
</illegal_locations>
```

Figure 36: A set of XML futile goal and illegal location nodes

The process of loading the new set of futile goals for each level is similar to the one of the loading of choices. Every time that a new level is loaded, the World Manager (WM) instructs the FGM to load the futile goals information for the new level (i.e. all the futile goal nodes which are relevant to the new level). The FGM makes an inquiry to the Level Manager to receive the choices information about the currently active level, and using that it asks the Parser to parse and return the set of futile goals, as well as the set of locations that the futile goals are not applicable in them, which are stored in memory. This process is illustrated in Figure 37.

The execution of agent's actions happens sequentially in turns. When it's an agent's turn to execute an action, the WM informs the agent that it's its turn, and asks for an action to execute. For the purpose of describing the process of requesting a futile goal, we will assume that the agent has neither a plan which is currently being executed nor any important goals for which it needs to find a new plan.

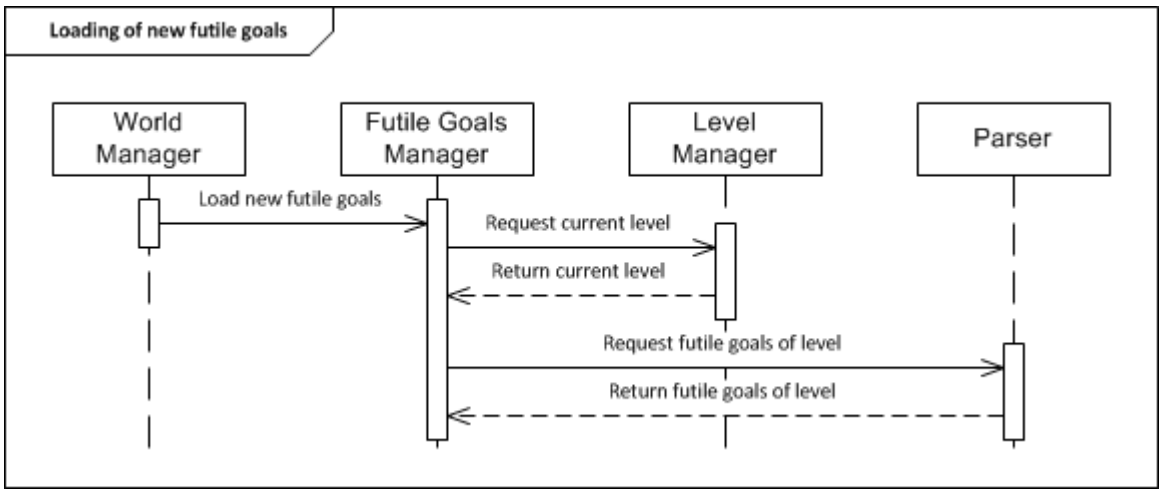

Figure 37: Loading of new futile goals sequence diagram

As soon as these conditions are met, the agent checks if has already waited for one turn. If not, the agent waits, alternatively it requests for a futile foal from the FGM, informing it of the agent's location. Considering that there are futile goals available for this level, the FGM checks if the location of the agent is in the list of the forbidden locations. If it is not, then a futile goal which was not allocated before to this agent is randomly selected.

Afterwards, it communicates with the Parser to request the PDDL representation of the goal, creates the goal for the specific agent, and returns it to the agent. As soon as the agent receives the goal, tries to generate a plan for it with the help of the Planner. If a successful plan is found then the agent returns to the WM the action which is about to execute, or just waits. This process is illustrated in Figure 38.

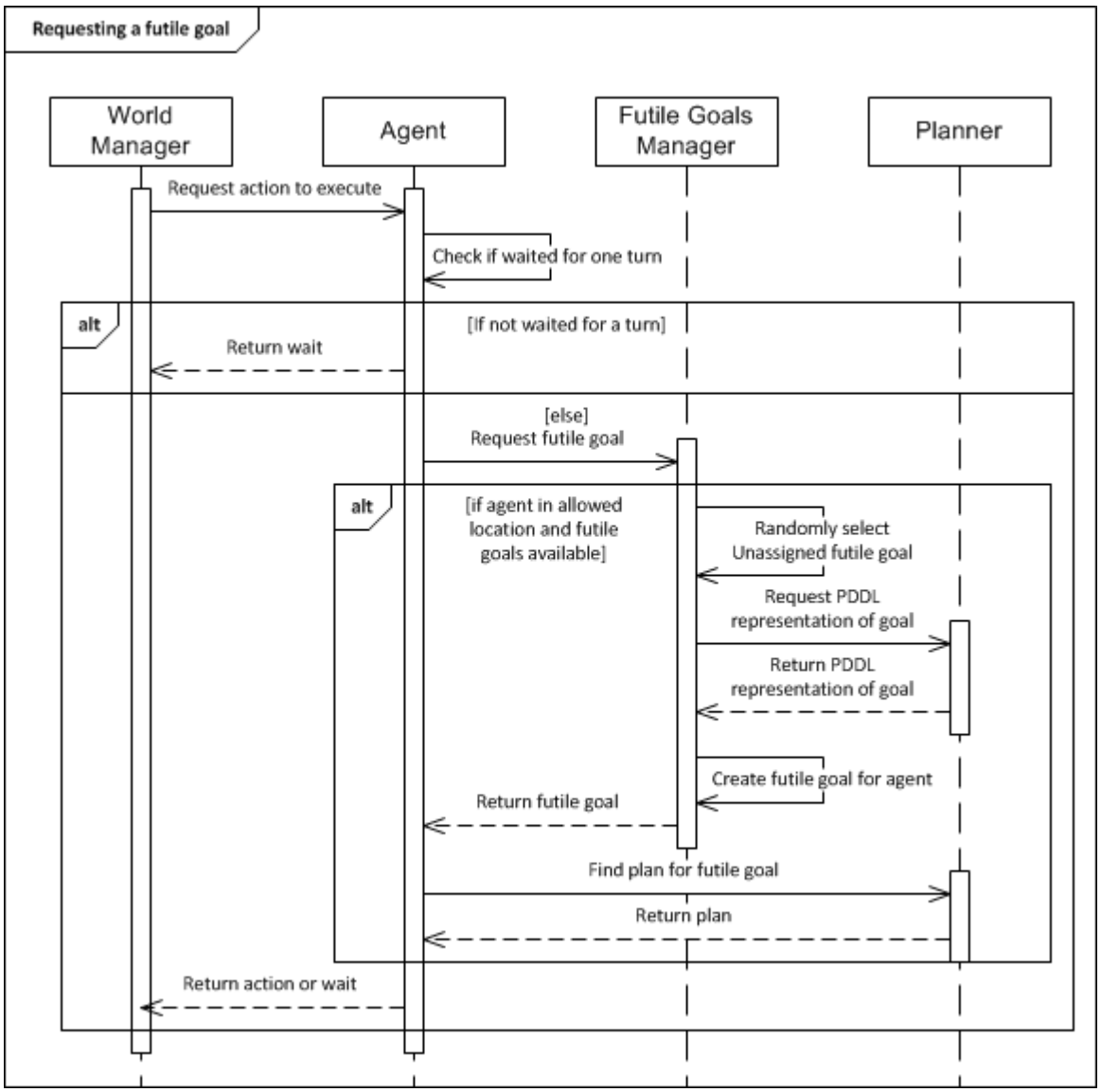

Figure 38: Sequence diagram of requesting a futile goal

4.10. ORACLE

There are certain situations during the generation and execution of a story, where a relatively random outcome needs to be calculated. For example, as we explain in section 4.16, soldiers and characters who are involved in battles have a fighting ability value. Therefore, the outcome of the battle needs to be calculated randomly, but based on that fighting ability.

The Oracle component deals with these random outcome calculations. The process is similar to the concept of a lottery, and is depicted in Figure 39. It receives as an input the number of tokens for each side (e.g. for options “A” and “B”), as well as an optional multiplier (the default value is 1).

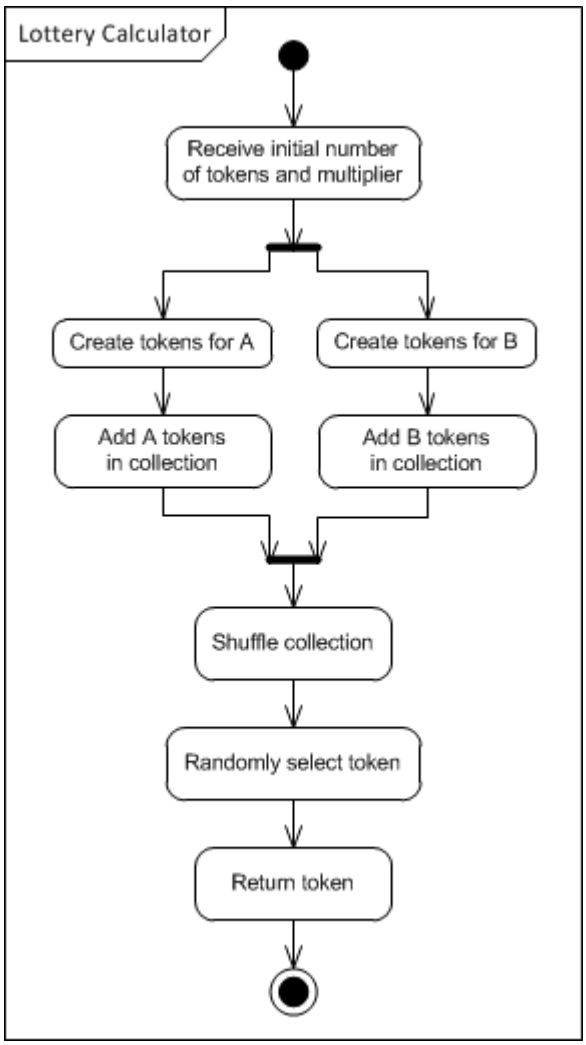

Figure 39: Calculating a random outcome activity diagram

The first step of the process is to create the tokens for each option, by multiplying the initial number of tokens with the multiplier. As an example, if option A has 20 tokens, option B 10 tokens, and the multiplier is 10, then the final number of tokens will be 200 and 100 respectively.

Afterwards, all the tokens are added in a common collection, the collection is shuffled, and a token is selected randomly, which is returned to the component which requested it.

4.11. Uncertain Actions Manager

Actions can fail or succeed based on preconditions or user interaction. But, there are some types of actions which makes sense that they should have a percentage that will succeed due to chance. We call these actions “uncertain”, and the component which deals with them the Uncertain Actions Manager (UAM).

As an example, a character may need to locate another character in a crowded area, such as a battlefield. Even if the preconditions of the action are met (e.g. both of the characters are located in the battlefield), it makes sense that the action should have a chance of succeeding or failing.

As illustrated in Figure 40, the storyteller can specify the name of such actions along with their success percentage.

```
<uncertain_actions>
  <action>
    <name>locate-in-battle</name>
    <succeed_percentage>30</succeed_percentage>
  </action>
</uncertain_actions>
```

Figure 40: A set of XML uncertain actions nodes

Every time that a new level is loaded, the World Manager (WM) instructs the UAM to load the uncertain actions of the new level. The UAM makes an inquiry to the Level Manager to receive the choices information about the currently active level, and using that it asks the Parser to parse and return the set of uncertain actions, which are stored in memory. This process is illustrated in Figure 41.

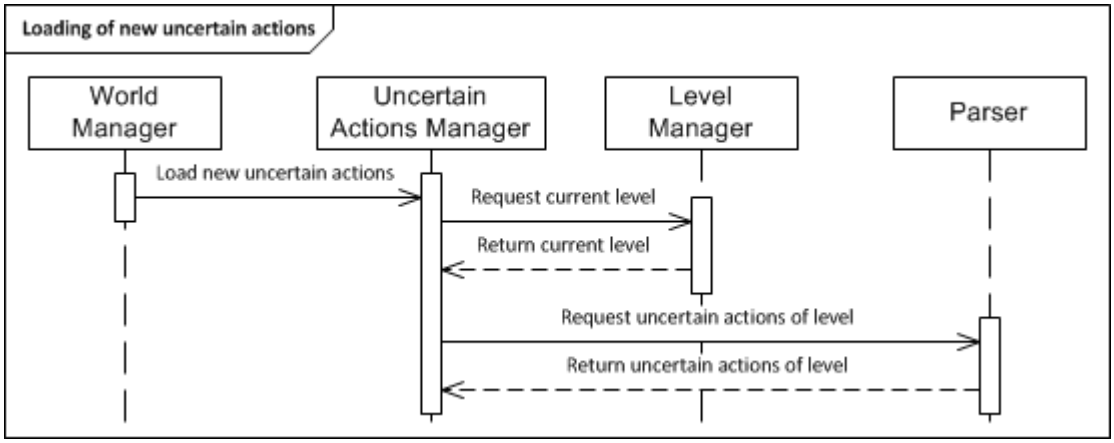

Figure 41: Loading of new uncertain actions sequence diagram

When an action is cleared for execution (i.e. the preconditions of the actions are met, the agents involved are available, etc.) WM instructs the UAM to check if an action is ok to be executed. The UAM checks if the action is in the uncertain actions list of the currently active level.

If it doesn't, the UAM sends a message back to the WM that the action is cleared for execution. If it does, then the UAM sends the percentages of success and failure to the Oracle, requesting a lottery outcome.

The Oracle calculates and returns the outcome, and based on it the UAM either informs the WM that the action is ok to be executed, or not. This process is depicted in Figure 42.

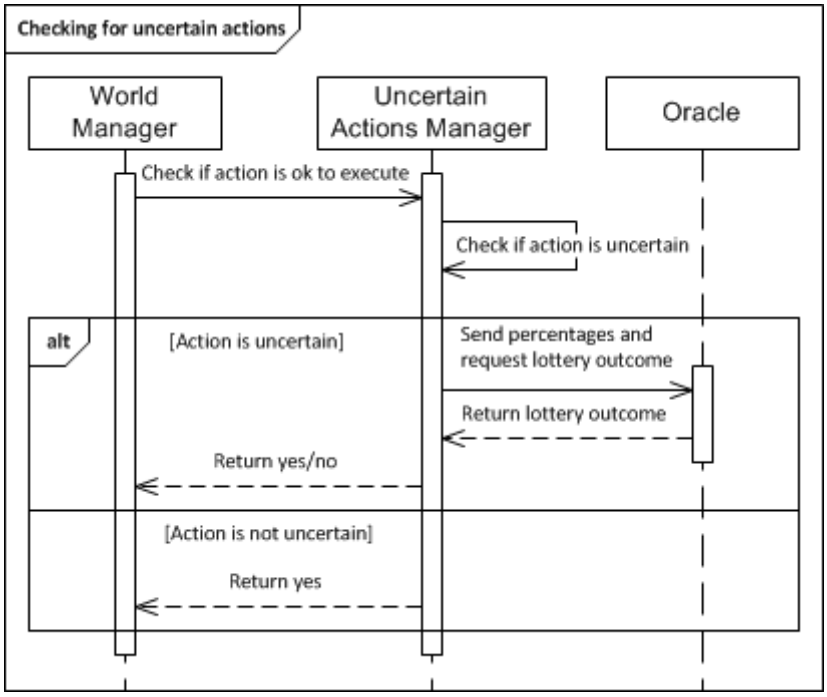

Figure 42: Sequence diagram of checking if an action is uncertain

4.12. Vantage Point Manager

As we discussed in section 2.11, DIS systems traditionally choose to apply either a first-person or a third-person perspective to present their stories to the player. In its default mode, DIEGESIS presents the generated story as a whole. At any point during the generation of the story the player is able to view any action that a character is executing, make choices related to any character, as well as view details about them (i.e. their current goals and plan). These abilities constitute a third-person perspective.

But, apart from the default mode, we want to provide the player with a first-person perspective as well; we want the player to be able to view the outcome of the story from the "eyes" of a specific character, meaning that the player will only view the story

outcome which is related to the chosen character, and will be available to interact with the story (i.e. make decisions) only when an action is related to the chosen character. We call this concept a vantage point.

The player should be able to choose between different vantage points or return to a full story view freely during run-time. Choosing a vantage point will not affect the outcome of the story, which will continue normally –yet invisible if it's unrelated to the chosen character- in the background. If an important action (i.e. choice) is about to be executed which is not related to the character whose vantage point is active, the framework will have to make the choice instead of the player.

As we mentioned in section 4.5, every time that an action is executed, the World Manager (WM) stores information about the action in the Knowledge Base (KB). For each action, the following information is being gathered and stored in KB: the characters related to the action; the time step (i.e. turn) when the action occurred, the level in which the action belongs to, and the order of the action related to other actions. We store this information so it can be used by the Vantage Point Manager (VPM) component.

Every time that a new level is loaded, the UM passes to the VPM the list of characters who are present in the level. The VPM checks if they are already in its characters list, and if they are not it stores them there.

If instructed by the player via the User Manager (UM) component, the VPM recreates the generation of a part of the story that a specific character is involved with, to allow the player to view the story from the vantage point of that character. It displays all the actions executed by the selected character, as well as any actions which were executed by other characters in which the selected character was involved, in chronological order, separated by levels (including each level's description). This process is illustrated in Figure 43.

Initially, the WM requests all the actions related to the chosen character from the KB. Afterwards, for each of the actions, it checks if it belongs to a level which information was not displayed to the player during the generation of the vantage point. If it does, then requests the level information from the Level Manager, and passes it to the

Output Generator (OG) to be displayed to the player via the UM. Then, the VPM passes the action's information to the OG so it can be displayed to the player as well.

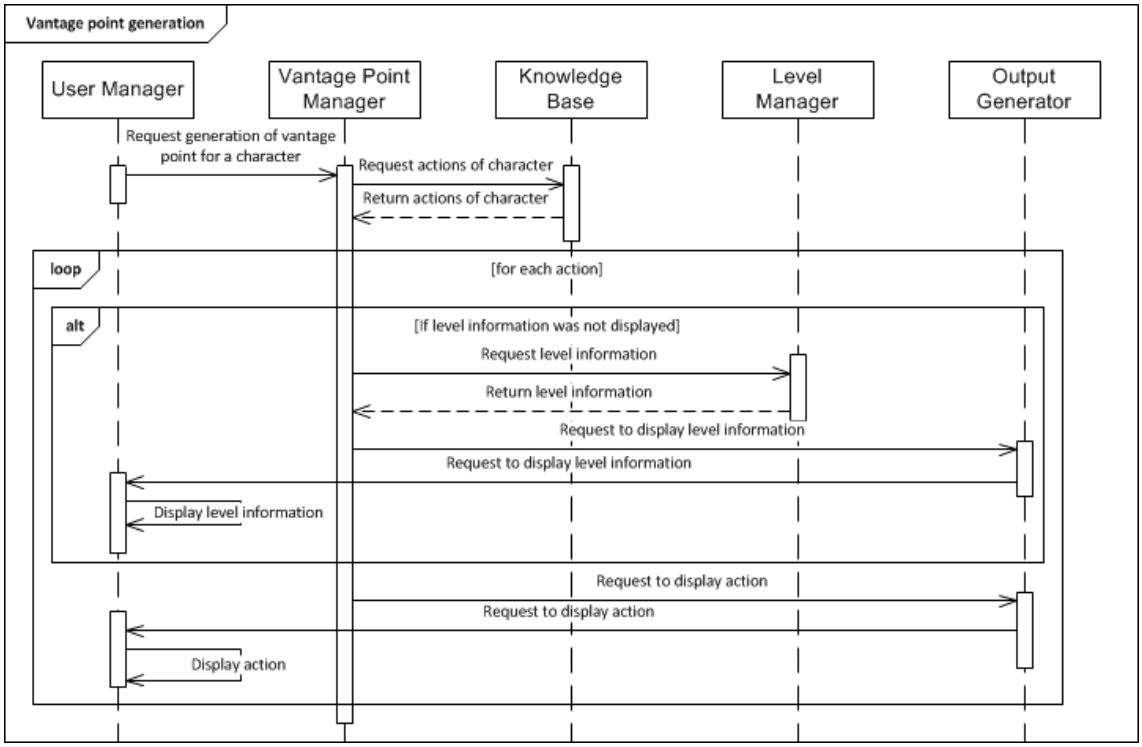

Figure 43: Vantage point generation sequence diagram

If the vantage point of a character is selected, the vantage point manager dictates which messages are going to be sent to the UM by the OG. Although the story is executed normally in the background, the player will view only the messages relevant to the selected character. This process is depicted in Figure 44.

Initially, the WM sends an action to the OG and requests to be displayed to the player and printed to the console (for debugging purposes). The OG prepares the appropriate message based on the action and prints it. Then, it checks with the VPM to check if there is an active vantage point. If it does, then sends the action to the VPM to identify if the enabled vantage point belongs to any of the characters involved in the action. If it does, then the OG sends the message to the UM to be displayed to the player. If it doesn't, then the action is not displayed to the player.

The vantage point can change at any point during the runtime, and the component is able to disable any active vantage points and restore the whole story in its default mode if requested.

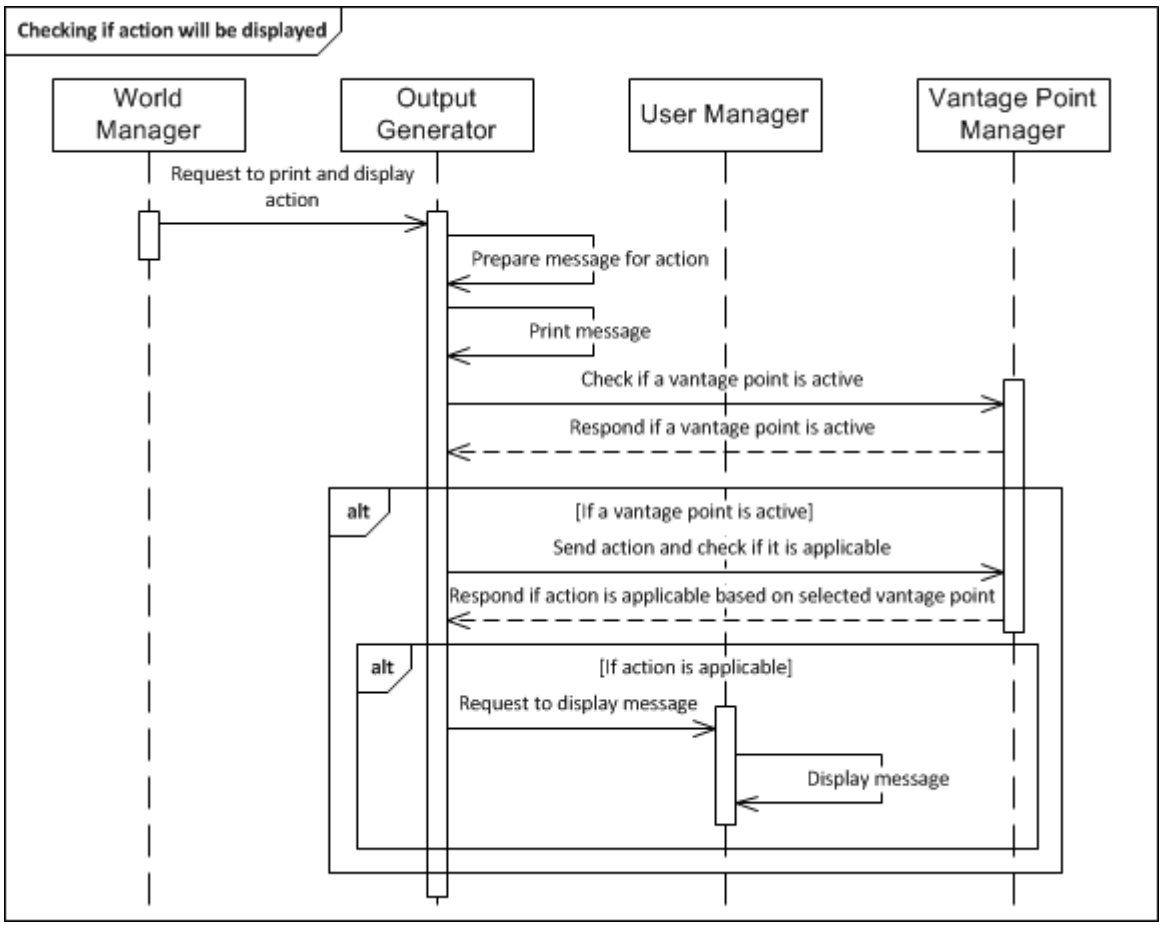

Figure 44: Checking if an action will be displayed sequence diagram

4.13. User Manager

The User Manager (UM) component is responsible of communicating with, as well as displaying the story and relevant information about it to the player.

Initially, it has been decided that based on the focus of this research, a graphical user interface (GUI) would not be developed, as being out of scope for this project. Instead, the player would receive all the relevant information and the available actions in an old-fashioned text-adventure way.

Although the first few prototypes of DIEGESIS were operating in this way, we quickly realised that it would serve our interests best and it would be easier for the player to interact with the system via a GUI.

Therefore, we decided to revisit and alter the design of the UM and include a GUI which would keep the text-based narration, but would provide an easy way for the

player to be able to view information about the story and its characters on demand, plus a few more experimental features which we later discarded. The initial mock-up of the GUI is illustrated at Figure 45.

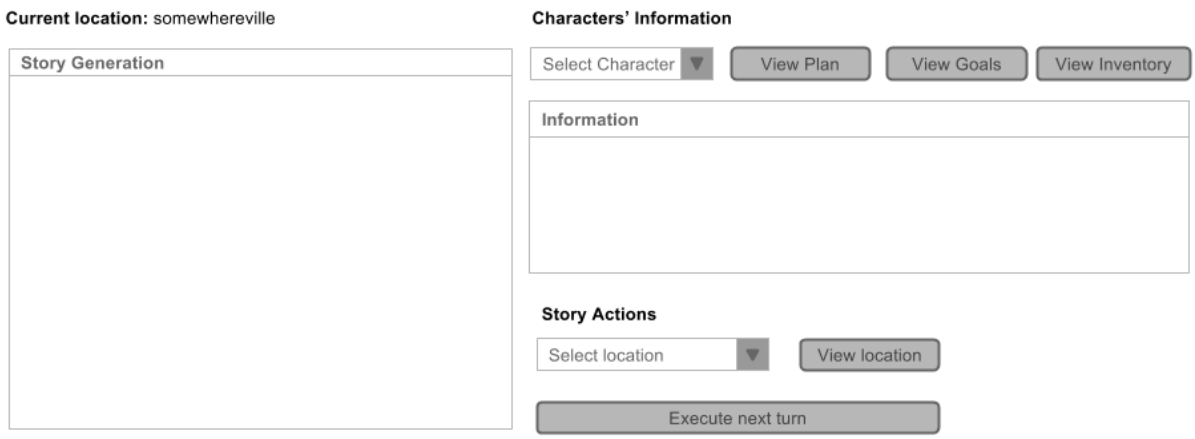

Figure 45: Initial GUI mock-up

While progressing with the design of DIEGESIS and prototyping at the same time, the design of the GUI involved. We removed features that we decided not to implement, and added others such as the ability for the player to make choices. An intermediate mock-up of the GUI is depicted in Figure 46.

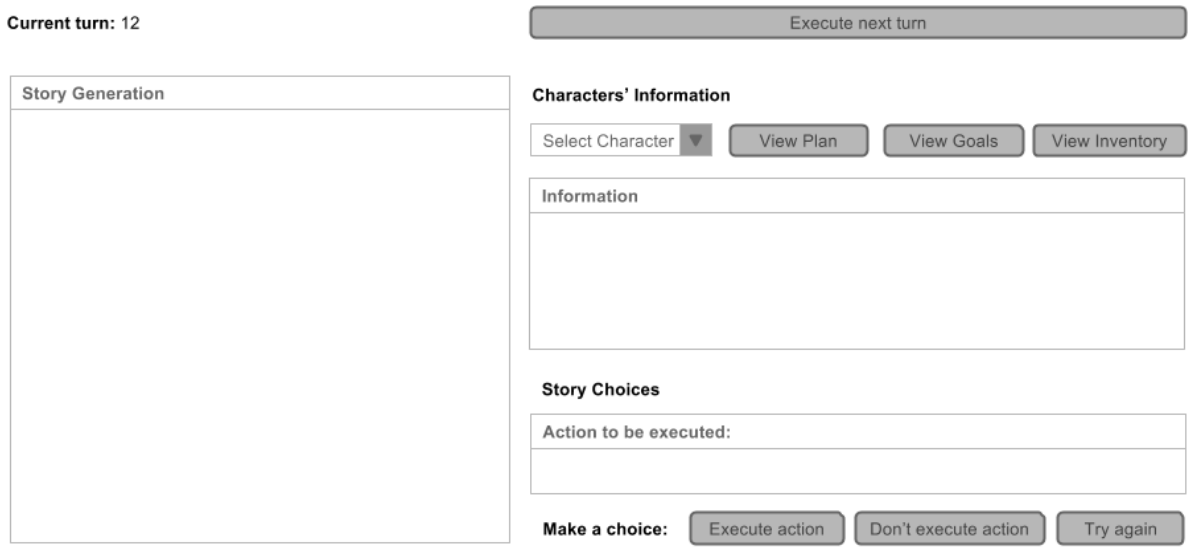

Figure 46: Intermediate GUI mock-up

Finally, as soon as we finalised the design of DIEGESIS, we included all the features we needed the player to have access to into the GUI, such as the selection of vantage points, etc. The final mock-up for the DIEGESIS' GUI is illustrated in Figure 47.

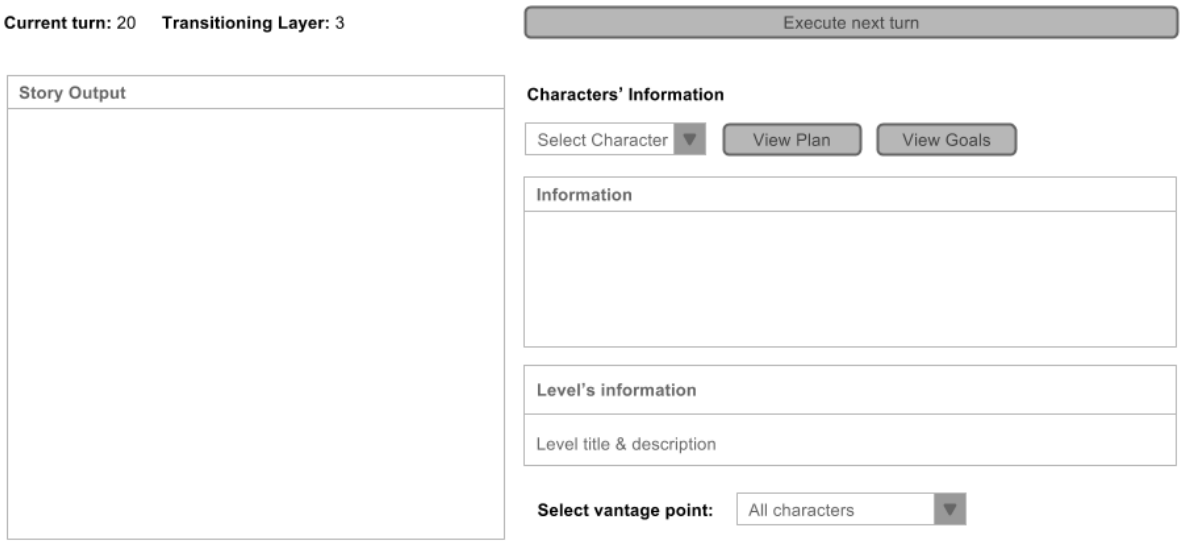

Figure 47: Final GUI mock-up

Apart from viewing the outcome of the story, the player can progress the story to the next turn, view at any point during the execution the current plan and goals for any of the activated characters in a level, view information about the currently active level, and, when is required, make choices for the outcome of the story. The player also has the ability to switch between different vantage points at any point during the generation of the story. The ability for the player to make a choice is removed from the main GUI, and added as a pop-up modal box, illustrated in Figure 48.

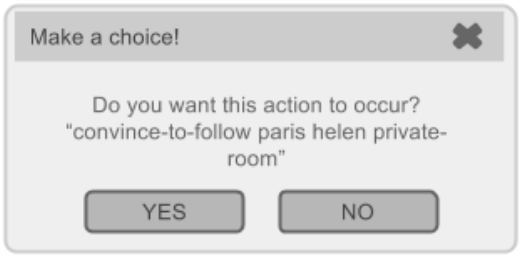

Figure 48: Modal box GUI mock-up

4.14. PLANNER

As we already discussed in section 2.2, planning algorithms are the most widely used technique in DIS systems, and this is the solution that we are using as well. DIEGESIS' planner consists of a planning and a re-planning algorithm, able to generate plans of actions based on each agent's state and context, considering both the current world state and the available resources. The planner is aware of the available time (duration) a character has for a plan when it is asked to generate one.

As we already discussed in section 2.11, our planning algorithm is based on Graphplan for solutions expansion, and backtracking heuristic search for plan extraction, enriched with constraints satisfaction and dynamic opportunistic restart when required. The Planner is the component which deals both with the planning and the re-planning processes.

When the World Manager (WM) instructs the Planner to initialise itself, the Planner requests from the Level Manager (LM) the current level's information, and then instructs the Planner to parse and return the PDDL representation of the current level, as well as any information about the duration of the level's actions. As soon as the Planner has this information, it pre-processes and initialises the planning domain that will be used to generate plans for the level. This process is illustrated in Figure 49.

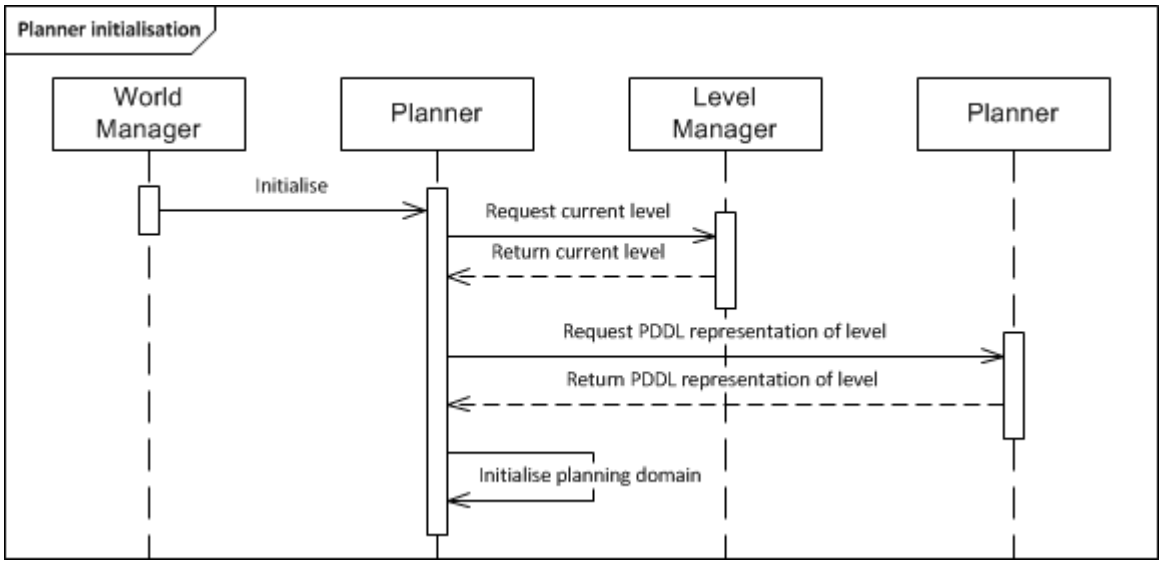

Figure 49: Planner initialisation sequence diagram

We want the actions that an agent can execute to be atomic and also we don't want to have actions that overlap each other, mostly for simplicity reasons, since it is easier to evaluate the generated plans of each character in the story and investigate how they affect each other. Therefore, we decided that instead of using PDDL's durative actions which are more complex for the storyteller to construct, to simply specify the total duration of an action (in seconds) in XML.

The storyteller can specify a set of action nodes for the actions of each level. If an action does not have a specified duration, the default duration of 0 seconds is used. An action node (illustrated in Figure 50), contains the name of a PDDL action, its duration

(in seconds), and –optionally– its type (i.e. an interruptive action, which was explained in section 4.5).

```
<actions>
  <action>
    <name>seduce</name>
    <duration>1200</duration>
  </action>
  <action>
    <name>negotiate</name>
    <duration>3600</duration>
  </action>
  <action>
    <name>kill</name>
    <duration>60</duration>
    <type>interruptive</type>
  </action>
</actions>
```

Figure 50: A set of XML action nodes

The initialisation of the planning domain in illustrated in Figure 51. Initially, the Planner gets a list of PDDL predicates and objects from the PDDL representation that was returned by the Parser. As we explained in section 3.3, predicates are expressions that describe simple or complex states of the world, and can be either true or false.

An example of a predicate is the following: “at ?who – character ?where – location”. The predicate’s name is “at”, and has two variables (words which begin with a question mark are variables): “?who” which can be only a character object, and “?where” which can only be a location object.

To make sense and to can be used in the planning process, predicates’ variables are substituted with actual objects. In our example, a substituted predicate can be the following: “at menelaos throne-room”. From now on, we will call any substituted variable, a “fact”.

As soon as the Planner has the list of predicates, for each predicate it performs any possible substitution (i.e. any possible combination of applicable objects) to create all the facts of the planning domain, and stores them in a facts list.

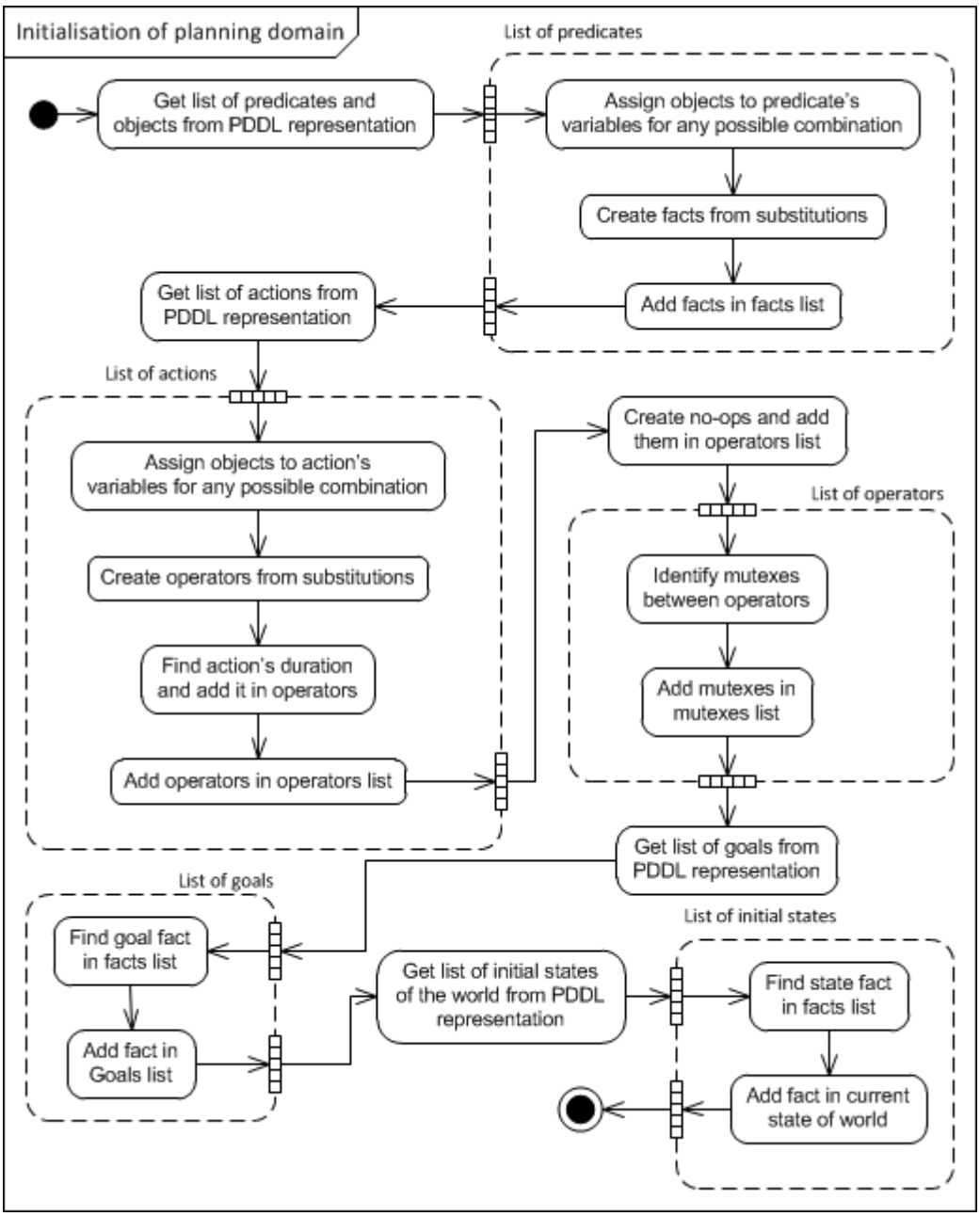

Figure 51: Activity diagram of the initialisation (pre-processing) of planning domain

Then, the Planner gets a list of all the PDDL actions from the PDDL representation. As explained in section 3.3, actions are usually made up of three parts: parameters, preconditions, and effects. Parameters are variables which define the objects which need to exist for an action to be executed, as well as their types. The preconditions are predicates related to the parameters which need to be either true or false for an action to be executed, and the effects are one or more predicates which will be turned to true or false after an action is executed successfully.

In the example of Figure 52, action "talk" has three parameters: two characters and a location; three preconditions: both of the characters need to be at the same location,

and they cannot be the same person; and one effect: the two characters will be in discussion.

```
(:action talk
    :parameters (?who - character ?to-whom - character ?where - room)
    :precondition (and
        (at ?who ?where)
        (at ?to-whom ?where)
        (not (= ?who ?to-whom))
    )
    :effect (and
        (in-discussion ?who ?to-whom)
    )
)
```

Figure 52: A PDDL action example

Similar to what happens with the predicates, the actions also need substitutions in their parameter variables to make sense. For example, one of the possible substitutions of the above action could be the following: “talk menelaos agamemnon throne-room”. Since every action performs an operation to the planning domain, within the context of the Planner we call them “operators”.

As soon as the Planner has the list of actions, it performs any possible substitution in the parameters of each action to create all the operators of the planning domain, and stores them in an operators list. It also searches the list of action’s duration, and adds the appropriate duration to each operator.

Apart from regular operators, there are also “no-op” operators. Each no-op represents a persistent action, which causes the fact present in a level of the planning graph to stay at the same state in the next level. A set of no-ops for each fact in the planning domain is created and added to the operators list.

Having created a list of all the possible operators that can be present in the planning domain, the next step for the Planner is to identify all the mutexes between them and add them to a list for future reference.

Two actions in the same level are considered mutex when their preconditions and effects are inconsistent. They can co-exists as possibilities in a level (since a level is a belief state; a state of possible states) but they cannot exist as a part of an actual solution.

By identifying all the possible mutexes before the actual generation of a planning graph and the extraction of a plan, the planning process is extremely speeded up.

After dealing with the mutexes, it is time to deal with any goals. As usual, the Planner gets a list of all the PDDL goals from the PDDL representation. Each goal is actually a substituted predicate, therefore a fact, so the Planner searches the facts list, identifies the fact which is identical with the each goal, and adds it to a goals list as well.

Finally, the same process is repeated to create the initial state of the world. Similar to goals, the initial state of the world is specified by the storyteller and consists of a set of substituted predicates, i.e. facts. The Planner creates a list of facts which represent the current state of the world. This list is passed to the WM, and is used to keep track of the state of the world at any time by setting facts to true or false. This process completes the initialisation of the planner.

To generate a plan, the Planner needs this initial pre-processed information, which can be altered during runtime. For example, when an agent requests a plan during runtime, typically it will change the initial list of goals to add its own, and will set the current state of the world as the initial state (as perceived by the Planner). The process of generating a new plan is illustrated in Figure 53.

The Planner keeps the information about the latest planning episode it worked on, and when instructed to generate a new plan, it first checks if the new set of goals is present in the list of goals it already has, and if the initial state of the world is the same. If these conditions are met, then the Planner removes the excess of goals which are not present in the new request, bypasses the expansion stage, and moves directly to the plan extraction phase. We'll get to the plan extraction phase in a bit, but to explain things properly we will first explain how the expansion process operates.

So, considering that there is a new planning problem to solve, the Planner first needs to generate the planning graph. The first level of the planning graph is the initial state of the world. To expand it, the Planner identifies all the operations (actions) which are applicable based on this current state, and "executes" them, adding their positive and negative effects (facts) to the new layer, but also keeping the facts which are already present in the initial state, copying them to the new layer via a no-op.

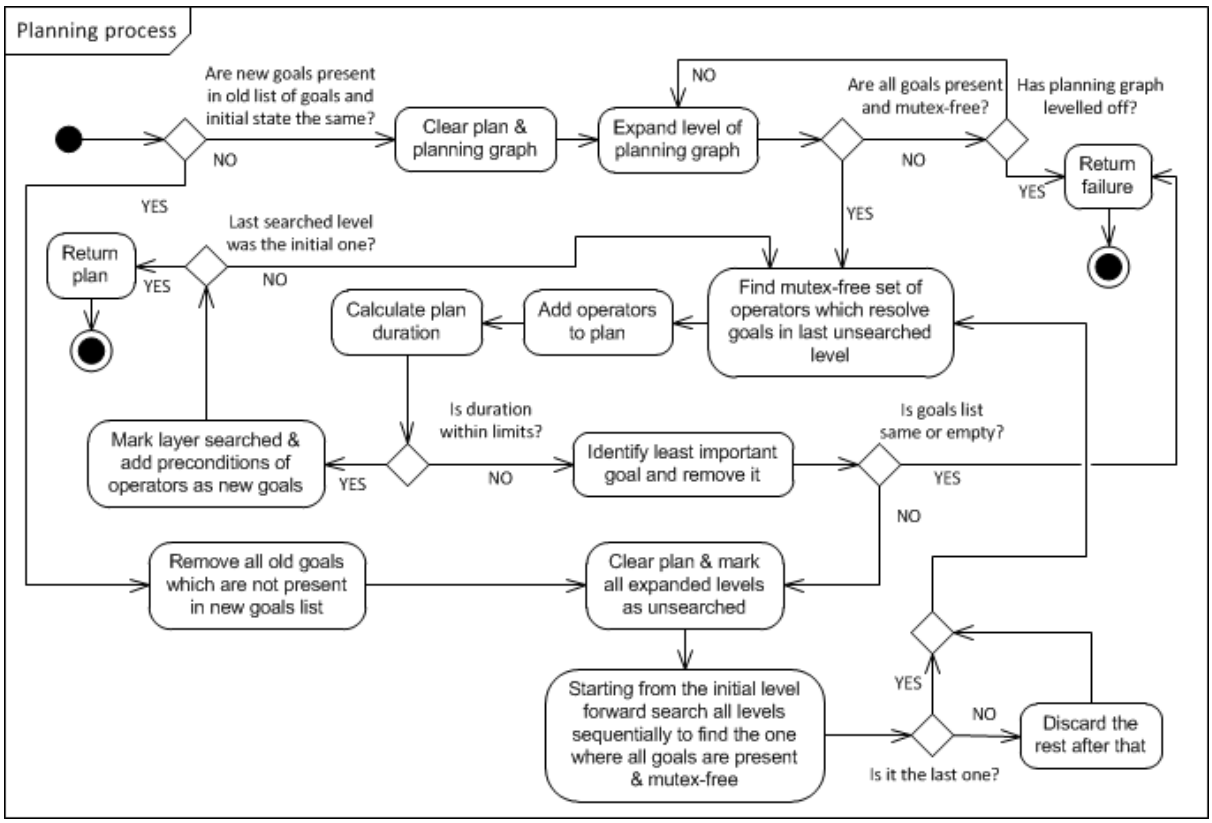

Figure 53: Activity diagram of the planning process

As an example, in Figure 54 we have a planning domain and problem (we combined them and omitted some information for simplicity reasons) with two characters ("Helen" and "Paris"), two locations ("room-1" and "room-2"), one predicate ("at" which indicates in which location a character is located) and a single action ("walk"), which allows the characters to move from one location to another. The initial state of the world is that Paris is located in room-1, and Helen in room-2.

```
(:objects
  paris - character
  helen - character
  room-1 - location
  room-2 - location
)

(:predicates
  (at ?x - character ?y - location)
)

(:action walk
  :parameters       (?x - character ?y - location ?z - location)
  :precondition (and
    (at ?x ?y)
    (not (= ?y ?z))
```

```
  )
  :effect (and
    (at ?x ?z)
    (not (at ?x ?y))
  )
)

(:init
  (at paris room-1)
  (at helen room-2)
)
```

Figure 54: A PDDL planning domain and problem

When the Planner expands the level, it creates a planning graph similar to the one depicted in Figure 55. L0 represents the first level of the planning graph, i.e. the initial state of the world. From there, as we already mentioned, to expand the level to the next, the Planner identifies all the operations which are applicable based on this current state, and "executes" them. There are two possible operators that can be executed, that is to walk Paris from room-1 to room-2, and Helen from room-1 to room-2. There are also two applicable no-op operators that preserve the facts present on the initial level, to the next one. These no-ops are represented by a dashed line. The dotted lines represent mutexes between operators or facts. As an example, although Paris can be either in room-1 or in room-2 in the expanded level 1, he cannot be at the two rooms at the same time.

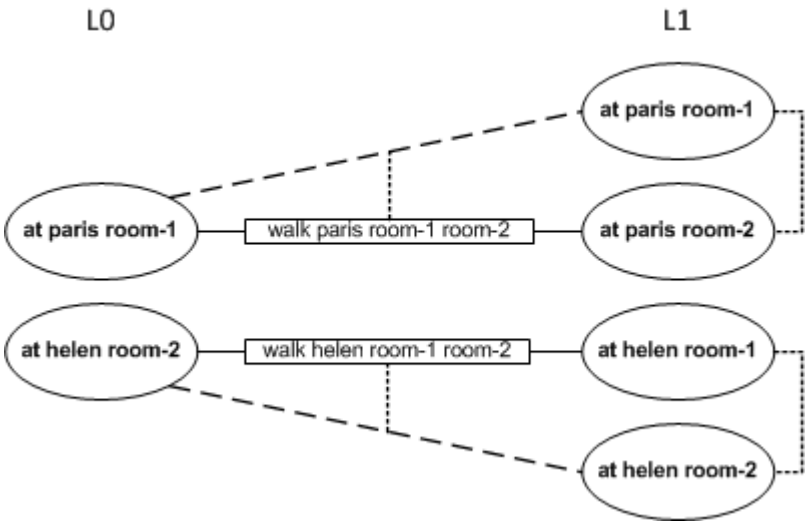

Figure 55: A simple planning graph example

As we described in section 4.5, a character's goal consists of a fact, and (optionally) from an importance (salience) value and a precondition. The precondition isn't

relevant to the Planner (is discussed in section 4.15), but the importance value of each goal is passed to the Planner along with the actual goal (i.e. the fact).

After a new expansion of the planning graph is complete and a new level is created, the Planner checks if all of the goals which it tries to generate a plan for, exist in the level and they are not mutex (i.e. they don't have any constraints) between them. For example, if the goal of the expansion example we just described was "at paris room-2", then it would be located in the first level.

If the goals are not present in a level, or they are not free of mutexes, then the planner checks if the planning graph levelled off before it expands another layer. If two consecutive layers are identical we consider that the planning graph has levelled off, meaning that we reached a point where the planning graph cannot be expanded any further (all the subsequent levels will be identical to each other). If that happens, then the process ends there and the Planner returns a failure message.

The above process is being performed until either the planning graph levels off, or a level is found containing all mutex-free goals. As soon as this happens, the Planner moves on to the plan extraction phase.

The extraction phase includes a backward search starting from the last generated level which contains all the goals. There, the Planner identifies a set of operators (actions) which resolve the goals and are mutex-free between them, and adds them to the generated plan as the last set of plan actions. A plan consists of one or more sets of actions, which can contain one or more action(s).

Then, the Planner calculates the duration of the whole plan using the action duration values of each action specified by the storyteller. If the total duration of the plan is within the specified limits, then the planning graph's level is marked as searched, and all the preconditions of the operators that solve the problem for this level (and were added to the plan) are extracted and added as the new set of goals.

If all the levels are marked as searched (i.e. the last searched level was the initial state of the world), then a valid plan has been found, and the Planner returns it. If not, then the process continues for the next level of the planning graph (as a reminder, we move backwards).

If the total plan's duration exceeds the available time, the planner chooses the least important goal (based on the "salience" value of the goals), and removes it from the list of goals. If (after the removal) the goal set is still the same (i.e. there was not a goal which wasn't important), or there are no other goals left, the process stops and the Planner returns a failure message.

If there are still goals to search for, the Planner clears the plan and marks all of the expanded levels as unexecuted. Then, starting from the initial level it performs a forward search in all levels sequentially to find the one in which all the new set of goals is present and all the goals are mutex-free between them. As soon as it found, if it is not the last one in the planning graph, then the levels after it are marked as ignored, and it becomes the last. Then, the process continues normally by searching in the last level for a solution.

As we already mentioned in section 3.5, we want each agent to be able to plan and operate as an individual, creating an autonomous plan considering its own needs. The planning approach we just explained suffers from the following: it best suited for a centralised planning, where we want to perform a decentralised planning.

Elaborating on the centralised planning issue, we will discuss a new expansion example (its details can be found in Figure 56), which is an expanded version of the expansion example we discussed earlier in this section.

```
(:objects
  paris - character
  helen - character
  room-1 - location
  room-2 - location
  item - item
)

(:predicates
  (at ?x - character ?y - location)
  (has ?x - character ?y - item)
)

(:action walk
  :parameters       (?x - character ?y - location ?z - location)
  :precondition (and
    (at ?x ?y)
    (not (= ?y ?z))
```

```
  )
  :effect (and
    (at ?x ?z)
    (not (at ?x ?y))
  )
)

(:action give
  :parameters (?i - item ?x - character ?y - character ?z - location)
  :precondition (and
    (has ?x ?i)
     (at ?x ?z)
     (at ?y ?z)
     (not (= ?x ?y))
  )
  :effect (and
     (has ?y ?i)
     (not (has ?x ?i))
  )
)

(:action take
  :parameters (?i - item ?x - character ?y - character ?z - location)
  :precondition   (and
     (has ?y ?i)
     (at ?x ?z)
     (at ?y ?z)
     (not (= ?x ?y))
  )
  :effect (and
     (has ?x ?i)
     (not (has ?y ?i))
  )
)

(:init
  (at paris room-1)
  (at helen room-2)
  (has paris item)
)
```

Figure 56: An expanded PDDL planning domain and problem

The differences with the previous example are that apart from the two characters and the two locations, it contains an item, a new predicate (“has” which indicates which character is in possession of an item), and two new actions, “give” which a character can use to give an item to another character willingly, and “take” which a character

can use to take an item from another one without asking permission. The initial state of the world is that Paris is located in room-1, Helen in room-2, and Paris has the item.

Considering that the Planner is instructed by the agent who represent Helen to find a valid plan for the goal “has helen item”, it will expand the planning graph up to the third level (L2). The expanded planning graph up to that point is illustrated in Figure 57. In level L2, there are four (4) actions which will accomplish the goal: Paris can give the item to Helen either in room-1 or in room-2 (one action for each room), and Helen can take the item from Paris, again either in room-1 or in room-2. Therefore, a possible valid plan is that Paris will walk from room-1 to room-2 where Helen is originally located, and there he will give her the item.

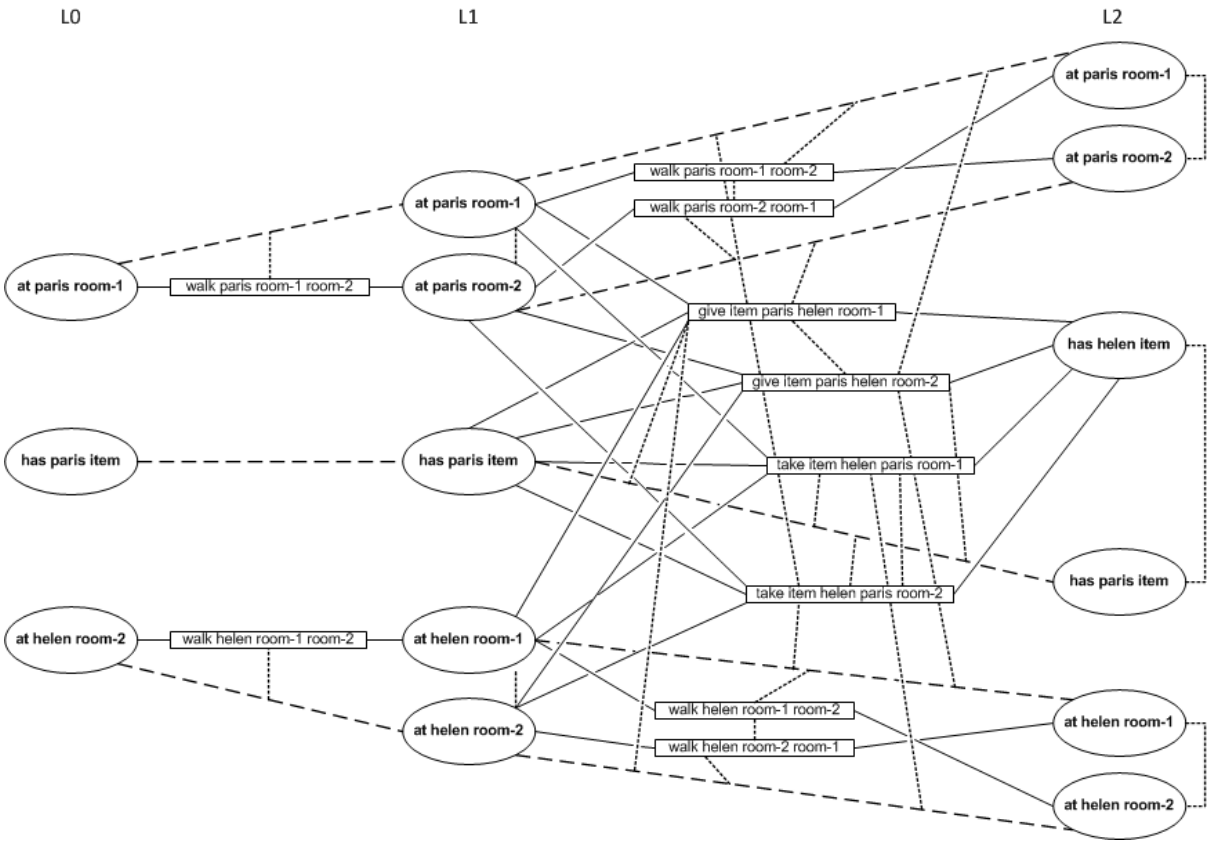

Figure 57: The expanded planning graph

But, in our context, that plan doesn’t make sense. Since the goal belongs to Helen, she is the one who needs to take action and accomplish that goal. Therefore, she should ask the framework to execute actions that concern her, and not request from the framework to force another agent to execute an action that he didn’t plan for. The solution to that problem is to specify to the Planner that it needs to generate a plan which includes actions that can only be executed by Helen.

If we do that only for the extraction phase though, for example to make the Planner identify which actions are not executed by Helen and ignore them while searching for a valid plan, it still doesn't make sense to expand the planning graph having these actions in the first place. Apart that they are irrelevant for the plan the Planner needs to generate, they can also potentially have a huge performance impact since they the planning graph will grow a lot.

For example, if in the planning domain we just discussed we had one hundred (100) characters instead of two (2) and the same two (2) locations, in the first level of the planning graph we would have ninety nine (99) facts which should not be there, since the ninety nine (99) "walk" actions which have them as effect should not be executed since they are irrelevant of the agent who actually plans.

To solve all of these issues, the Planner is aware of the name of the agent who instructed it to generate a plan. Furthermore, for every action which involves more than one character, we specify which character is the main one, i.e. the one who initiates the action.

These steps have the effect that the Planner now expands the planning graph only for the main character, leaving the rest in their initial state (i.e. the state that the character knows they are when initiates the plan generation). The new planning graph is illustrated in Figure 58.

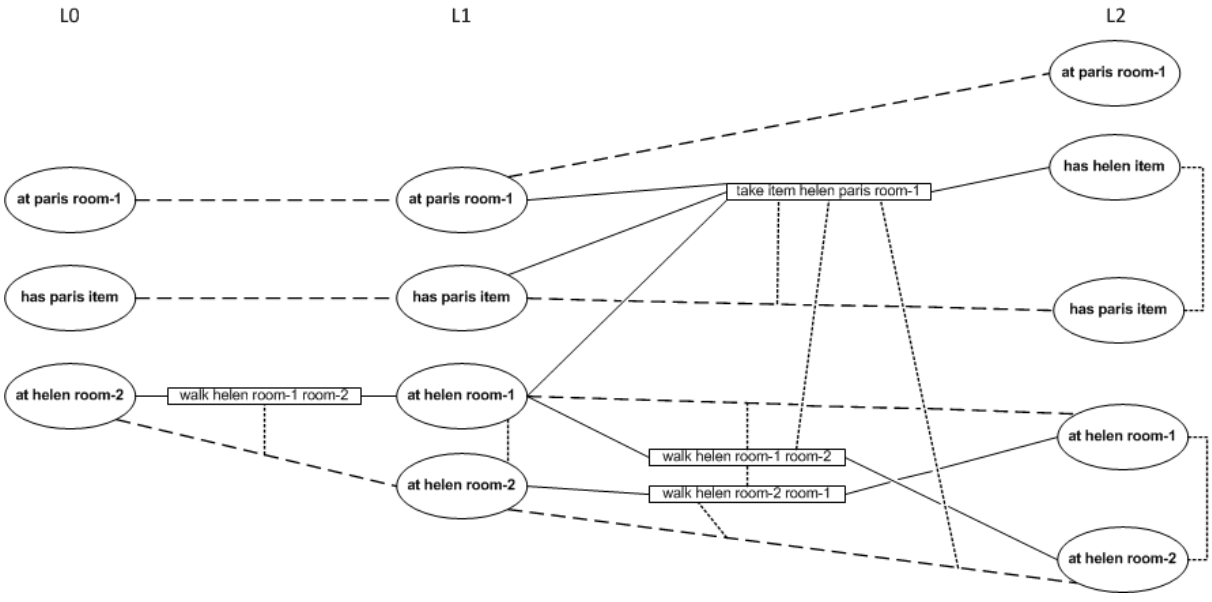

Figure 58: The new simplified planning graph

If we compare the new planning graph of Figure 58 with the old planning graph of Figure 57, we can see that even in that quite small and simple example, the complexity of the planning graph is highly reduced, which will cause the Planner to find a valid plan which makes sense easier.

Of course, by planning only for one agent without any idea of how the rest are going to operate in the future (e.g. if they will stay in the same location, etc.) can often cause a plan to fail while being executed.

Plan failures do not apply only to DIEGESIS though. In any type of planning systems, even if a plan is thoroughly tested and well-constructed there are unpredicted situations that might cause it to fail. Since (as we already mentioned) each agent operates as an individual generating its own plans based on its own goals, it is common for an agent's plan to interfere with another agent's plan, causing the latter to fail. The interference can be something simple such as the availability of a resource, or even something more complicated, like the death of a character which can alter significantly an agent's goals, therefore its plans. Another option that can trigger re-planning is user interference.

When a plan fails, it is necessary for a new plan to be constructed, and this can typically happen in two ways: either by planning from scratch, discarding any previously generated plans, or by revising a previously generated plan. We believe that the re-planning methods that reuse information from previously generated plans are much faster than the approach of solving each planning task from scratch. We validated this assumption performing an evaluation (Goudoulakis et al., 2013) (discussed in section 6.5.4), where we evaluated our re-planning method against planning from scratch.

Our proposal for re-planning in DIS interleaves plan generation and plan execution to be able to re-plan as soon as an unexpected change happen in the environment, while making minimal disruption to the original plan.

The re-planning process is illustrated in Figure 59. When instructed to do so, the Planner retrieves all the action sets of the plan which needs to revise. Then, it selects the action set in which at least an action failed. For each of the actions, it checks if the

action is failed or is still pending. For any of these conditions, it retrieves the action's effects and adds them as goals for the new plan. Then, it checks if there are other action sets after the one it just checked. If there are, then it selects the first of these sets, and for each action (since they are all pending), it adds their effects as goals as well.

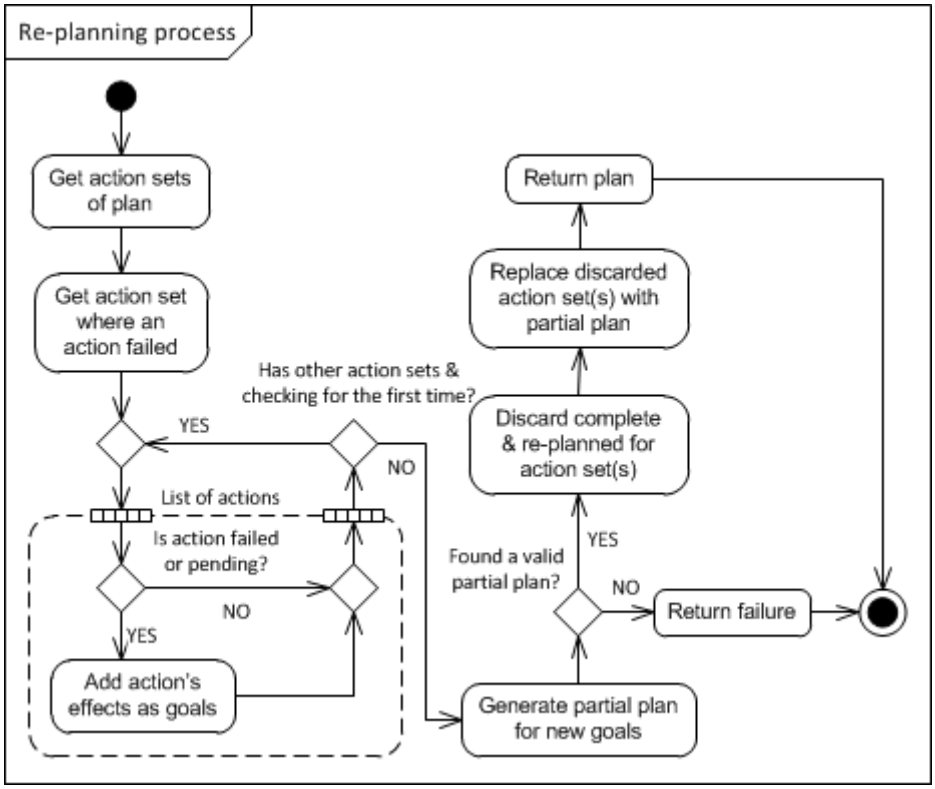

Figure 59: Activity diagram of the re-planning process

As soon as the new goals list is prepared, the Planner generates a new partial plan for these goals using the planning process we described earlier in this section. If a new plan is not found, then it returns failure and terminates the process. If a new valid plan is found, then it moves to the merging phase, where it discards the action sets which are already complete, as well as the action sets for which it generated the new plan, and replaces them with the new partial plan, forming a complete valid plan.

Figure 60 illustrates an abstract example of how our re-planning solution is operating. There is an initial plan, consisting of n sequential sets of actions. While an agent is executing the third action set (A2), one if its actions fails.

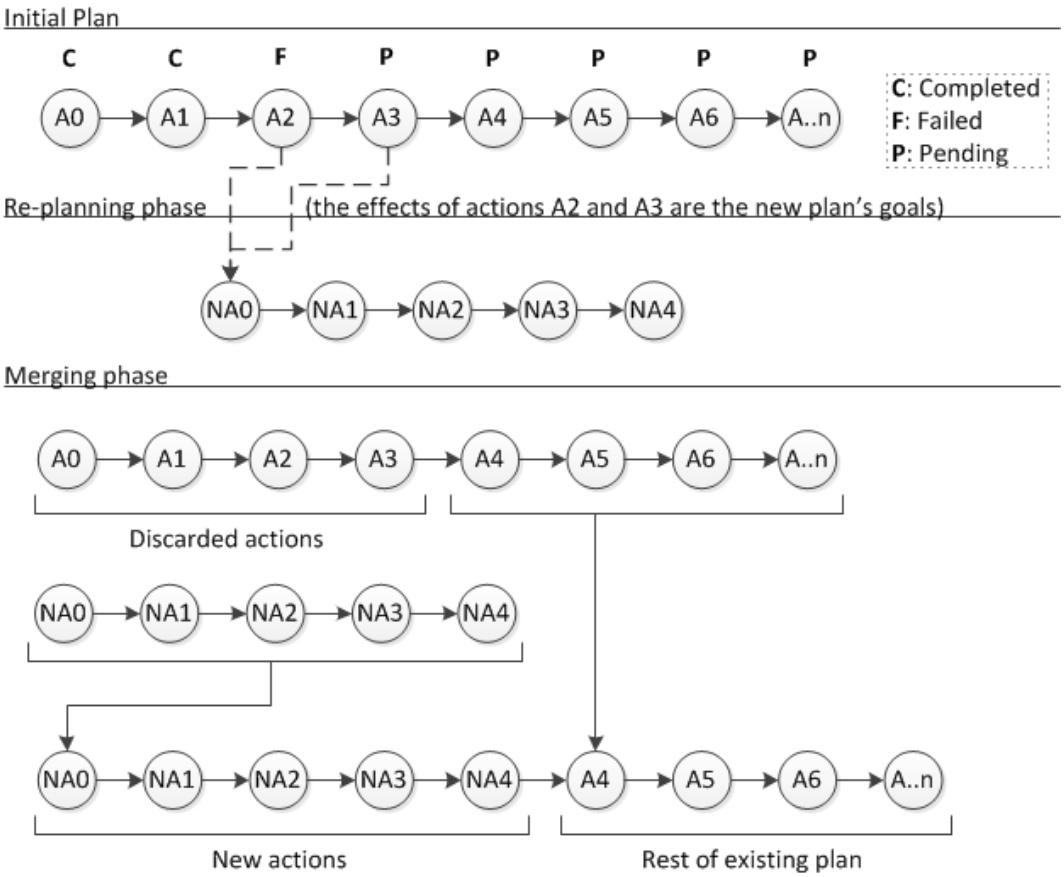

Figure 60: An abstract re-planning example

Our re-planning algorithm identifies the actions which are pending or failed in the A2 set, and adds their effects as goals for the new partial plan which will be generated. It also identifies that there are other action sets after the failed one, so adds the effects of the actions of the next set (i.e. A3) as goals as well. Based on these goals, a new partial plan is being generated using our planning algorithm.

As soon as a valid plan is generated, we proceed to the merging phase, where the first 4 action sets (A0 – A3) are discarded since they consist of completed action sets (A0 – A1) and action sets for which the algorithm has just generated a new partial plan to replace them (A2 – A3).

The new plan consists from the action sets of the partial plan (NA0 – NA4) which are placed at the beginning of the plan, and the rest of the sets of the existing plan (A4 – A..n).

4.15. AGENT

As we already discussed, DIEGESIS is designed as a multi-agent solution. Each agent represents a character in the game world and uses an instance of the Planner component to be able to generate plans of actions, and regenerate them if needed. Each agent is designed to operate as an individual, creating an autonomous plan considering its own needs.

As discussed in section 2.6, there are several intelligent agents' architectures that are used in multi-agent systems. DIEGESIS' agent architecture, follows a hybrid approach; it includes elements of reactive agents (the agent receives input, processes it, and produces an output), elements of deliberative agents (the agent keeps an internal view of its environment), and elements of BDI agents (Beliefs – the agent's view of the world, Desires – the agent's goals, Intentions – the agent's plans).

The initialisation process of an agent was discussed in section 4.5; when initialised, the agent is fed with relevant information: the character's (who the agent represent) name, the available time to complete any goals (if applicable), its fighting ability and the alliance it belongs to (if applicable), if the agent can request futile goals when has nothing to do in the level, the location it is currently located into, the current state of the world, and a set of initial goals for the specific level.

After an agent is initialised, the Agents' Manager (AM), a sub-component of the World Manager (WM), instructs the agent to generate an initial plan based on the goal set that the agent has. To do that, and only if the agent has at least one goal to generate a plan for, the agent sends to the Planner all the information it's going to need to generate a plan (i.e. the current state of the world, the available time of the agent, and the set of agent's goals), and requests for a new plan generation. The Planner tries to generate a plan, and either returns the plan or a message of failure (as discussed in section 4.14). This process is illustrated in Figure 61.

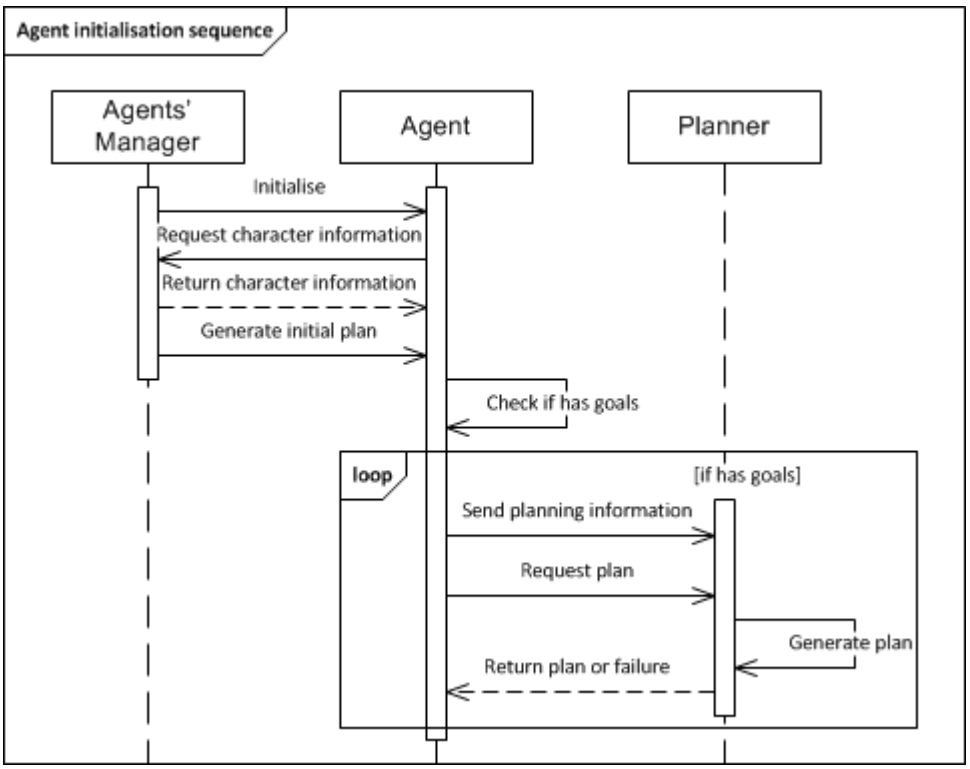

Figure 61: Sequence diagram of the initialisation of agents

For our evaluation needs (and specifically for the evaluation documented in section 6.5.5), we decided to model the agent to be able to request a plan in two different ways: either by requesting a plan for all of the goals in its goal list, or by requesting a plan only for the most important of its goals (the concept of goal importance was discussed in section 4.14), and when that goal is complete to move to the next goal, etc. The storyteller can choose for the agent to operate in any of these two ways.

The way in which the agent is constructing the list of goals to generate a plan for is illustrated in Figure 62. The agent holds two lists of goals: a list containing all of the goals which are still pending, and a list (which is basically a sub-set of the aforementioned list) which contains the set goals which will be sent to the Planner to generate a plan for them.

Initially, the agent clears the list of goals to generate a plan for. Then, considering that there are goals which are pending, it iterates through the list of pending goals, and for each one of them checks if the goal is marked as complete during the execution of a previously generated plan. If it is, then is removed from the original pending goals list. If not, then the agent checks if the goal has any preconditions. If it does not, then the goal is added to the goals to search for list. In case the goal has preconditions, the

agent iterates through them and checks their status against the current state of the world. If all of the preconditions are met, then the goal is added to the list. If not, then the goal is ignored (but is left in the original pending goals list).

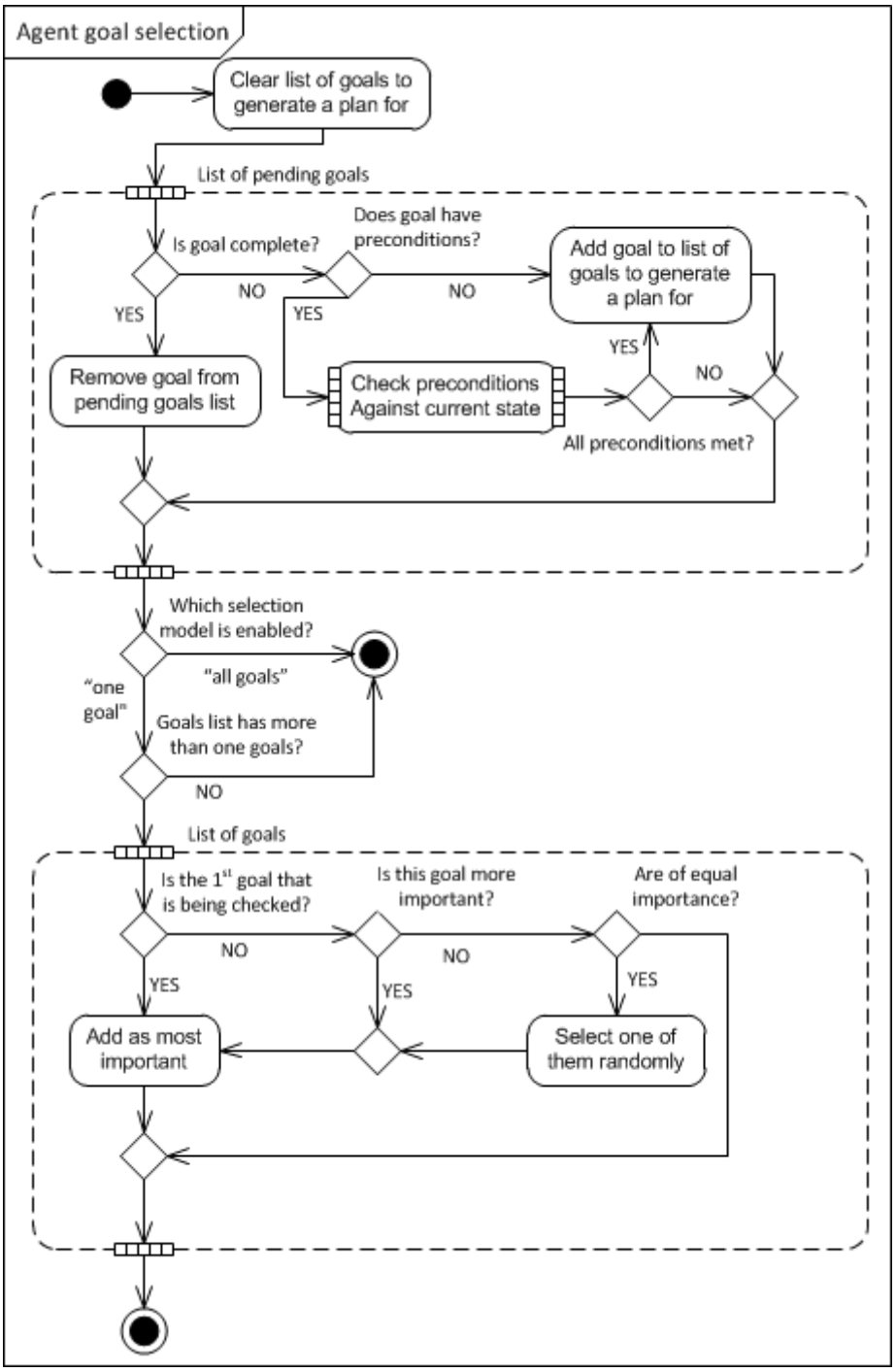

Figure 62: Activity diagram of the agent goal selection process

As soon as the new goals list is populated, the agent checks which of the two modes we discussed earlier (i.e. planning for all the goals at the same time versus planning for one goal at a time) is enabled. If the mode of searching for all of the goals is enabled, then the process ends. Alternatively, the agent needs to identify the most important of

these goals, and remove the rest. To accomplish that, the agent checks the volume of goals in the list.

If the list is empty or has only one goal, then the process ends. If there are more goals in the list, then for each of the goals the agent first checks if the goal is the first one that is being checked. If it is, then the goal is kept as the most important goal. If it is not, then the importance value of this goal is checked against the goal which is currently the most important one. If it is higher, then this goal replaces the other as the most important. If it is lower, then the goal is ignored. In case they are the same, then the agent selects one of them randomly and uses that as the most important.

As soon as this process ends, the agent checks if there is at least one goal to generate a plan for. If there is, then it sends all the appropriate information to the Planner, and requests a plan. If the Planner returned a plan, the agent stores the plan to be executed later, and ends the process. If not, then the stored plan remains empty. This process is illustrated in Figure 63.

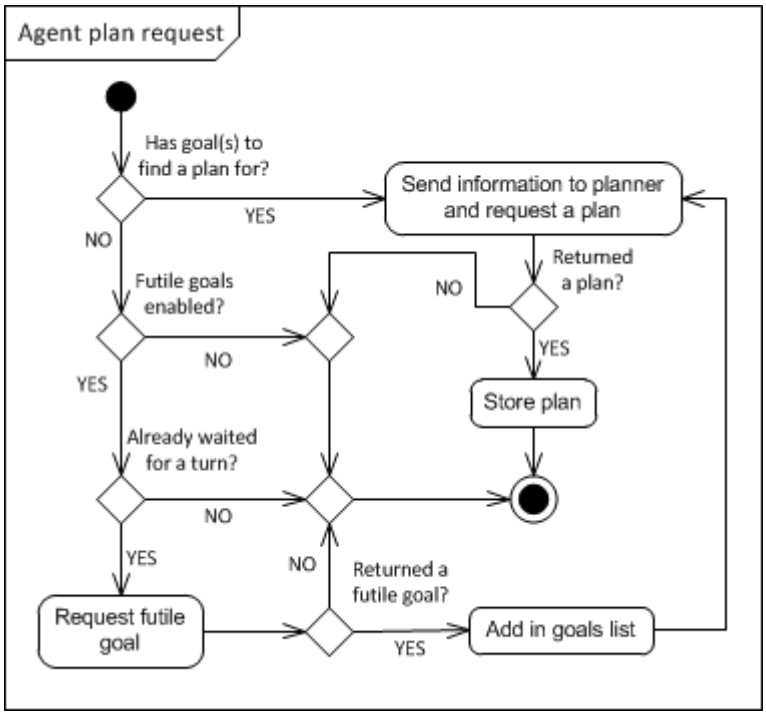

Figure 63: Activity diagram of agent plan request process

In case there are no goals to find a plan for in the list, then the agent checks its configuration to identify if it can accept a futile goal, and if already waited for one turn. If both of these conditions are true, then requests a futile goal from the Futile Goals Manager (FGM); the details of the process are discussed in section 4.9. If the FGM

returned a futile goal, then this goal is added in the goals to search for list, and the agent sends the goal along with the rest of the required information to the Planner to generate a plan.

When an agent has a plan, is able to keep track of which parts of the plan already executed and if the plan is complete, as well as which is the next action set that needs to execute when requested to do so by the WM (as we discussed in detail in section 4.5). If during the execution of a plan is informed by the WM that an action set was failed and instructed to re-plan, then the agent communicates with the Planner sending all the relevant information requesting the Planner to re-plan based on the provided information and current state of the world.

The agent is also aware if the character that represents is still alive as well as the location in which is located at any point during the execution of a level. When a level ends, every agent which was enabled during the level is deactivated.

4.16. Battle Manager

There are cases in the story that we are using to evaluate DIEGESIS, in which we need battles to occur, which add an added form of complexity to and have the potential to affect greatly the generated story. Therefore, it makes sense to create a component which will handle all these battles between either armies of Non-Player Characters (NPCs) or story characters/agents, even both. The Battle Manager (BM) component is responsible of dealing with all the battles in the system.

First of all, to specify if a battle occurs in a level, the storyteller needs to mark it as a battle level in the XML file containing the main information about the level which we discussed in section 4.4. An example of the additional semantics is displayed in Figure 64.

In each battle level, there can be up to two alliances fighting against each other. In the example of Figure 64, there is the alliance called "*Greece*" fighting against the alliance called "*Troy*". Each alliance may consist of different battle groups (e.g. "*Myrmidons*", "*Trojans*", etc.) each possessing a different volume of soldiers for a specific level. The existence of a battle group may be depended by a character option. In the example of Figure 64, the battle group "Myrmidons" will only exist in this level if the character

called “Achilles” has the option “will-fight” set as “true”. Also, each alliance has a “retreat percentage” associated to it. When this threshold is reached (e.g. when the volume of soldiers of the Greek alliance becomes equal to or lower than 30% or the initial volume of soldiers), the battle will stop and the alliance will retreat.

```
<levels>
  <level>
    (…)
    <is_battle_level>true</is_battle_level>
    <battle_alliances>
      <battle_alliance title="greece" retreat_percentage="30">
        <battle_group volume="500" exist_if_char="achilles"
exist_if_option="will-fight" exist_if_value="true">myrmidons</battle_group>
        <battle_group volume="2000">mycenaeans</battle_group>
        <battle_group volume="2000">spartans</battle_group>
      </battle_alliance>
      <battle_alliance title="troy" retreat_percentage="0">
        <battle_group volume="5000">trojans</battle_group>
      </battle_alliance>
    </battle_alliances>
  </level>
</levels>
```

Figure 64: A set of XML battle details

Each battle group has a set of other attributes as well, which are specified in another XML file created by the storyteller, which is used in every level of the story and is displayed in Figure 65. They have a different fighting ability (which will be used to calculate the outcome of each battle), a total volume of soldiers, and a character name as leader. The volumes included in each individual level cannot exceed the total volume of a battle group which specified here.

```
<battle_groups>
  <battle_group>
    <title>spartans</title>
    <leader>menelaos</leader>
    <total_volume>10000</total_volume>
    <fighting_ability>5</fighting_ability>
  </battle_group>
  <battle_group>
    <title>mycenaeans</title>
    <leader>agamemnon</leader>
    <total_volume>50000</total_volume>
    <fighting_ability>3</fighting_ability>
  </battle_group>
  <battle_group>
```

```
    <title>ithacans</title>
    <leader>odysseus</leader>
    <total_volume>5000</total_volume>
    <fighting_ability>3</fighting_ability>
  </battle_group>
  <battle_group>
    <title>myrmidons</title>
    <leader>achilles</leader>
    <total_volume>1000</total_volume>
    <fighting_ability>10</fighting_ability>
  </battle_group>
  <battle_group>
    <leader>priam</leader>
    <title>trojans</title>
    <total_volume>50000</total_volume>
    <fighting_ability>4</fighting_ability>
  </battle_group>
</battle_groups>
```

Figure 65: A set of XML battle groups

When the BM component is initialised by the World Manager (WM) component (Figure 66), the BM with the help of the Parser component parses all the generic battle group information, and stores them in the Knowledge Base (KB).

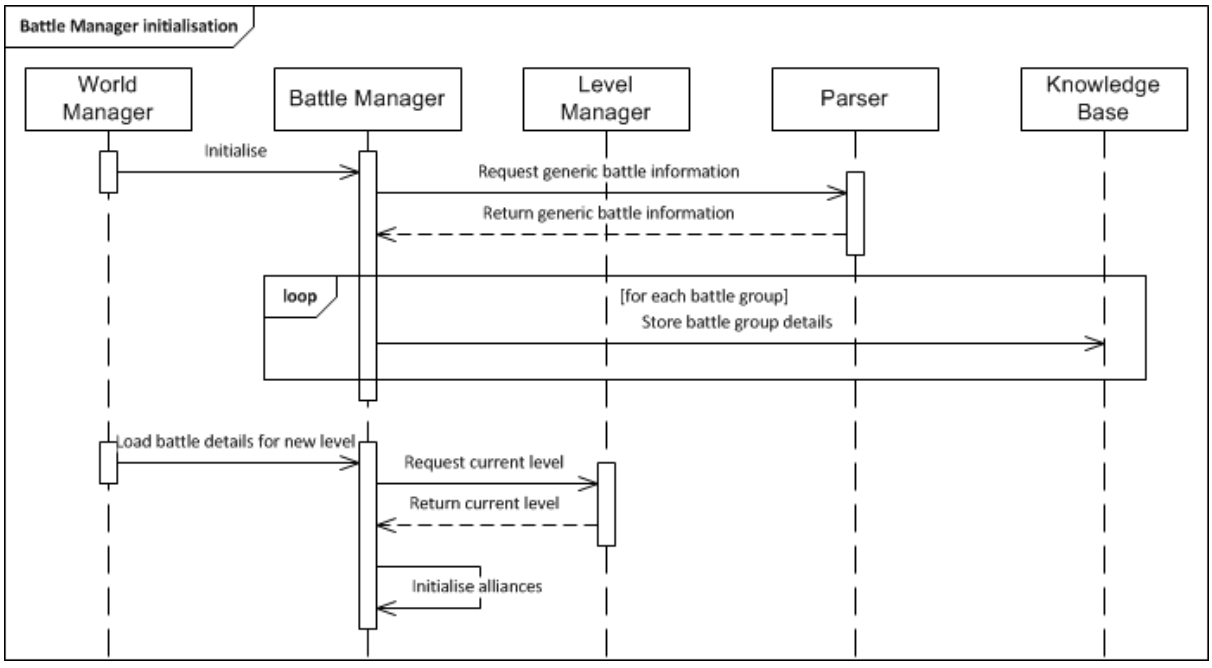

Figure 66: Battle Manager initialisation sequence diagram

When a battle level is loaded, the WM instructs the BM to initialise the alliances so it can be ready to calculate any battles; the BM requests the current level information from the LM, and then it loads the new alliances.

The loading of the alliances is depicted in Figure 67. Initially, the BM loads the details of the new level, and resets any counters which will be used later to deal with the battles. Then, it creates a list of the alliances present in the new level, and for each of the alliances, it creates a list of the battle groups which belong to the alliance.

For each group, the BM first checks if the group has an existence precondition. If it does, then it gets the character option from the KB and check if the precondition is met. If it is not met, then it removes the battle group from the list (i.e. the battle group will not be present and fight in this level). If the precondition is met (or if there is not a precondition set), then the BM retrieves the current total volume of the battle group which is stored in the KB.

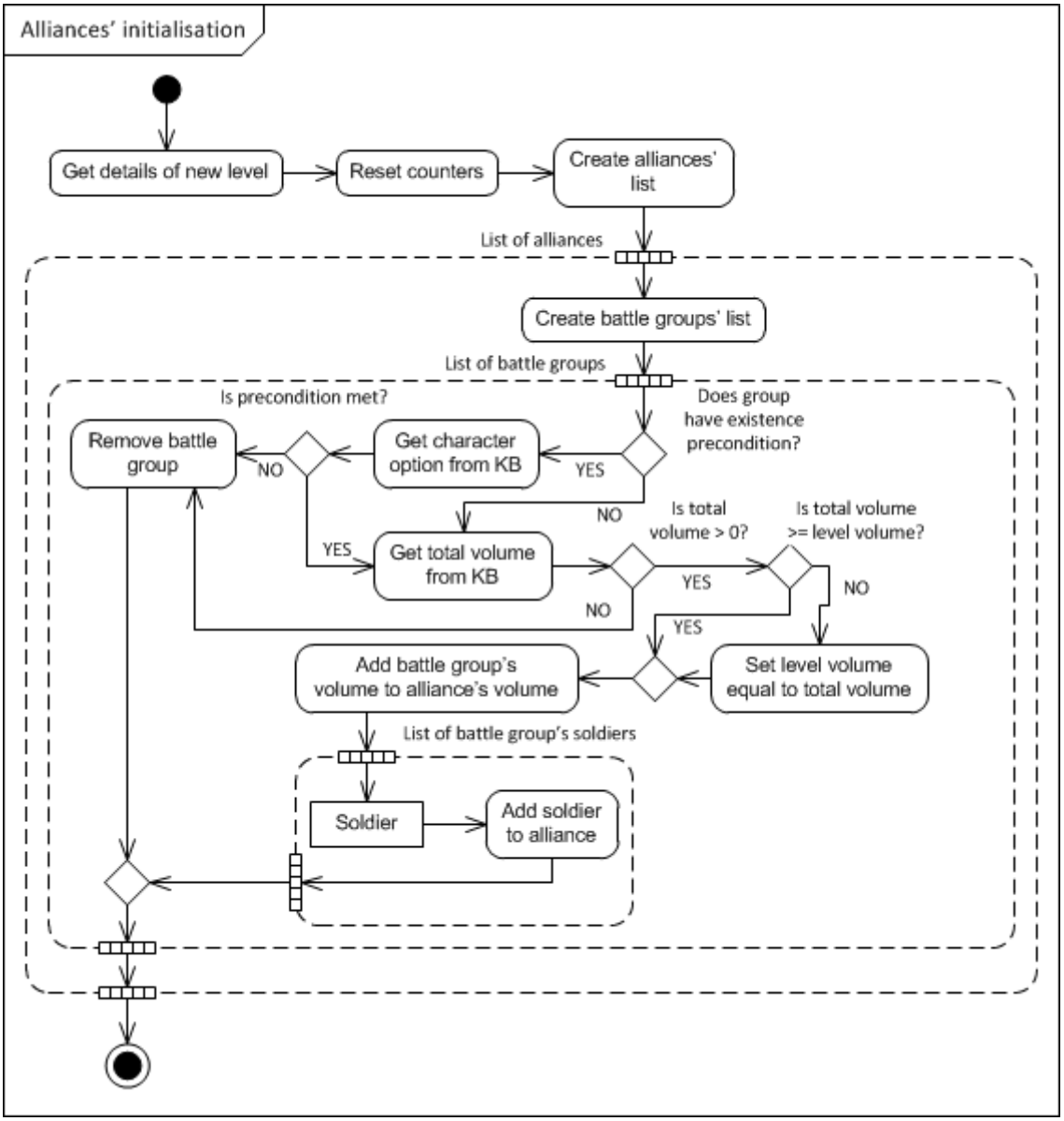

Figure 67: Alliances' initialisation activity diagram

The total volume is irrelevant of a specific level and it is used globally in the story. For example, the total volume for a battle group might be 1000 at the beginning of the story, but as the story progresses, it might be reduced to 100 if 900 of the soldiers died in battles in previous levels.

If the total volume is zero (i.e. no soldier is left in this battle group), then the battle group is removed. If there are soldiers left, then the next step is to check if the total volume is higher than or equal to the required volume of soldiers for the battle group, which refers to this level. If it is not, then the volume of soldiers for this level becomes equal to the total volume.

For example, we might model a level where we specify that there are 300 soldiers of a specific battle group. But, if something happened in a previous level and a lot of soldiers were killed and there are only 100 soldiers left, then only 100 are going to be present in this level instead of 300.

Afterwards, the BM calculates the appropriate amount of soldiers for this level, it adds their volume to the total volume of the alliance they belong to, creates a Soldier instance for each one of them, and adds it to the alliance's collection of soldiers.

As we discussed in section 4.5, when a level is marked as a battle level, a battle occurs in each turn of the level until it stops based on certain retreat conditions. To check if an alliance needs to retreat, the BM first iterates through the list of alliances and calculates the percentage of their current volume, against the volume they started the level with. If the current percentage is either lower than or equal to the retreat percentage of the alliance (specified by the storyteller), then the alliance is marked as "retreated".

After this iteration ends, the BM checks if at least one alliance retreated. If there was one, then the battle is set to complete, and BM iterates through the alliances again. For each one of them, it iterates through their battle groups, and calculates the final volume of the battle group (i.e. the number of soldier who were not killed in battle), and based on this information calculates the new total volume of the battle group which is used globally in the story, and updates the KB with this information. This process is illustrated in Figure 68.

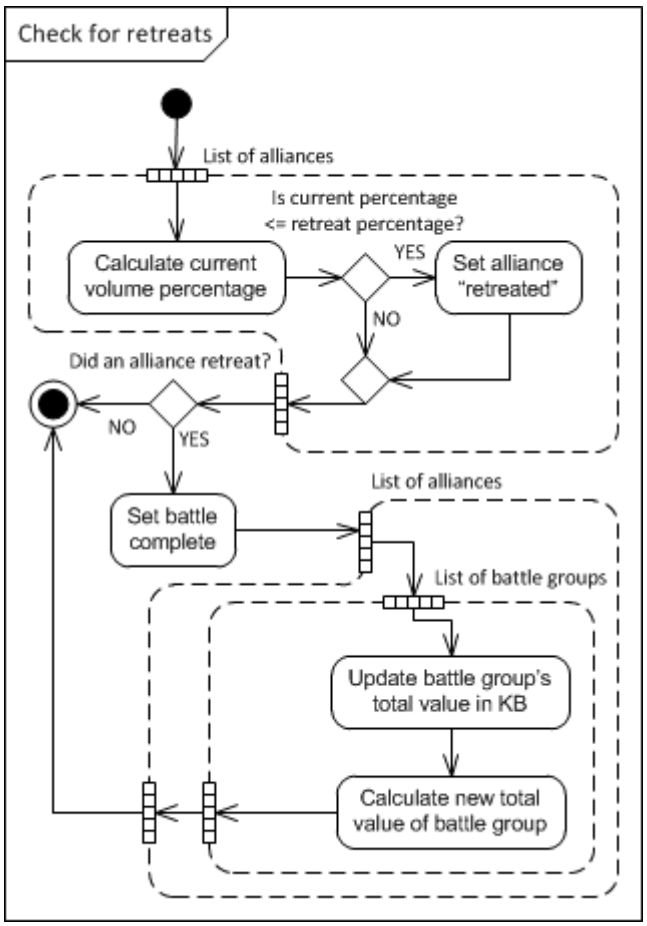

Figure 68: Activity diagram of checking for an alliance retreat

To calculate the outcome of a battle turn (illustrated in Figure 69), the first step for the BM is to discover which the larger one of the alliances is. Afterwards, each soldier of the larger alliance fights (in sequence) a single soldier of the smaller one, until every soldier of the smaller alliance has been involved in a fight.

Then, depending on the configuration of the system by the storyteller, the fight might continue or not. If the configuration is that every soldier should fight at least once, considering that there are still soldiers from the smaller alliance who have won the battle (i.e. they are still alive), they are marked as available, and they fight the remaining soldiers of the larger alliance who haven't been involved in a fight yet.

To summarise, if the storyteller has chosen that every soldier should fight in a battle, then each soldier of the larger alliance will fight a soldier of the smaller alliance once during a battle turn (considering that the smaller alliance has not been eliminated during the battle turn), but some of the soldiers of the smaller alliance will be involved in fight more than once. If there isn't such a requirement, then the amount of fights that will occur in a battle turn is equal to the initial volume of the smaller alliance.

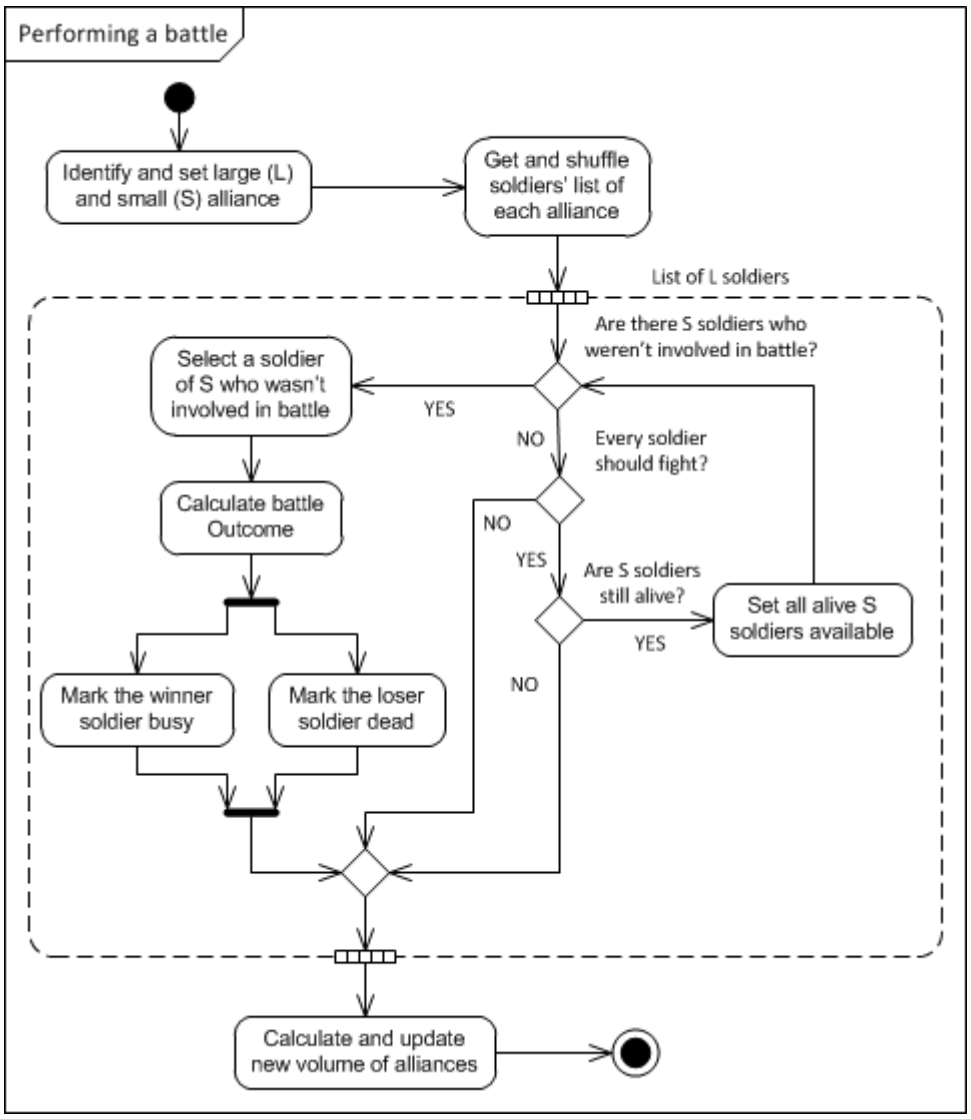

Figure 69: Activity diagram of performing a battle

To calculate the outcome of a fight between two soldiers, their fighting abilities are being used. A higher fighting ability means a larger chance that the soldier will come out from a battle victorious. The BM feeds their fighting abilities to the Oracle, and received the outcome of the battle. The soldier who won is marked as busy (i.e. already involved in a battle), and the other as dead.

At the end of each battle turn, the BM calculates and updates the total volumes of remaining soldiers for each alliance (and battle group), removed the dead soldiers, and resets the status of the rest to available for battle.

Finally, apart from NPCs (i.e. generic soldiers) fighting other NPCs, an agent might fight a soldier as well. Firstly, as we already discussed in section 4.5, the WM needs to check if an agent is in a position to fight. Apart from the agent's availability, the WM checks

with the help of the BM if the agent is located in a battlefield location, and if there is a battle in progress. The battlefield location is specified by the storyteller (Figure 70).

```
<battle_configuration>
  <pddl_battlefield_location_name>battlefield</pddl_battlefield_location_name>
  <percentage_change_of_interruption_to_fight_if_in_battlefield>
    50
  </percentage_change_of_interruption_to_fight_if_in_battlefield>
</battle_configuration>
```

Figure 70: Battle configuration XML nodes

If these conditions are met, then there is a chance that the agent's action will be interrupted by a battle. The WM makes a decision with the help of the Oracle whether the action will be interrupted or not (using the interruption percentage provided by the BM and displayed in Figure 70), and if it does, a battle is performed between the agent and a soldier of the enemy alliance.

To perform the battle, the BM first identifies the alliance in which the agent belongs to, and checks if the enemy alliance has any soldiers alive. If it does, then the BM randomly chooses one and with the help of the Oracle calculates the outcome of the battle and returns it to the WM.

4.17. Evaluation Monitor

During the course of our research, we need to evaluate several aspects of our framework. While some of the evaluations can be implemented by coding small scripts which use some specific components of the framework for the needs of each evaluation, we also need to be able to evaluate aspects of the whole framework and the generated narrative while the story is being executed and generated.

To this end, we designed the Evaluation Monitor (EM) component which can be easily enabled or disabled via a configuration variable and is responsible of gathering the following data while the framework is generating and executing a story:

- For each agent, the volume of its successful, failed, and initiated by other agents actions, the times an agent fought a soldier, and information about an agent's planning and re-planning episodes, wrapped as a group for each individual level the agent participated into.

- For each planning and re-planning episode, the volume of goals, operator nodes, fact nodes, and plan steps, the planning/re-planning duration, and if the planning/re-planning process was successful.
- For each level, its pre-processing duration, the volume of pre-processing fact and operator nodes, its loading duration (including the pre-processing), the list of agents involved in it, the volume of choices and goal injections, and the duration of each turn, calculating it from the time the player chooses to execute the turn, up to the point that is fully executed (i.e. all agents finished with their planning/re-planning processes, they tried to execute their respective actions –if they have any–, and any battles are complete), excluding any time the system waited for a choice by the player during a turn execution.
- For every transition between levels, the duration the transition needed to be calculated/processed, and the volumes of the processed milestones, successors, triggers, character triggers, fact triggers, and mutually exclusive levels.

At the end of each level, the EM produces a report (saved to a text file), which contains all the information mentioned above which corresponds to a single level, including that level's agent information.

At the end of the story, the EM produces another report (also saved to a text file), which contains calculated data combined from all the executed levels. The report includes the following details:

- The total volume of levels;
- the total volume of agent appearances in all the levels, how many unique agents appeared in the story, and the average volume of agents in each level;
- the total, average, maximum (in a single level), and minimum (again in a single level) volume of choices, goal injections, turns, pre-processing fact & operator nodes, milestones, successors, triggers, character triggers, fact triggers, mutually exclusive levels, successful/failed/involved actions, planning/re-planning episodes, goals, planning/re-planning operator & fact nodes, plan steps, and successful/failed plans;

- the total, average, maximum (in a single level), and minimum (again in a single level) duration of turns, pre-processing, loading, transitioning, and planning/re-planning episodes;
- similar data to the above grouped by individual agents.

4.18. Tools for Story Modelling

As we already described in previous sections, DIEGESIS uses a combination of modelling approaches: The storyteller needs to model the game world both in PDDL and in XML. To make the authoring process easier, we are using a PDDL editor created by (Cooper, 2011).

The PDDL editor (illustrated in Figure 71) consists of three parts: an actions editor, used to create new action and assign parameters, preconditions and effects; a predicates editor, used to create new predicates made of parameters and types which return true, and an initial states editor, used to setup the state of the world and everything in it when the story starts.

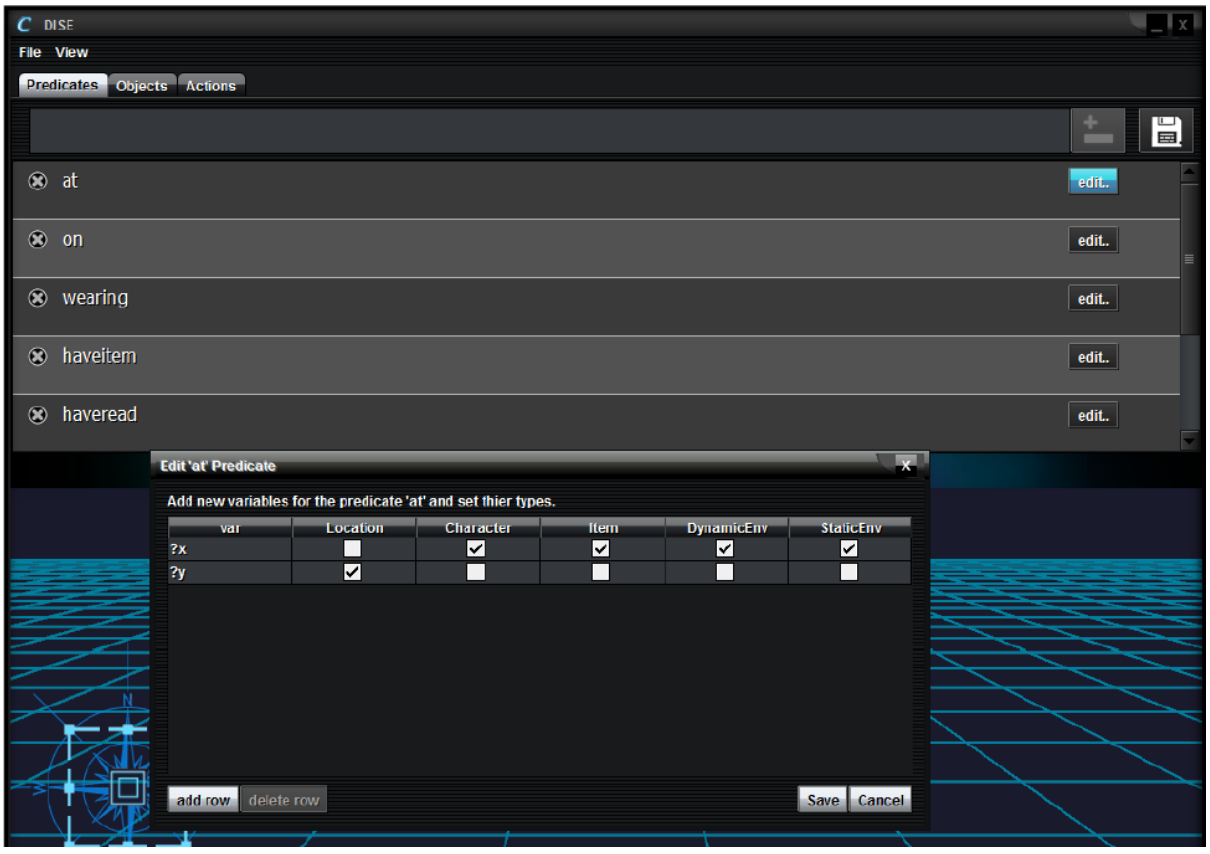

Figure 71: Screenshot of (Cooper, 2011)'s PDDL editor

Although the authoring process in XML is quite easier to the end-user compared to PDDL, we designed an XML editor as an extension to the PDDL one, so the storyteller can be able to create all the XML nodes described in this chapter easier.

In this chapter, we documented and discussed in detail the design aspect of every component of our framework. In the next chapter, we will document and discuss the implementation of our framework.

5 IMPLEMENTATION

In this chapter, we document all the details about the implementation of the multi-agent Digital Interactive Storytelling (DIS) framework we designed in the previous chapter.

5.1. CHOICE OF IMPLEMENTATION PLATFORM

As an implementation platform, we have chosen to use Java, a language that runs in any operating system and includes all the modern programming techniques that we need. For our relational database needs we opted to use MySQL, one of the most widely used open-source relational database management systems (RDBMS).

Figure 72 illustrates DIEGESIS' software architecture. Moving upwards from the operating systems and the technologies we just discussed, we are using the PDDL4J library to be able to parse PDDL files, and a MySQL driver that enables Java to communicate with the MySQL server. Finally, the last stack of our architecture contains user-created content, i.e. a set of PDDL and XML files which (with the help of our framework) produce a story.

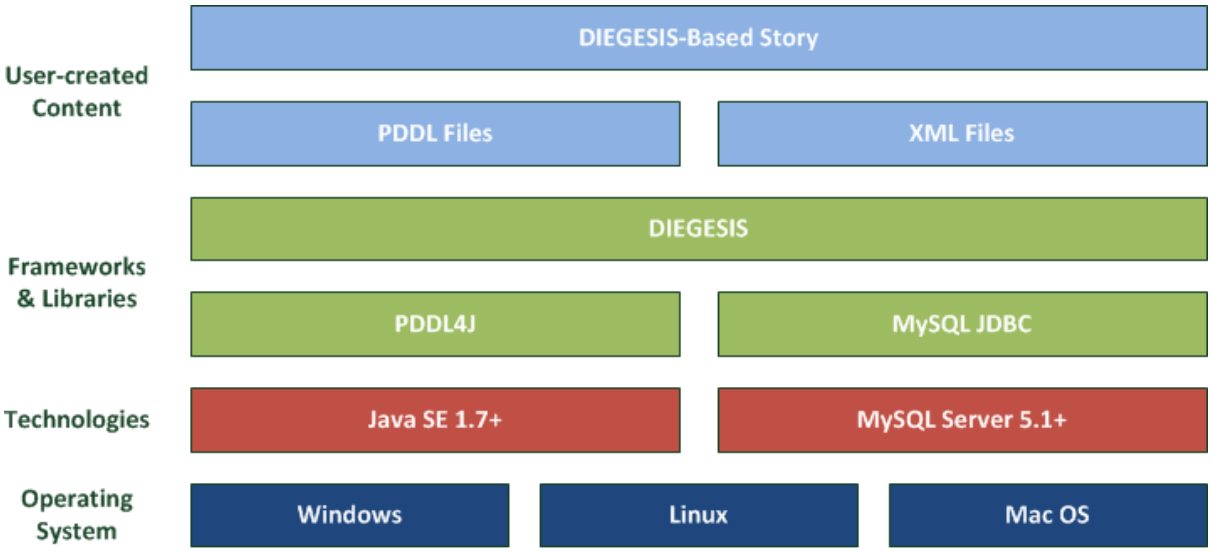

Figure 72: DIEGESIS Software Architecture

5.2. PARSING COMPONENT

Figure 73 illustrates the Parsing Component's implementation. We use two types of parsers; a PDDL and an XML parser. The PDDL parser is implemented using the PDDL4J library (Pellier, 2009) and we do not illustrate its structure for simplicity reasons, since it can be found in its official documentation.

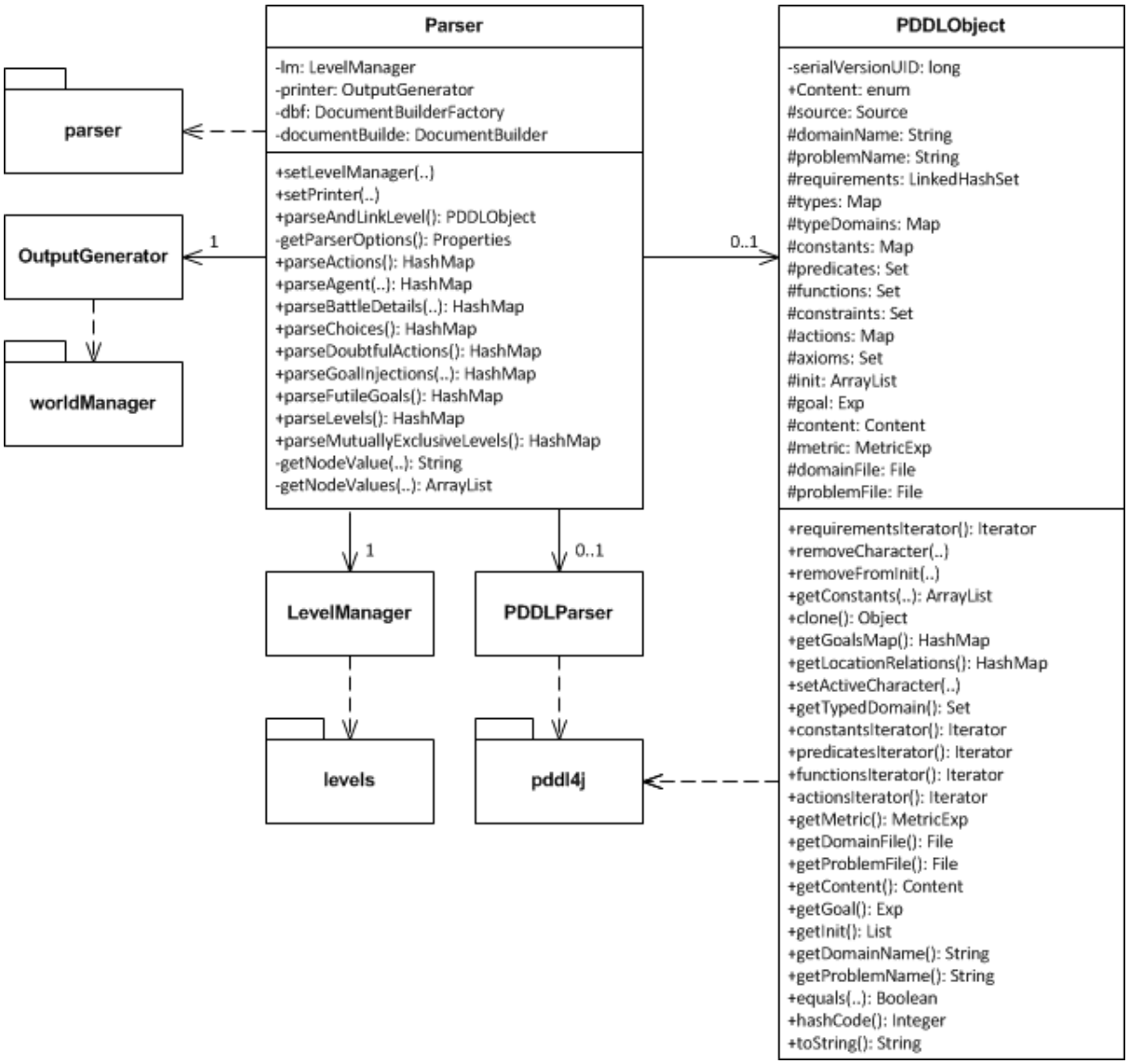

Figure 73: Parser class diagram

The XML parser is part of our Parser implementation, and is used to parse a number of XML files to retrieve any additional constraints and other information applicable to the specified level, based on the semantics we presented and discussed in chapter 4 .

The Parser class always needs an instance of the LevelManager class (discussed in section 4.4) which provides the paths of the currently active level so the Parser can parse the correct files, as well as an instance of the OutputGenerator class (discussed

in section 4.5), to be able to display messages in the console which are used for debugging.

When parsing a level modelled in PDDL, the Parser class uses an instance of PDDL4J's PDDLParser class to perform the parsing of the PDDL files, as well as an instance of PDDL4J's PDDLObject class to represent (and return) the PDDL model resulted by the parsing of the PDDL files.

Considering our needs, we modified PDDL4J's PDDLObject class to be able to perform a few functions that we need, such as to provide lists of specific lists contained in the PDDL representation of the level, e.g. a list of characters.

As we explained in section 4.7, although a character may be present in the model of a level as modelled by the storyteller, he/she might need to be removed from the level based on what happened in previously executed levels. Therefore, the PDDLObject class is now able to remove any references to a character from the PDDL representation of a level if requested.

Finally, the PDDLObject class in now able to identify in which location is every character located and return a map containing this information.

5.3. KNOWLEDGE BASE COMPONENT

As we discussed in section 4.3, DIEGESIS includes two types of Knowledge Bases (KB); a relational database, and information stored in memory. Figure 74 illustrates the implementation of the KB component.

The memory-based part of the KB (KnowledgeBase class) is responsible to keep the current state of the world for the active level, a map with the locations of each character that is active in a level, and two helper lists which are related to the relational database and hold the database ids of the characters and the levels.

The second class illustrated in Figure 74, (i.e. the Db class) is able to perform any function related to the relational database (i.e. select, insert, delete, and update information). As we discussed in section 4.3, the relational database includes tables about characters and their options, levels (and mutual exclusions between them),

milestones, story actions, transitions, and information about the characters and any battle groups. Its schema is illustrated in Figure 75.

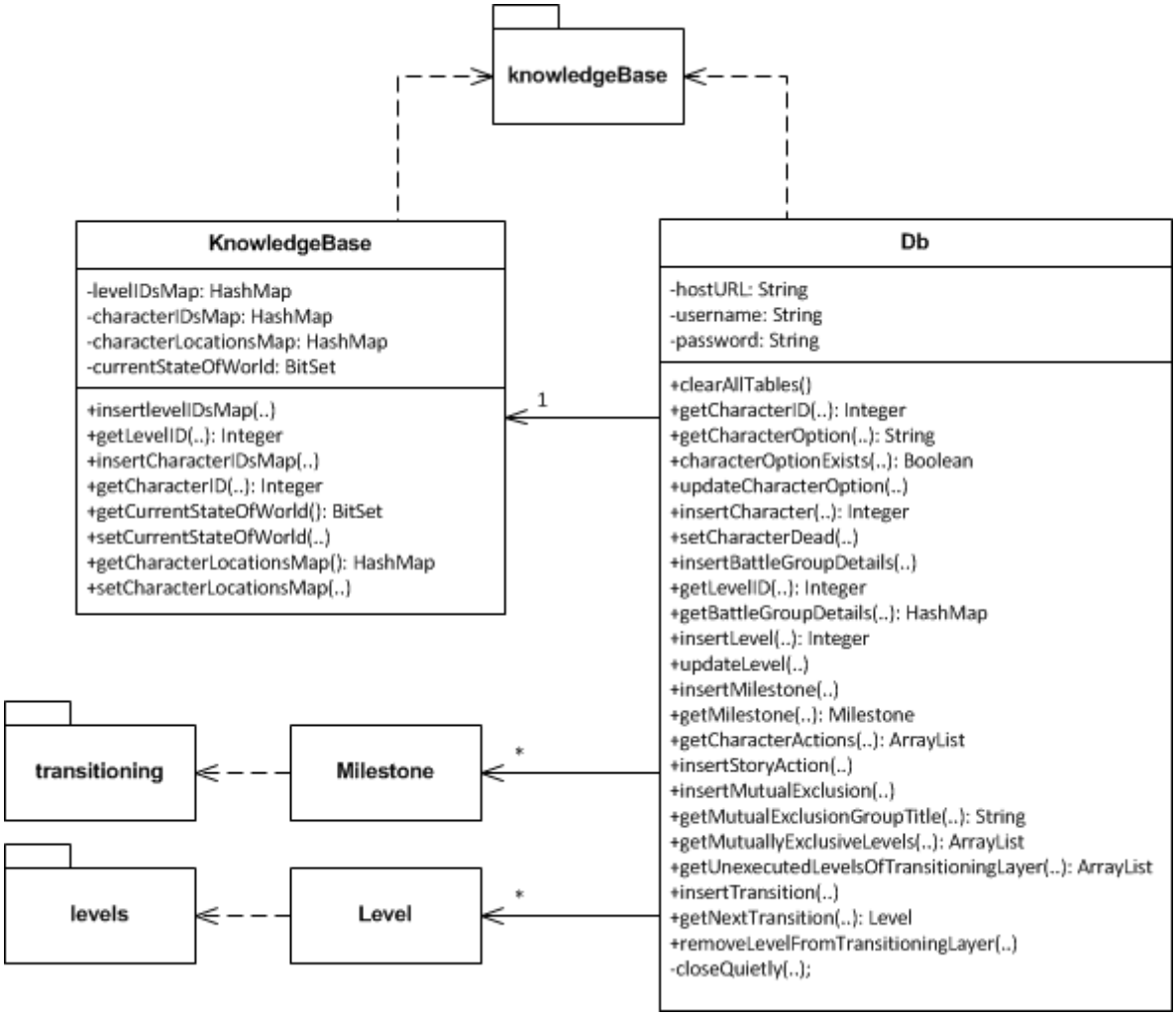

Figure 74: Knowledge Base class diagram

Characters table include the following information about each individual character: The character's unique id (the id's type is integer and it is set to auto increment when a new character is inserted) which is also the primary key of the table, the character's name (which type is varchar), and if the character is alive or not (is_dead, which is Boolean and represented in MySQL by a tinyint).

Characters_options table is able to store the characters' options described in section 4.7. The table is designed in a way to allow the storyteller to represent any types of options. It includes the name of the option (varchar), the character's id (character_id – integer) which is a foreign key pointing to an id of a character in the characters table, and the value of the option (varchar). Option and character_id are a double primary key for this table.

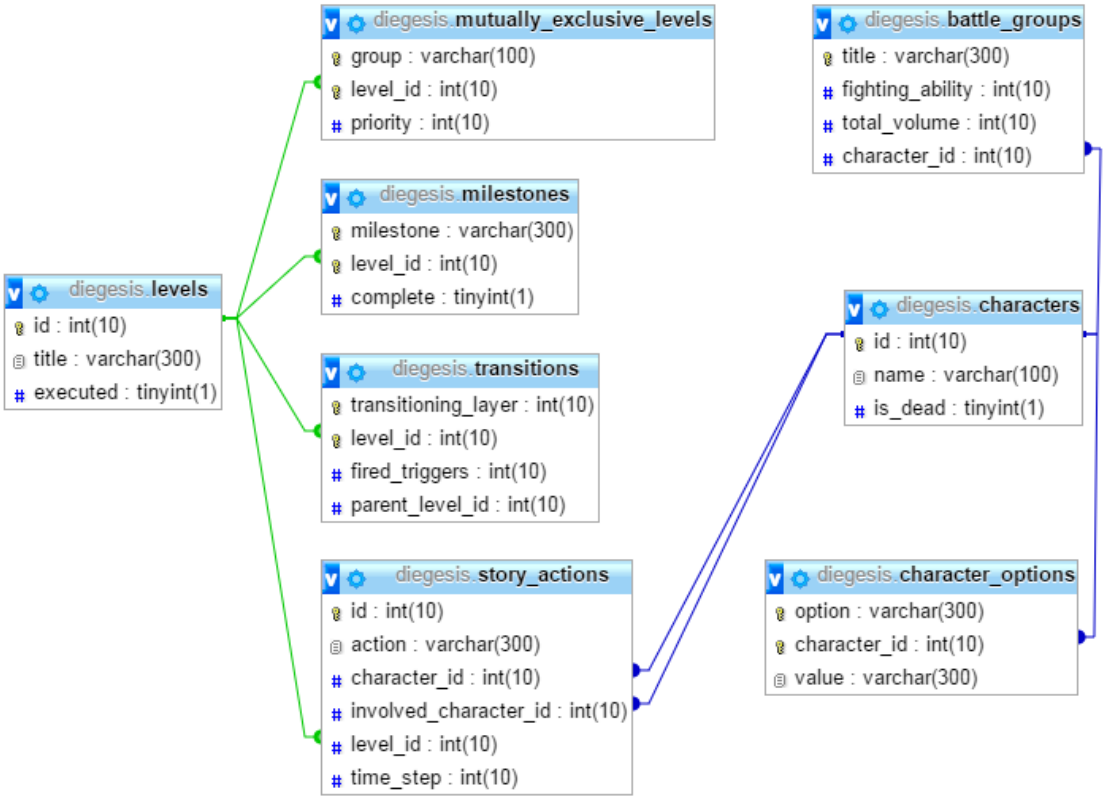

Figure 75: Knowledge Base database schema

Levels table contain information about the different possible levels which are present in the story. It contains a unique id for each level (which type is integer, auto-increments, and it's the primary key of the table), the level's title (type), and if it was executed (Boolean) or not. Any mutual exclusion between levels is stored in the mutually_exclusive_levels table. The information stored there includes the mutual exclusion group's name (varchar), the level's id (level_id – integer; foreign key of levels table id), and the priority (integer) of the level. Group and level_id are a double primary key for the mutually_exclusive_levels table.

The milestones table is used to store the milestones (varchar) of each level (level_id – integer; foreign key of levels table id), along with each milestone's state, i.e. if the milestone is complete (Boolean) or not. Milestones and level_id are a double primary key for the milestones table.

Transitions table holds information about all the transitions which occurred or will occur during the execution of the story. It includes the transitioning_layer (integer) in which the transition occurs, the id of the level (level_id – integer; foreign key of levels table id) which the transition refers to, the number of fired_triggers (integer) of the

transition, and the id of the level (parent_id – integer; foreign key of levels table id) which triggered the transition. The fields transitioning_layer and level_id are a double primary key for the transitions table.

The story_actions table is used to store and keep track of all the actions that occurred during the execution of the story, along with information about them. Specifically, it includes a unique id (integer; auto-increment; primary key) for each action, the actual action (varchar), the ids of the characters who either executed the action or are involved in it (character_id & involved_character_id – integer; foreign keys of characters table id), the level_id (integer; foreign key of levels table id) in which the action occurred, and the time_step (int) in which the action was executed into.

Finally, the battle_groups table includes the following information about any battle groups present in the story: The group's title (varchar; primary key), the group's leader (character_id – integer; foreign key of characters table id), the group's fighting_ability (integer), and the group's total_volume (integer).

5.4. Level Manager

As we already discussed in section 4.4, The Level Manager (LM) component is responsible of keeping track of most of the information about each possible level of the story.

The implementation of the LM component is illustrated in Figure 76. The LevelManager class uses an instance of the Parser class to parse information related to each level, and an instance of KnowledgeBase and Db classes to store whatever is needed to the Knowledge Base. It also uses an instance of the BattleManager class to request any information needed which is related to a battle which may occur in a level.

Each level is represented by a Level class, which stores all the information related to a level which was previously parsed. Each level may contain an infinite amount of BattleAlliance instances (each BattleAlliance instance may contain an infinite amount of BattleGroup instances) which represent battle information about a level, and are further discussed in section 5.16.

Each level class may also include an infinite amount of: CharacterOptionsGroup instances (with each CharacterOptionsGroup instance able to contain an infinite

amount of CharacterOptionsGroupOption instances); Trigger instances; CharacterTrigger instances; and FactTrigger instances. All of these classes contain information which is used by the Transitioning Manager, and are further discussed in section 5.7.

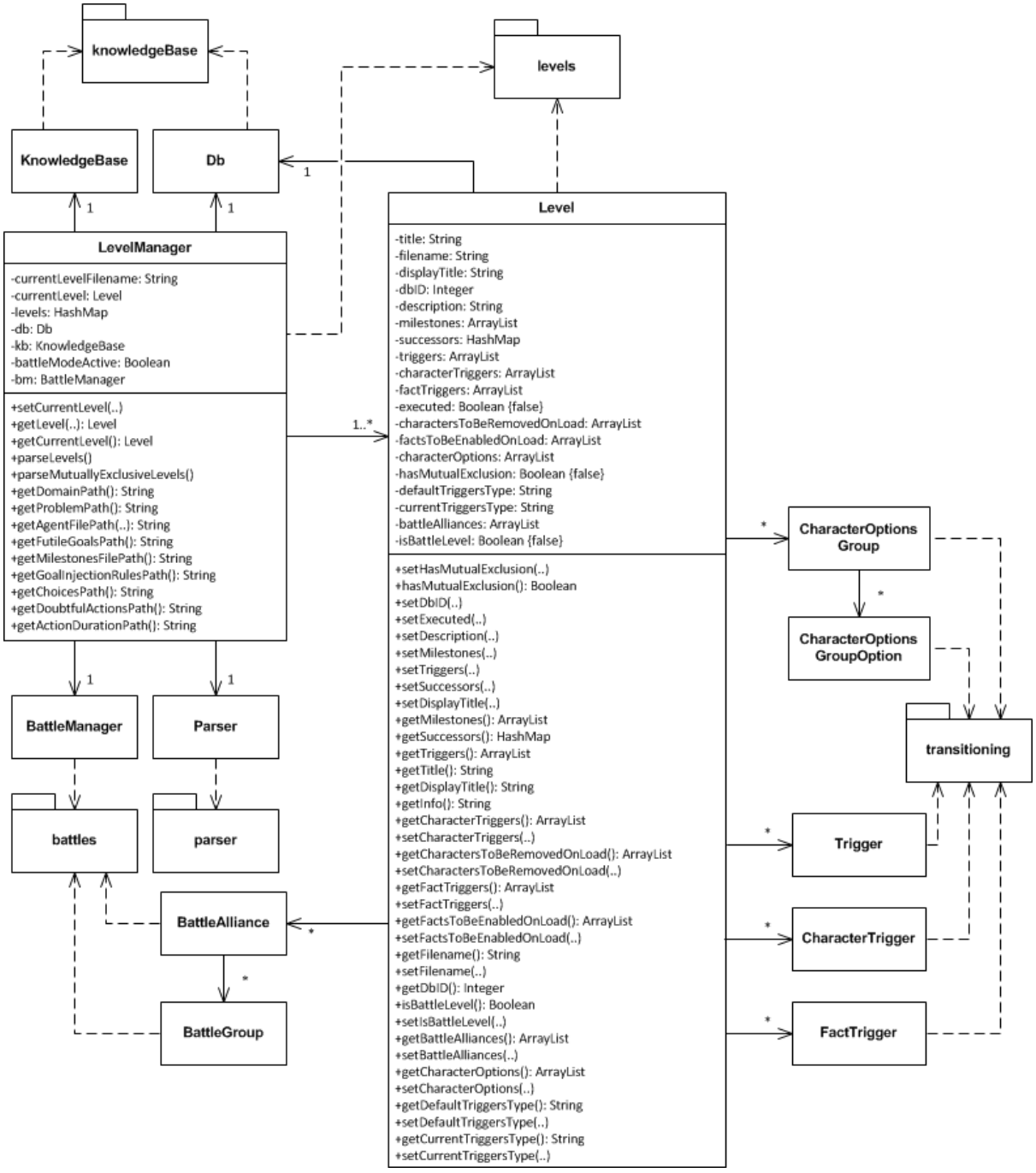

Figure 76: Level Manager class diagram

5.5. World Manager

As discussed in section 4.5, the World Manager (WM) is the main component which coordinates the whole system, and it has direct access to most of the other

components of the system. The implementation of the WM component is illustrated in Figure 77).

The WorldManager class uses an instance of the TimeManager and the OutputGenerator classes. The TimeManager class is responsible of keeping track of the time (i.e. turns) in a level and globally in the story. The OutputGenerator class is responsible of sending messages to the User Manager component to be displayed to the player, or print messages to the console for debugging purposes, and includes a few templates such as a header template, an alert message template, etc.

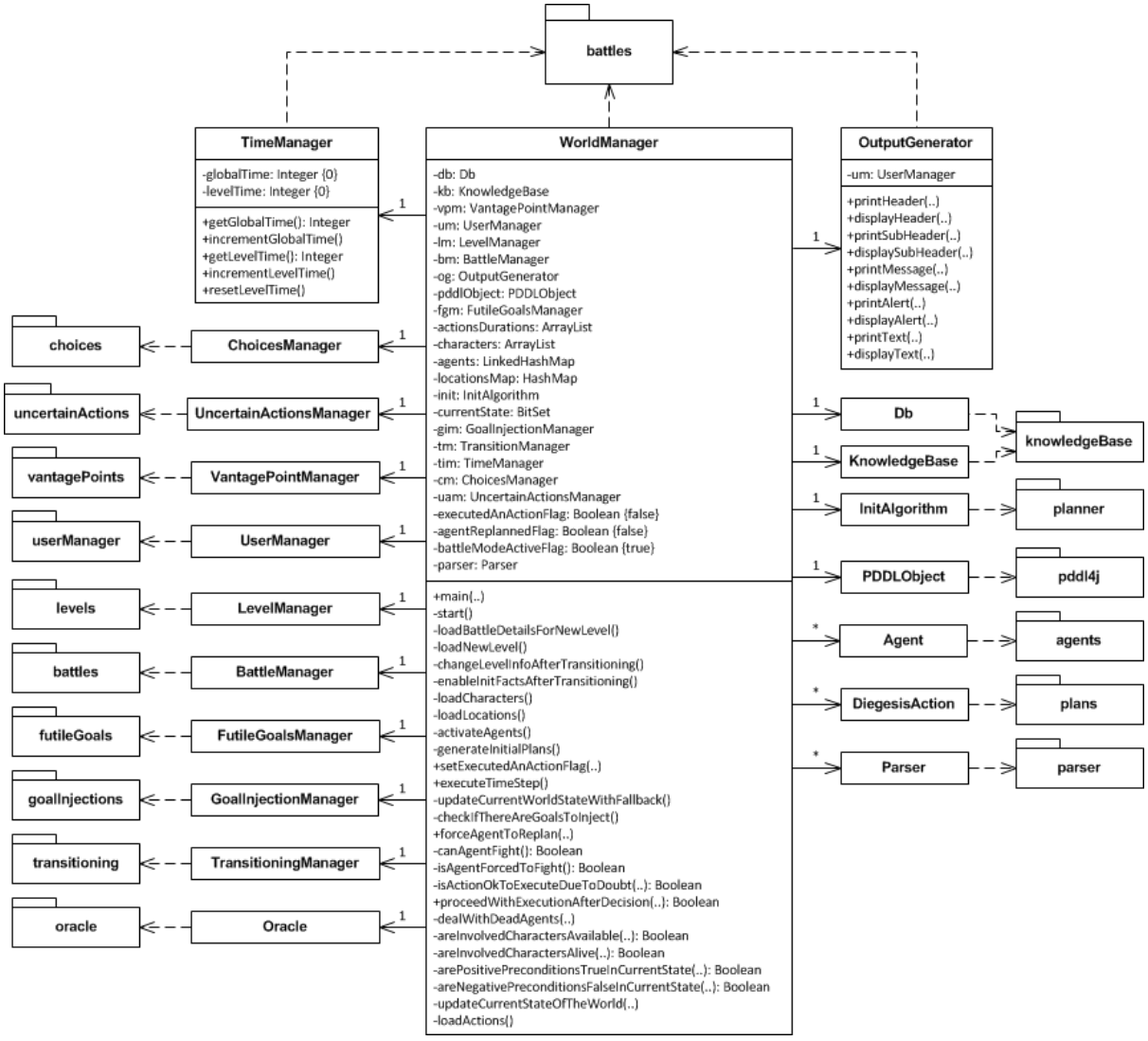

Figure 77: World Manager class diagram

As we already mentioned, the WM has access to most of the other main components of the system, either to initialise them (and have them help the WM initialise new levels), or to help during the execution of the story. All of the classes of Figure 77

which does not contain variables and functions are discussed in detail in their own sections.

Apart from initialising the system and the currently active level with the help of the rest component of the system, its main responsibility is to dictate the execution of the agent's actions based on their individual plans, keeping track of the current state of the world, etc.

Algorithm 1 presents the execution of the agents' plans in a given time step (i.e. turn). Initially, the WM informs the User Manager (UM) component that the execution of a turn has started. Then, it checks (with the help of the Level Manager) if the currently active level is a battle level, and, with the help of the Battle Manager (BM), if the alliances are initialised. If the level is a battle level and the alliances are not initialised, then it instructs the BM to initialise them. Afterwards, if a battle is still in progress (i.e. an alliance not retreated in a previous turn) it instructs the BM to check if an alliance needs to retreat. If the battle ended due to a retreat, it retrieves the retreat info from the BM and updates the current state of the world; alternatively it instructs the BM to perform a battle.

Then, the WM queries all the agents one by one to deal with their plans. Initially, checks if an agent is dead. If it is, then it moves to the next agent.

The next step is to identify if the agent has an active plan. If there is a plan, then the WM requests the set of actions the agent wants to execute. Alternatively, it instructs the agent to generate a new plan, and then requests the actions set (considering that the agent successfully generated a plan).

Then, the WM checks if the agent is busy at the given time step (i.e. if it is involved in another action initiated by another agent already), as well as if the agent returned a set of actions to execute.

If the agent is busy, then the WM moves to the next agent. If the agent didn't return a set of actions (i.e. doesn't have a valid plan to execute), the WM with the help of the BM checks if the agent is able to fight. If it is, then the WM instructs the BM to perform a battle between the agent and an enemy soldier.

Considering that the agent is not busy and has a set of actions to execute, the WM iterates through the actions of the action set. For each of the actions, the WM identifies if any other agents are involved in the action, and checks their availability as well.

If any of the involved agents is not available, and the action is not interruptive (meaning that the availability of the ignored agents will be ignored) then the WM checks if all of them are still alive. If they are, then the agent is instructed to wait (the action remains pending). If at least one of them is dead, then the action is impossibly to be executed, therefore the action is marked as failed, the execution of all other actions is stopped, the current state of the world is sent to the agent, and the agent is instruct for any completed goals (based on the new state of the world), and ultimately re-plan.

ALGORITHM 1 – Multi-agent execution of plans

Input: A: a list active agents; T: a time step; C: the current state of the world; UM: UserManager; BM: BattleManager; TiM: TimeManager: L: currently active level
Output:

```
1:  inform UM that turn execution started;
2:  if (L is battle level) AND (alliances are not initialised) do:
3:      instruct BM to initialise alliances;
4:  end if

5:  if battle in progress do:
6:      instruct BM to check if an alliance needs to retreat;
7:      if battle ended do:
8:          get retreat info from BM;
9:          update C with retreat info;
10:     else
11:         instruct BM to perform battle;
12:     end if
13: end if

14: for each A as agent do:
15:     if agent is dead do:
16:         go to line 89;
17:     end if

18:     update agent with C;
19:     if agent has plan do:
20:         get next set of actions from plan;
21:     else
22:         generate new plan;
23:         if agent has plan do:
24:             get next set of actions from plan;
25:         end if
26:     end if

27:     if agent not busy at T AND size of actions > 0 do:
29:         for each actions as action do:
30:             identify agents involved in action;
31:             if agents not busy at T OR action is interruptive do:
```

```
33:             if preconditions of action exist in C do:
34:                 if action marked as choice do:
35:                     get choice result (YES/NO) either from player or from system;
36:                 else
37:                     choice = YES;
38:                 end if

39:                 if choice = YES do:
40:                     if agent can fight AND agent is forced to fight do:
41:                         instruct BM to perform battle between agent and enemy soldier;
42:                     stop execution of all other actions;
43:                     else if action is clear to execute due to doubt do:
44:                         update C with action effects;
45:                         mark action as executed;
46:                         set involved agents busy at T;
47:                         if action is duel do:
48:                             instruct BM to perform battle between agents;
49:                             update C with battle outcome;
50:                         end if
51:                         deal with any dead agents;
52:                         store action details in KB;
53:                         inject new goals to agents based on changes in C;
54:                     end if

55:                 else if choice = NO do:
56:                     mark action as failed;
57:                     update C with action's fallback;
58:                     update agent with C;
59:                     instruct agent to check for completed goals;
60:                     instruct agent to re-plan;
61:                     stop execution of all other actions;
62:                 end if
63:             else
64:                 mark action as failed;
65:                 stop execution of all other actions;
66:                 instruct agent to re-plan;
67:             end if

68:         else
69:             if all involved agents alive do:
70:                 instruct agent to wait;
71:             else
72:                 mark action as failed;
73:                 update C with action's fallback;
74:                 update agent with C;
75:                 instruct agent to check for completed goals;
76:                 instruct agent to re-plan;
77:                 stop execution of all other actions;
78:             end if
79:         end if
80:     end for each

81:     if at least an action was executed do:
82:         inform agent which actions were executed;
83:         set agent busy in T;
84:         update agent with C;
85:         instruct agent to check for a location change;
86:         instruct agent to check for completed goals;
87:     end if

88: else
89:     if agent can fight do:
90:         instruct BM to perform battle between agent and enemy soldier;
91:     else
92:         instruct agent to wait;
93:     end if
```

```
94:     end if
95:   end for each

96:   set TiM's level time to T+1 and increment global time;
97:   inform UM that turn execution finished;
```

Afterwards, the preconditions of the action are checked against the current state of the world. If the preconditions are met, then the system checks (with the help of the Choices Manager) if the action is marked by the storyteller as a choice. If it is, then a choice needs to be made. If the action is not marked as a choice, then the WM sets the outcome of the choice as "YES".

If the choice is "NO", then the action is marked as failed, the current state of the world is updated with the fallback of the choice, the agent is updated with the new current state of the world, the agent is instructed to check for any completed goals and to re-plan, and the WM stops the execution of all other actions of the agent.

If the choice is "YES", then the WM checks (with the help of the BM) if the agent is able to fight, and if the agent will be forced in a fight interrupting the execution of the action. If the agent needs to fight, then the BM is instructed to perform a battle between the agent and an enemy soldier, and the WM stops the execution of any remaining actions of this agent.

If the agent evades the fight (or if a fight is not applicable due to the level not being a battle level or the agent being in a location which is unsuitable for a fight) then the WM checks (with the help of the Uncertain Actions Manager) if the action is clear to be executed due to doubt. If it is not, then the action remains pending for this turn.

Alternatively, the current state of the world is updated with the effects of the action, the action is marked as executed, and the involved agents are set as busy for this time step. Then, the WM checks if the action is a duel between two agents. If it is, then it instructs the BM to calculate the outcome of the battle, and updates the current state of the world with it.

Then, the WM deals with any dead agents, stores the executed action details to the Knowledge Base, and (with the help of the Goal Injection Manager) injects new goals to agents based on the changes on the current state of the world.

After the end of the actions execution, if at least an action was executed, the WM informs the agent of which actions were executed (i.e. are complete), sets the agent busy for the time step, sends to the agent the updated current state of the world, and instructs the agent to check for any location change, as well as for any completed goals.

Finally, when the agents list is finished, the WM instructs the Time Manager to iterate the global and level time, and informs the UM that the execution of the turn is finished.

5.6. CHOICES MANAGER

As discussed in section 4.6, the storyteller has the ability to mark actions as choices, and the Choices Manager (CM) is responsible of keeping the relevant information. The implementation of the CM component is illustrated in Figure 78.

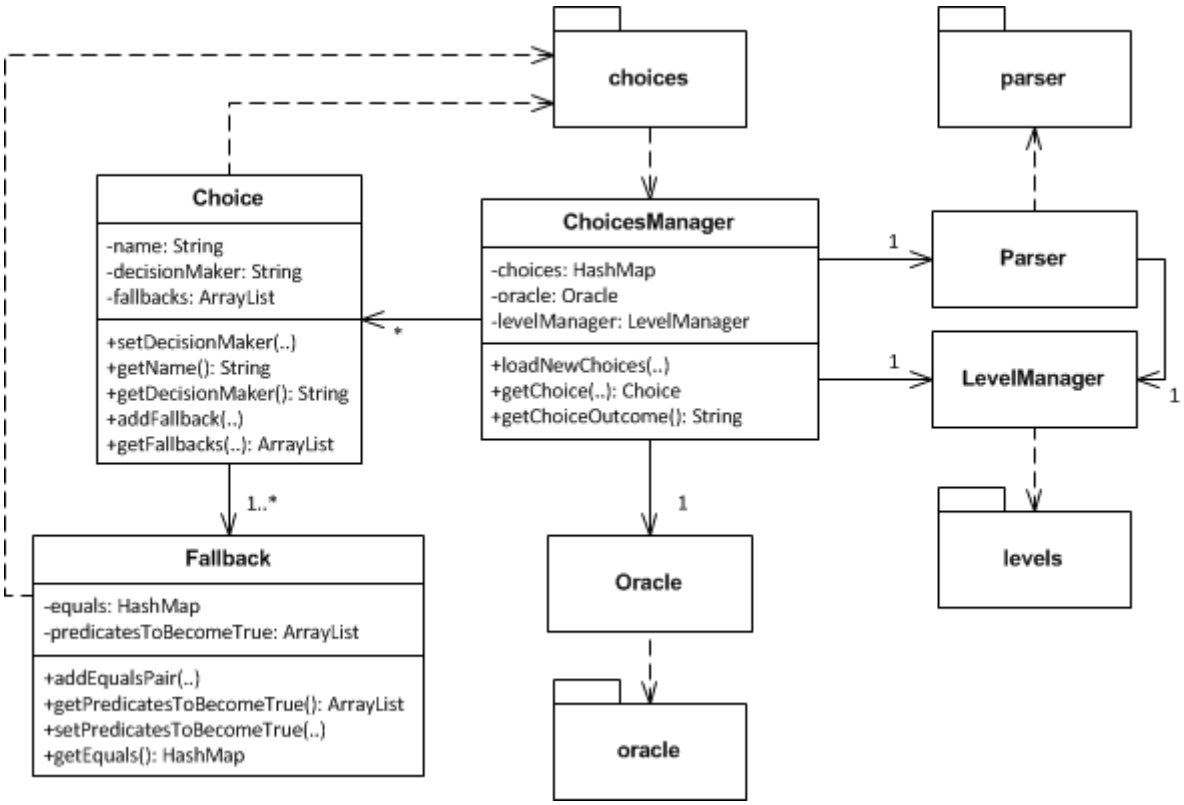

Figure 78: Choices Manager class diagram

The ChoicesManager class uses an instance of the Parser class to be able to parse the choices of a new level, as well as an instance of the LevelManager class (which is used to be passed to the Parser class), and an infinite amount of Choice class instances. Each instance of the Choice class represents a choice for a specific level, and may contain

from one to an infinite amount of fallbacks (each fallback is represented by an instance of the Fallback class).

Finally, the ChoicesManager class uses an instance of the Oracle class (discussed in section 5.10) to be able to make a choice if the decision maker is the system (or the player is unable to make a decision at that time).

5.7. TRANSITIONING MANAGER

As discussed in section 4.7, when a level of the story ends, it is a time for a new level to be selected and enabled, and that's what the Transitioning Manager (TM) component is responsible of. The implementation of the TM component is illustrated in Figure 79.

The TransitionManager class is the class which (when asked by the World Manager component) can calculate all the transitions deriving from the level which just completed its execution, and select the next level which will be loaded for execution, using the processes described in section 4.7. It uses an instance of the LevelManager class to be able to request information about any levels involved in the transitioning process, as well as an instance of the KnowledgeBase and Db classes to be able to retrieve and store information about milestones, triggers, etc.

As we explained in section 4.7, one of the transitioning steps is to check the state of each of the milestones of the level against the current state of the world, and store the relevant information in the Knowledge Base. Each instance of the Milestone class represents a milestone of a level. To be able to map them against the current state of the world, the TrasitionManager class uses an instance of the InitAlgorithm class, which will be further discussed in section 5.14.

During the calculation of any future transitions and the selection of the next applicable level, the TransitionManager class may also use multiple instances of the Level class. As we discussed in section 4.7, each level is associated with a number of triggers, fact triggers, character triggers, and character options. Each of these concepts are represented respectively by an instance of the Trigger class, the FactTrigger class, the CharacterTrigger class, or the CharacterOptionsGroup class (which can contain multiple instances of CharacterOptionsGroupOption class).

Finally, the TransitionManager class may use instances of the MutuallyExclusiveLevel class to keep information about mutually exclusive levels and deal with them (as discussed in detail in section 4.7).

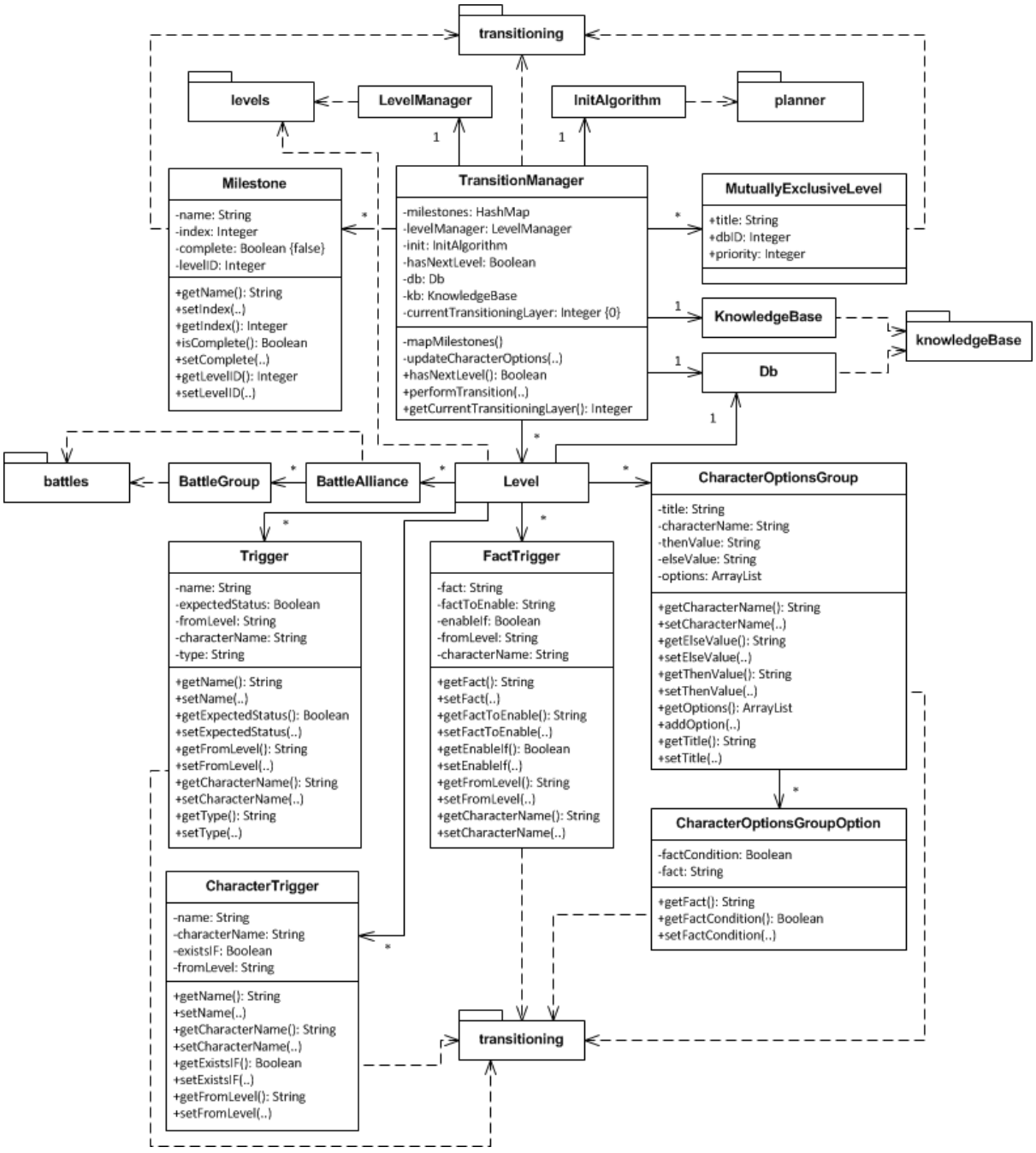

Figure 79: Transitioning Manager class diagram

5.8. GOAL INJECTION MANAGER

As we discussed in section 4.8, while the story is executed the Goal Injection Manager (GIM) constantly monitors the current state of the active level with the help of the

World Manager (WM), to identify if a goal needs to be injected to the goal list of an agent/character based on goal injection rules created by the storyteller.

The implementation of the GIM component is illustrated in Figure 80. The GoalInjectionManager class uses an instance of the LevelManager class and an instance of the Parser class to be able to parse and prepare the goal injection rules of a new level. To help mapping these rules against the current state of the world, an instance of the InitiAlgorithm class along with an infinite amount of Fact class' instances are used. Both of these classes are part of the Planner component and they are discussed in section 4.14.

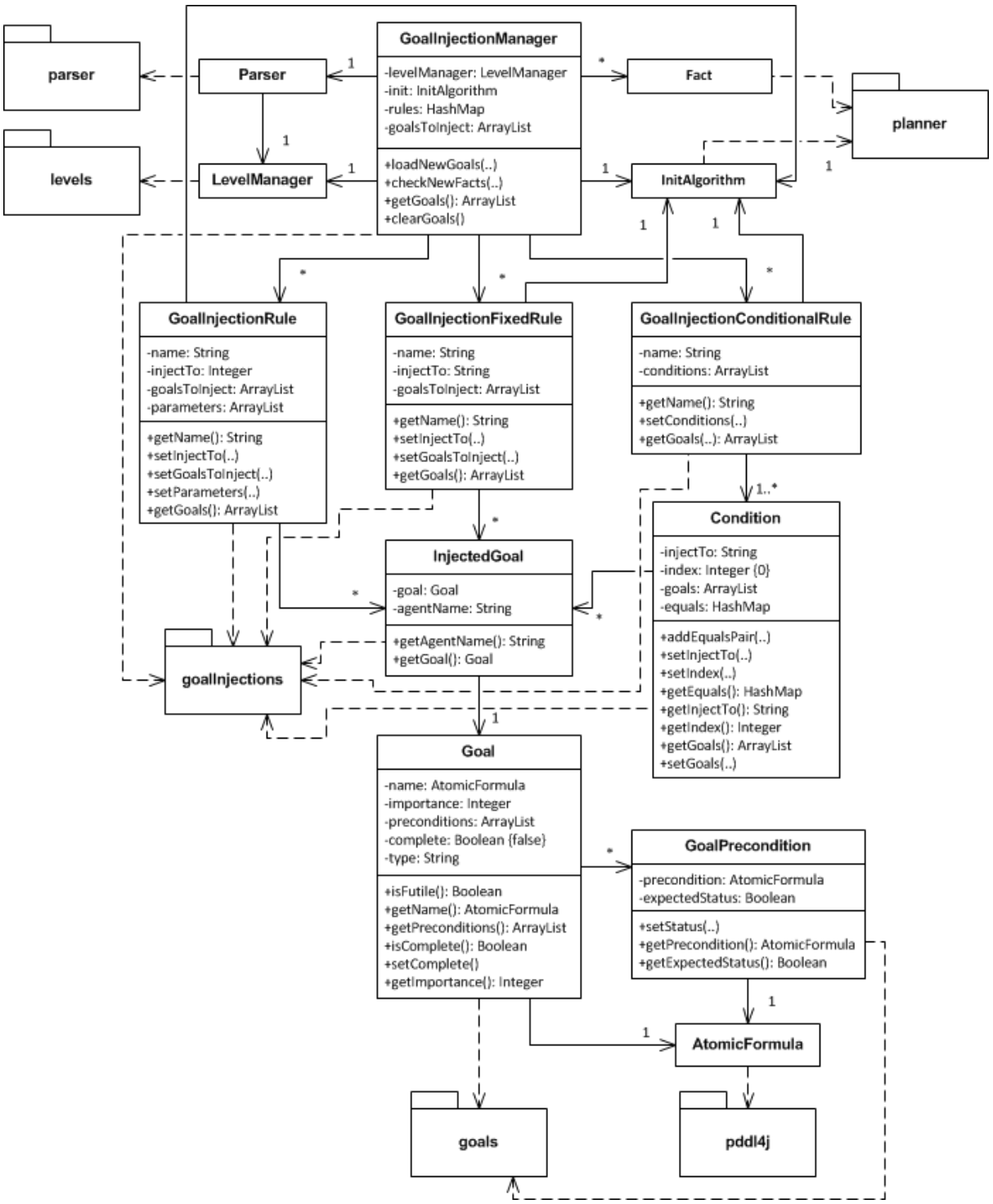

Figure 80: Goal Injection Manager class diagram

Each different type of goal injection rules (i.e. default, fixed, and conditional) are represented by a different class (i.e. GoalInjectionRule, GoalInjectionFixedRule, and GoalInjectionConditionalRule respectively). GoalInjectionConditionalRule instances include at least one condition, in the form of a Condition class instance.

All of the classes which represent goal injection rules ultimately return a list of goals to be injected, in the form of InjectedGoal instances. Each instance of the InjectedGoal

class includes the name of the agent in which the goal will be injected to, along with the goal which will be injected. The goal is represented by an instance of the Goal class, containing information about the goal such as its importance, its PDDL name (AtomicFormula), etc. A goal may also include the names of other goals as preconditions along with their expected status, which are represented by instances of the GoalPrecondition class.

5.9. FUTILE GOALS MANAGER

As discussed in section 4.9, the idea behind the creation of futile goals is that sometimes (depending on the story) the characters present in a level may not have any important goals to achieve, so instead of having them staying there doing nothing, the storyteller can specify a set of futile goals which are applicable in a level, and they can be used to keep the characters busy. The implementation of the Futile Goals Manager component is illustrated in Figure 81.

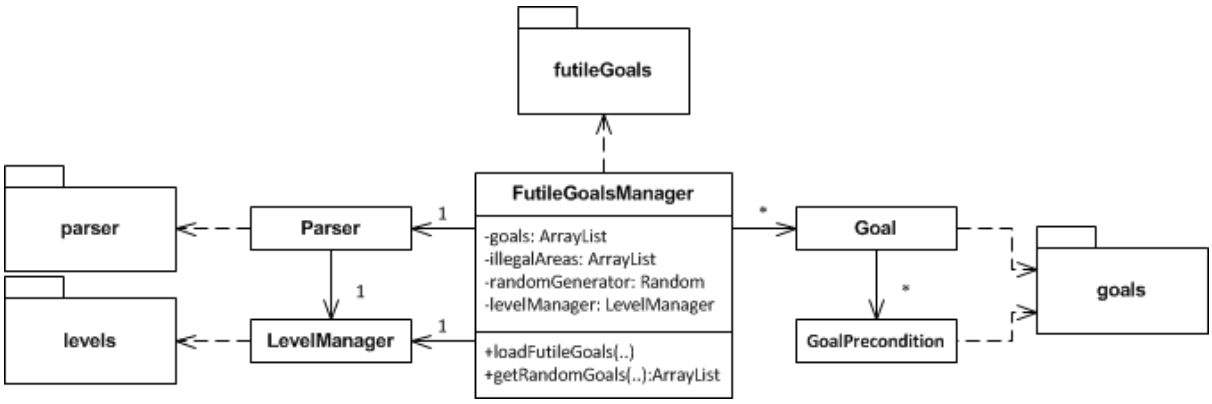

Figure 81: Futile Goals Manager class diagram

The FutileGoalsManager class uses an instance of the Parser class as well as an instance of the LevelManager class to parse and load the futile goals of a new level, and store them in a goals list. Each goal is represented by an instance of the Goal class and may contain a number of preconditions represented by instances of GoalPrecondition class as discussed in section 5.8. When requested to do so, the FutileGoalsManager class selects a goal randomly and returns it to the World Manager component so it can be added to the agent who requested it.

5.10. ORACLE

As discussed in section 4.10, there are certain situations during the generation and execution of a story, where a relatively random outcome needs to be calculated. The Oracle component deals with these random outcome calculations. Its implementation is depicted in Figure 82.

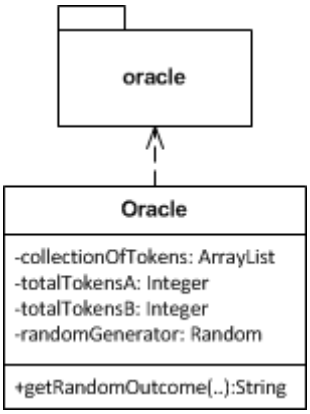

Figure 82: Oracle class diagram

The Oracle class is self-contained and when requested to do so, it is able to calculate and return a random outcome between different candidates.

5.11. UNCERTAIN ACTIONS MANAGER

As discussed in section 4.11, there are some types of actions which make sense that they should have a percentage that will succeed due to chance. We call these actions "uncertain", and the component which deals with them the Uncertain Actions Manager (UAM). UAM's implementation is displayed in Figure 83.

Like in most of DIEGESIS' components, UncertainActionsManager component uses an instance of the Parser class as well as an instance of the LevelManager class to parse and load the uncertain actions of a new level, and then stores them in an actions list. Each uncertain action is represented by an instance of the UncertainAction class.

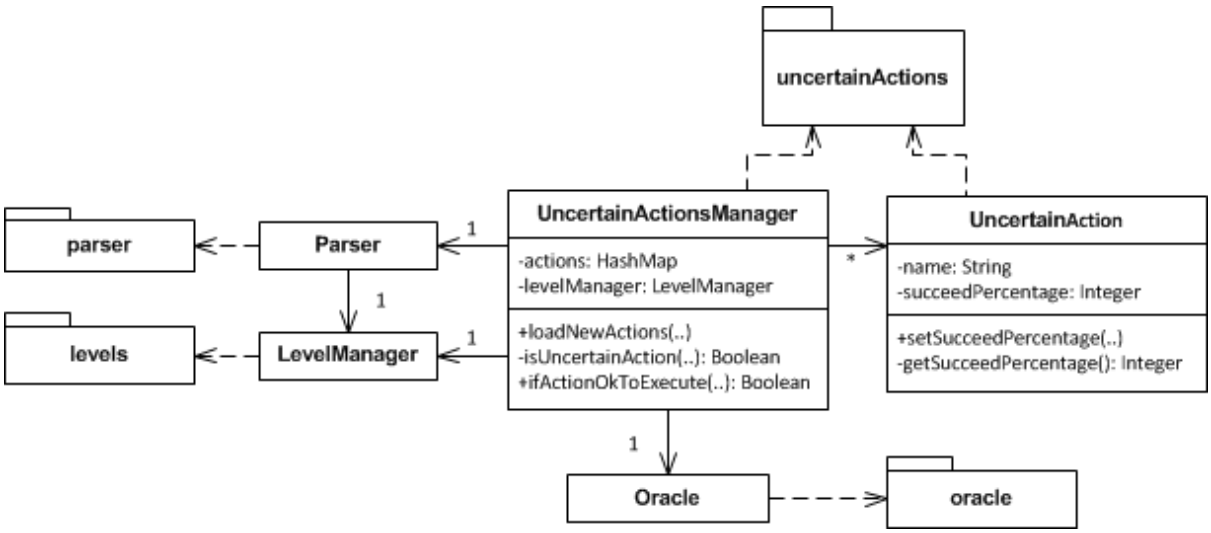

Figure 83: Uncertain Actions Manager class diagram

When requested by the World Manager component, it checks if an action is in the list of uncertain actions, and if it is then uses an instance of the Oracle class to calculate an outcome which represents if the action is cleared to be executed or not.

5.12. Vantage Point Manager

As we discussed in section 4.12, the player needs to be able to view the outcome of the story from the "eyes" of a specific character, meaning that the player will only view the story outcome which is related to the chosen character, and will be available to interact with the story (i.e. make decisions) only when an action is related to the chosen character. We call this concept a vantage point, and the Vantage Point Manager (VPM) component to deal with vantage points. The component's implementation is depicted in Figure 84.

The VantagePointManager class uses an instance of the KnowledgeBase class and an instance of the Db class to be able to fetch information about the characters and their actions which were already executed and stored in the Knowledge Base. It also uses an instance of the LevelManager class, to fetch information about the different levels each action was executed into.

Each action is represented by an instance of the VantagePointAction class, which contains the name of the action, the time step in which the action was executed, and the level's title in which the action belongs to.

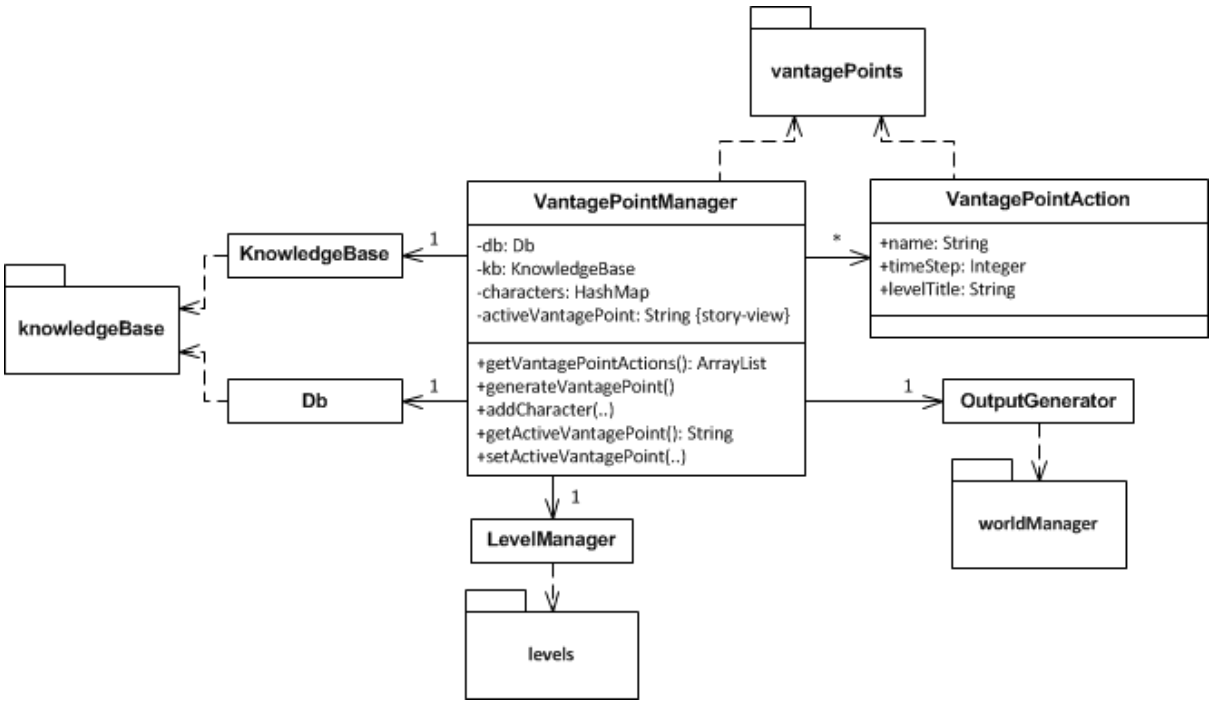

Figure 84: Vantage Point Manager class diagram

When the VantagePointManager class is requested to generate the vantage point for a new character (or revert back to the main story view), after gather all the relevant actions, it uses an instance of the OutputGenerator class to send display all the relevant information to the player (via the User Manager) as well as to the console for debugging.

5.13. User Manager

As we already discussed in section 4.13, we initially decided that based on the focus of this research, a graphical user interface would not be developed, as being out of scope for this project. Instead, the player would receive all the relevant information and the available actions in an old-fashioned text-adventure way. The first prototypes of DIEGESIS were operating in this way (Figure 85 displays an example).

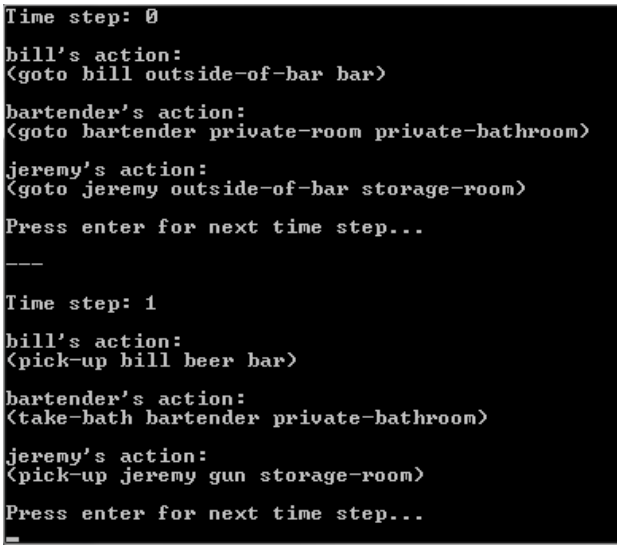

Figure 85: The initial User Interface of the system

As further discussed in section 4.13, apart from displaying the outcome of the story and relevant information about it to the player, the User Manager (UM) component is also responsible of communicating with the player. Therefore, we quickly decided that we should implement a graphical user interface (GUI). One of the first versions of DIEGESIS' GUI is illustrated at Figure 86.

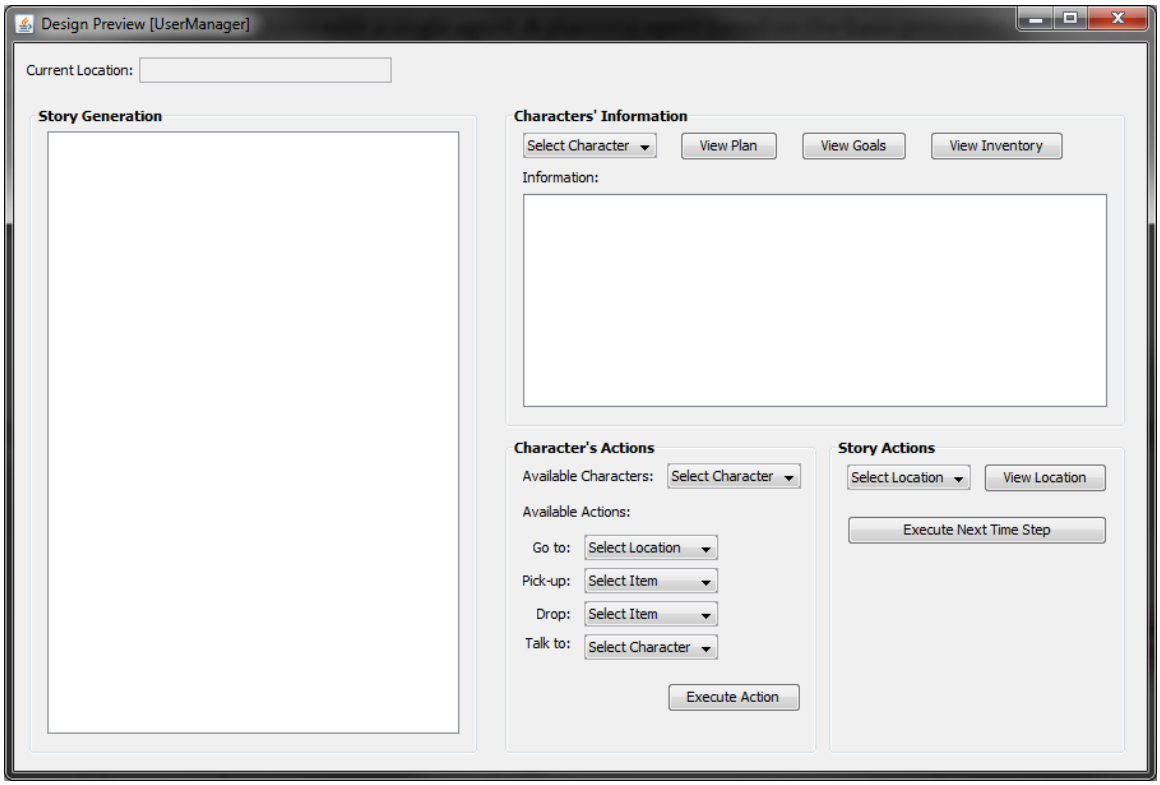

Figure 86: First version of DIEGESIS' GUI

While progressing with the design of DIEGESIS and prototyping at the same time, the design of the GUI involved, as we removed features that we decided not to implement, and added others, such as the ability for the player to make choices. An intermediate version of the GUI is displayed in Figure 87.

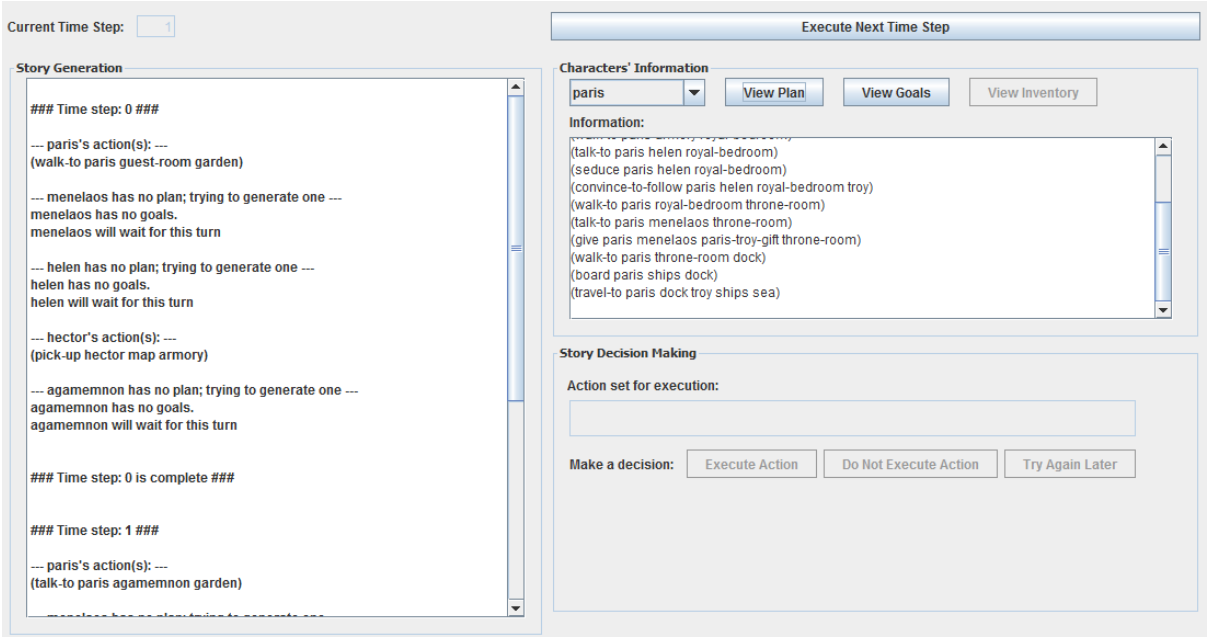

Figure 87: Intermediate version of DIEGESIS' GUI

The finalised version of the DIEGESIS' GUI is displayed in Figure 88. It includes all the features we need the player to have access to via the GUI. Apart from viewing the outcome of the story, the player can progress the story to the next turn, view at any point during the execution the current plan and goals for any of the activated characters in a level, and view information about the currently active level. The player also has the ability to switch between different vantage points at any point during the generation of the story.

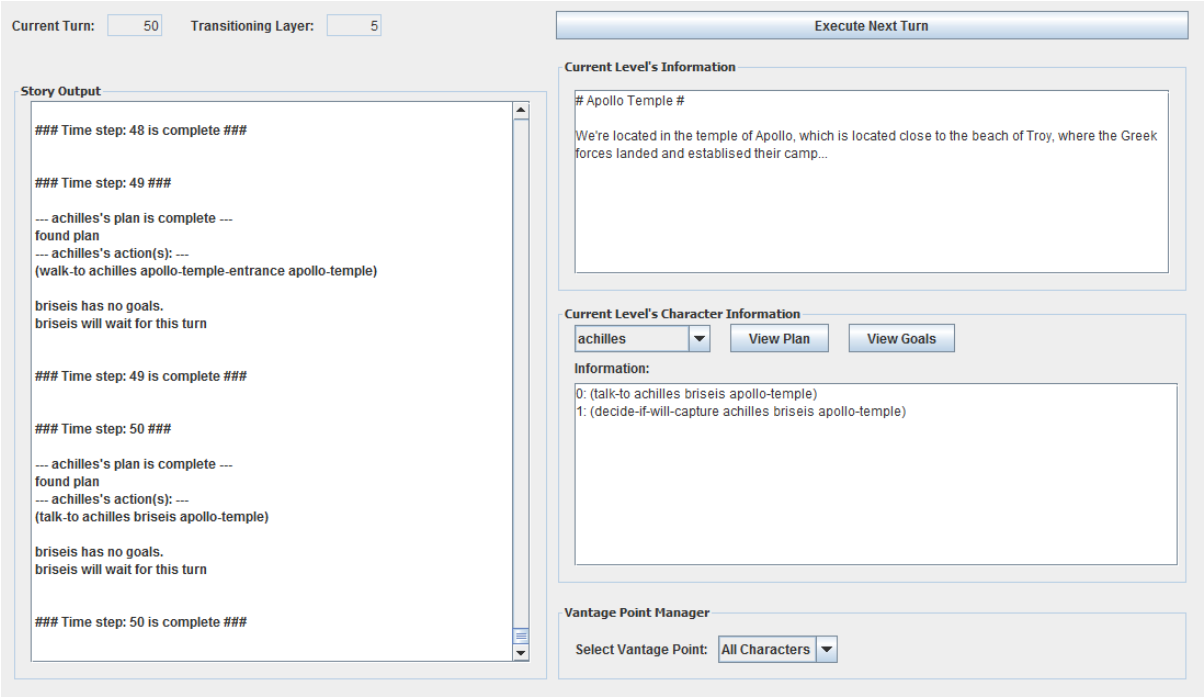

Figure 88: Finalised version of DIEGESIS' GUI

Finally, when is required, the player is able to make choices for the outcome of the story. This interactivity is removed from the main GUI, and added as a pop-up modal box, illustrated in Figure 89.

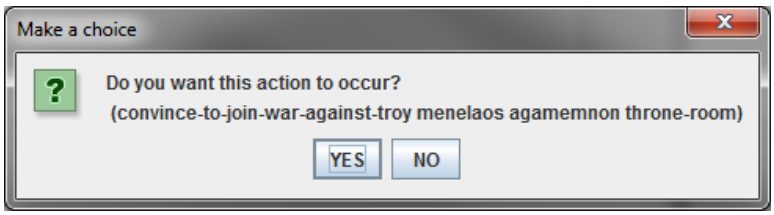

Figure 89: A pop-up modal box

The implementation of the UM component is illustrated in Figure 90. The UserManager class uses an instance of the WordManager class to be able to instruct the World Manager (WM) component about player's interactions (e.g. to advance a turn) that need to been take care by the WM.

It also uses an instance of the LevelManager class to receive information about each level (i.e. their display titles and descriptions), and an instance of the VantagePointManager class to deal with the vantage points. Finally, it have access to a list of Agent class' instances which represent story characters who are active in a level, so the LM can get information from them (e.g. their current goals, locations, etc.).

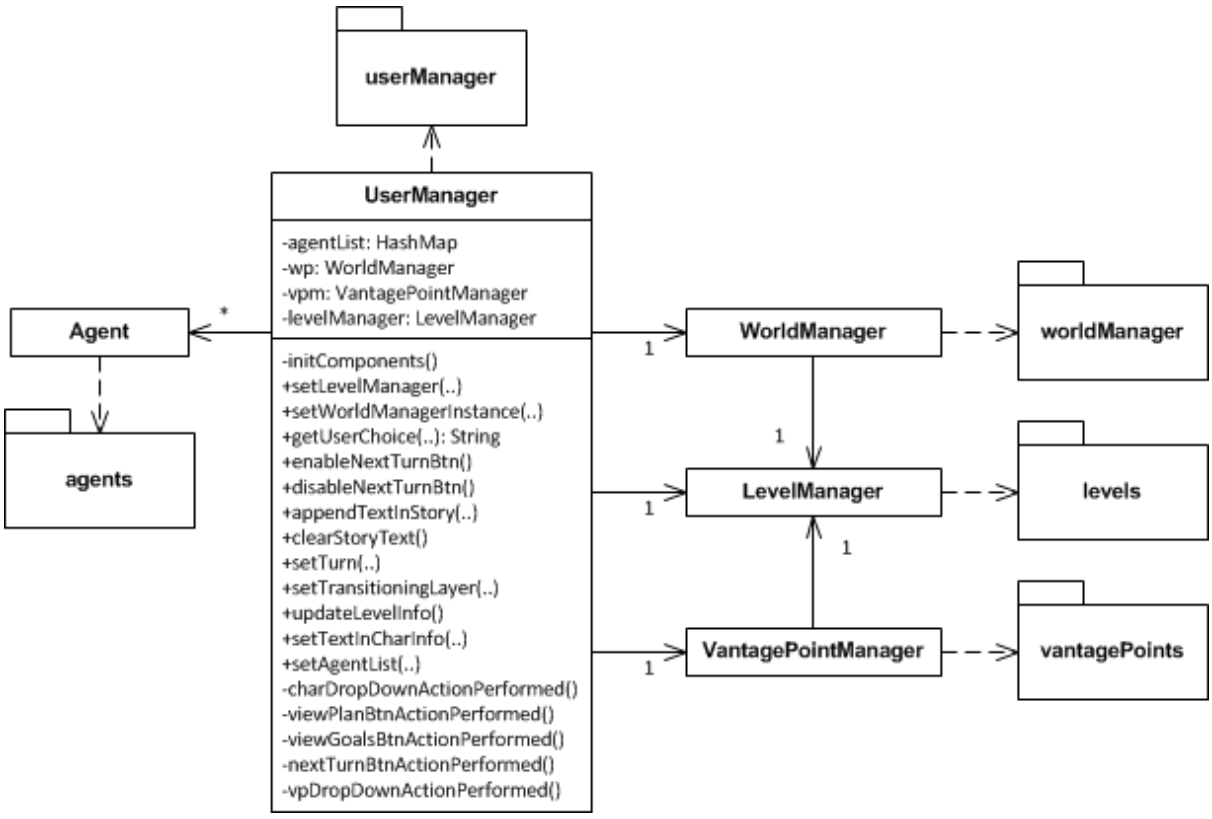

Figure 90: User Manager class diagram

5.14. PLANNER

As we discussed in section 4.14, the planner component consists of a new planning and a new re-planning algorithm, able to generate plans of actions based on each agent's state and context, considering both the current world state and the available resources. The Planner is aware of the available time (duration) a character has for a plan when it is asked to generate one.

The Planner's implementation is illustrated in Figure 91. The Planner class is the main class of the component, which deals with both the planning and re-planning processes. The InitAlgorithm class is responsible of pre-processing (i.e. initialising) the planning domain (represented by an instance of the PDDLObject class), the process which we described in depth in section 4.14. The Planner class uses an instance of the InitAlgorithm class to retrieve this information so it can use it in the planning and re-planning processes.

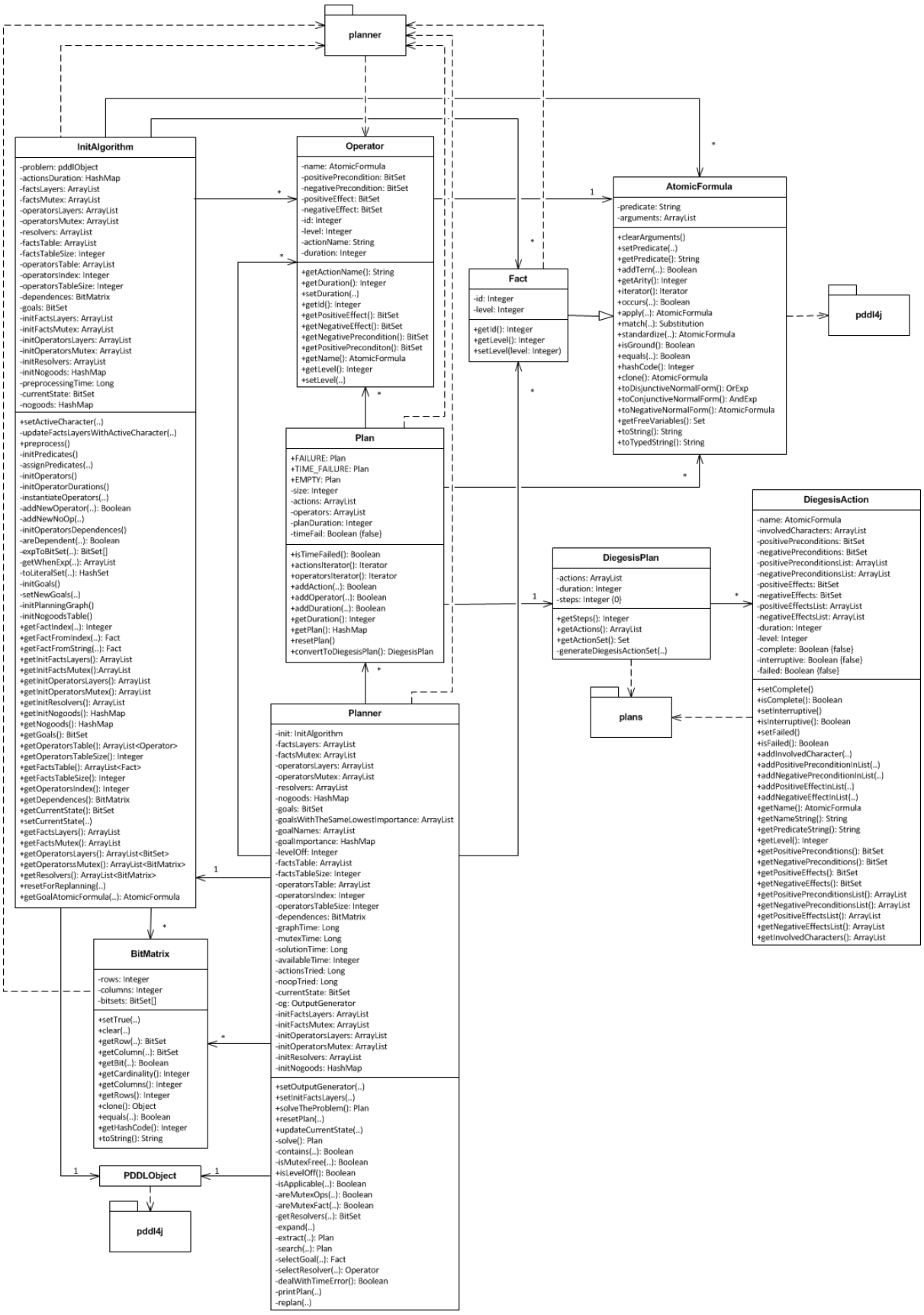

Figure 91: Planner class diagram

The AtomicFormula class is part of the pddl4j PDDL parser which we use to parse PDDL files, and represents an substituted PDDL predicate, which is a fact about the world (e.g. character X is in location Y) which can be either true or false at a given moment. The Fact class of the planner extends the AtomicFormula class and is used to represent these facts.

Each possible action in the planning domain is represented by an instance of the Operator class, and includes (among others) sets of positive and negative preconditions, as well as sets of positive and negative effects.

The planner represents several concepts as BitSets. A BitSet is a vector of bits, which can be either true (1) or false (0). For example, the planner keeps all the possible facts of the planning domain in a facts list, and keeps their state (i.e. the current state of the world) in a BitSet. There are situations where we need matrixes of BitSets, which are represented by an instance of the BitMatrix class.

Finally, a plan (either valid or failed) is represented by an instance of the Plan class. While implementing and evaluating DIEGESIS, we wanted to experiment with different concepts, for example different types of plan execution, which would require changes to the implementation of the plan. Also, we wanted the rest of the system to be as de-coupled as possible from our planner (so it can be connected to other planners in the future), therefore we decided to de-couple our planner's plans from the plans that DIEGESIS agents' use, and included the ability that the Plan could be converted to an instance of the new plan implementation.

DiegesisPlan class is the representation of the plan DIEGESIS uses. A DiegesisPlan includes a set of actions, the number of plan's steps (i.e. the number of action sets), and the total duration of the plan. Each plan's action is represented by an instance of the DiegesisAction class, which includes information about the action which will is used by the agent and the World Manager while executing the plan.

To help us with debugging during the creation of our planning solution, we visualised the planning graph, using a graphical interface. An example of the outcome of the planning graph visualisation (Shen, 2006) can be seen in Figure 92 and Figure 93.

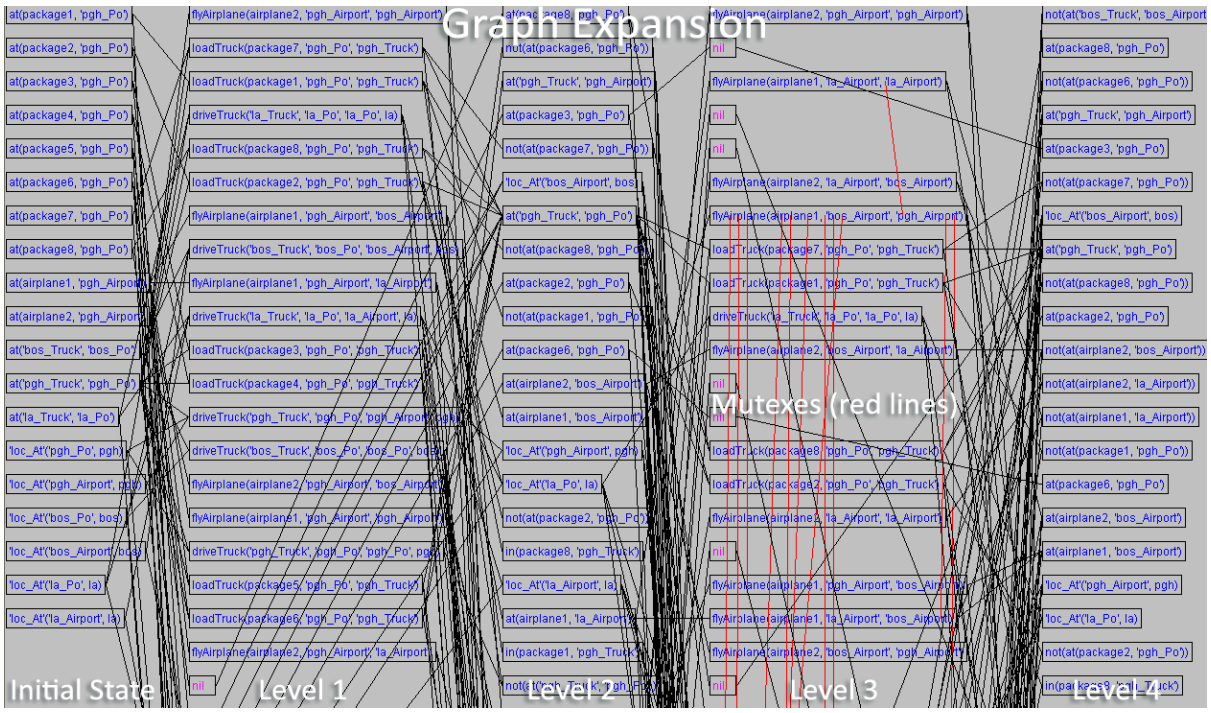

Figure 92: Planning graph expansion visualisation

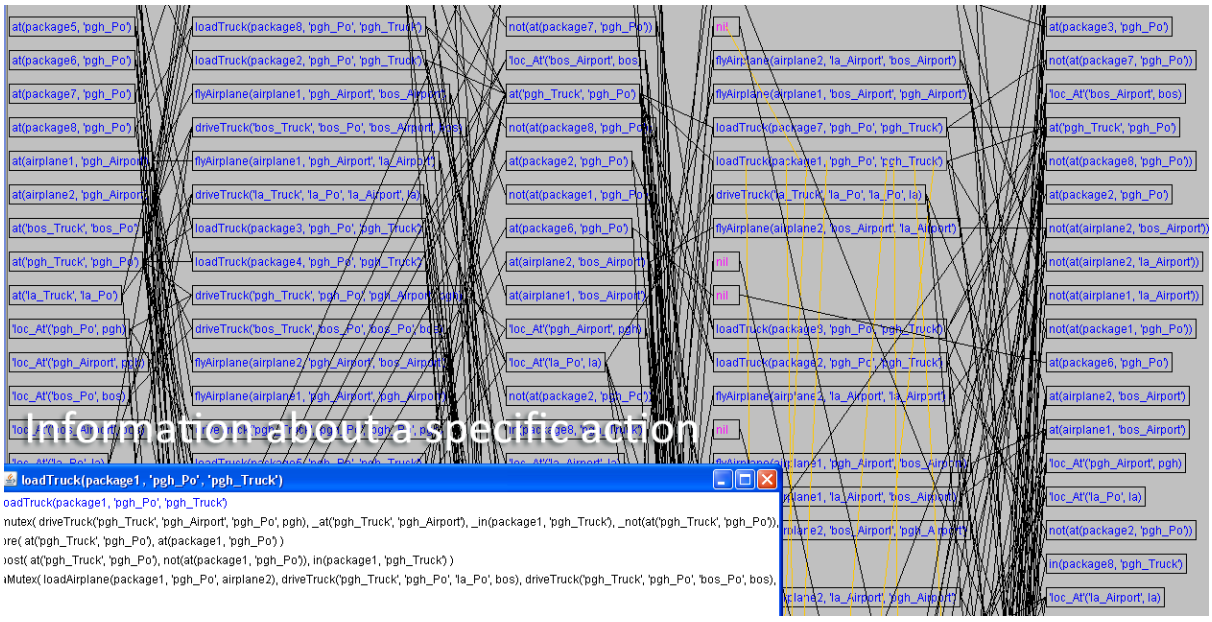

Figure 93: Information about a specific action in the visualisation of the planning graph

The planning process is presented in Algorithms 2-5. The planner receives as an input a list of goals (G), and the current state of the world (S), which is added as the first level of the planning graph. The planner also keeps the last list of planning graph levels as well as the last list of goals used in the latest planning operation for the agent who is requesting a new plan.

First, the planner checks if the new list of goals is equal to or a subset of the old list of goals. If it does, then it copies the old list of planning graph levels to the new one and searches for a level in which the goals are present into. Then, it bypasses the expansion phase, moving to the extraction phase.

Alternatively (if the above conditions are not met), the planner starts to expand the layers of the planning graph until all the goals are present in the planning graph, or the algorithm has levelled-off, considering the constraints (mutexes) between them. This is shown in Algorithm 2 – lines 10-13. As we explained in section 4.14, two actions in the same level are considered mutex when their preconditions and effects are inconsistent. If two consecutive layers are identical (the algorithm has levelled-off) and the goals are not present and mutex free in the layers, the expansion stops.

ALGORITHM 2 – Planning algorithm

Input: G: a list of goals; S: the current state of the world
Output: The generated plan P

```
1:  L: list of planning graph levels
2:  oL: old list planning graph levels
3:  oG: old list of goals

4:  copy S to L0;
5:  if (G equal to or subset of oG) AND (L0 = oL0) do:
6:      copy oL to L;
7:      search for Lx where G present in Lx;
8:      go to line 17;
9:  end if

10: while (G not present in Lx OR G not mutex-free in Lx) AND algorithm not levelled-off do:
11:         expand Lx;
12:         increment Lx;
13: end while

14: if G not present in Lx OR G not mutex-free in Lx then:
15:         return P = FAILURE;
16: end if

17: P = extract G from Lx;

18: if P = TIME_FAILURE then:
19:         while P = TIME_FAILURE AND G not empty do:
20:             select goal from G with lowest salience value;
21:             remove goal from G;
22:             search for Lx where G present in Lx;
23:             P = extract G from Lx;
24:         end while
25: end if

26: copy G to oG AND clear G;
27: copy L to oL AND clear L;

28: return P;
```

The complete extraction phase is documented in Algorithm 3. The algorithm receives as an input a list of goals (G), and the level of the planning graph (L) which needs to be expanded. As it is shown in Algorithm 3 – lines 1-9, if the algorithm is not levelled-off, then the facts (possible states of the world) of level L are retrieved, and a logical or is applied with the facts of the previous level (L-1), so the previously applied actions can be copied.

Afterwards, a check is being performed to identify which actions are applicable in L, the facts of the level L are updated with the effects of these actions, the mutexes between the actions are calculated and stored, and the resolvers (used to speed up the search process) are updated. In the case that the algorithm is levelled-off, Algorithm 3 – lines 10-15, all the information for level L is the same with the layer L-1.

ALGORITHM 3 – Expand function

Input: G: a list of goals; L: a planning graph level
Output: The data in the expanded level L

```
1:  if algorithm is not levelled-off then:
2:      get facts of L;
3:      if L > 0 then:
4:          apply logical or with facts in L-1 to copy results of previous actions;
5:      end if
6:      check which actions are applicable in L;
7:      update facts of L based on positive & negative effects of applicable actions;
8:      calculate mutexes of actions in L;
9:      update resolvers of actions in L (to speed up the search process);
10: else
11:     facts of L = facts of L-1;
12:     applicable actions in L = applicable actions in L-1;
13:     mutexes in L = mutexes in L-1;
14:     resolvers in L = resolvers in L-1;
15: end if
```

After the planning graph is created, the extraction phase begins, where a backward search is being performed starting from the last generated level (containing all the goals) until a valid plan is discovered. This is shown in Algorithms 4 and 5. The search function searches iteratively for a set of actions that resolve a set of goals G in a level L. These sets of actions form a complete plan. While searching, the total time that the plan needs to be performed is calculated (Algorithm 5 – line 21). If at some point the plan time exceeds the available time, the search stops.

ALGORITHM 4 – Extract function

Input: G: a list of goals; L: planning graph layer
Output: A plan P

```
1:  if L = 0 then:
2:      return P = EMPTY;
3:  end if

4:  P = search for G in L;

5:  return P;
```

Then, as it is shown in Algorithm 2 – lines 18-25, the planner chooses the least important goal from the list (based on the "importance" value of the goals), removes it and restarts the search process from the level that all the remaining goals are present. This happens until a valid plan within the time constraints has been discovered.

ALGORITHM 5 – Search function

Input: G: a list of goals; L: planning graph level; A: a set of actions; M: a set of mutexes
Output: A plan P

```
1:  if G not empty then:
2:      select goal from G;
3:      get actions that resolve goal;
4:      while actions not empty AND P = NULL do:
5:          select action from actions;
6:          remove action from actions;
7:          select mutexes for action;
8:          newA = A + action;
9:          newM = M + mutexes;
10:         newG = G – positive effects of action;
11:         P = search for newA, newM, newG, in l;
12:     end while
13: else
14:     for each A as action do:
15:         add action's positive precondition in G;
16:     end for each
17:     P = extract for G in L-1;
18:     if P is not FAILURE OR TIME_FAILURE then:
19:         for each A as action do:
20:             add action to P in layer L-1;
21:             calculate duration of P;
22:             if duration of P > available duration for P then:
23:                 P = TIME_FAILURE;
24:             end if
25:         end for each
26:     end if
27: end if

28: P = search for G in L;

29: return P;
```

As discussed in section 4.14, our proposal for re-planning in DIS interleaves plan generation and plan execution to be able to re-plan as soon as an unexpected change happen in the environment, while making minimal disruption to the original plan.

Our re-planning algorithm is illustrated in Algorithm 6. The first step is to get the set of actions related to the step of the plan which failed. The effects of each action which is either failed or not executed yet are added as the goals of the new plan. If there are more steps in the plan, then the effects of the actions of the next step are also added in the goals list.

ALGORITHM 6 – Re-planning algorithm

Input: P: an existing plan; S: step of the plan which failed;
Output: A new plan P

```
1:  A: set of actions; p: partial plan; G: goals list;

2:  get A from P for S;
3:  for each A as action do:
4:      if action is failed OR action is not complete then:
5:          get effects of action;
6:          add effects in G;
7:      end if
8:  end for each

9:  if size of P > S then:
10:     get A from P for S + 1;
11:     for each A as action do:
12:         get effects of action;
13:         add effects in G;
14:     end for each
15: end if

16: p = generate plan for G;
17: if size of P > S then:
18:     remove actions of S + 1 from P;
19: end if
20: remove completed and failed actions from P;
21: add actions of p at the beginning of P's actions list;

22: return P;
```

Afterwards, a new partial plan is generated for these goals via our planning algorithm. After the generation of the new partial plan, the completed and failed actions (as well as the actions of the step right after the failed step, if such a step exists) of the existing plan are removed, and the two plans (the existing and the new partial plan) are merged into one, which is returned to the agent for execution.

5.15. AGENT

As we already discussed in section 4.15, each agent in DIEGESIS represents a character in the game world and is designed to operate as an individual, creating an autonomous plan considering its own needs. The implementation of the Agent component is displayed in Figure 94.

The Agent class is able to keep all the information about the story world that an agent is aware of, as well as information about the character that the agent represents. An instance of the Db class is being used to fetch information about the character, as well as to update such information (e.g. if the character dies). To be able to display relevant information to the player, as well as for debugging reasons, the Agent class uses an instance of the OutputGenerator.

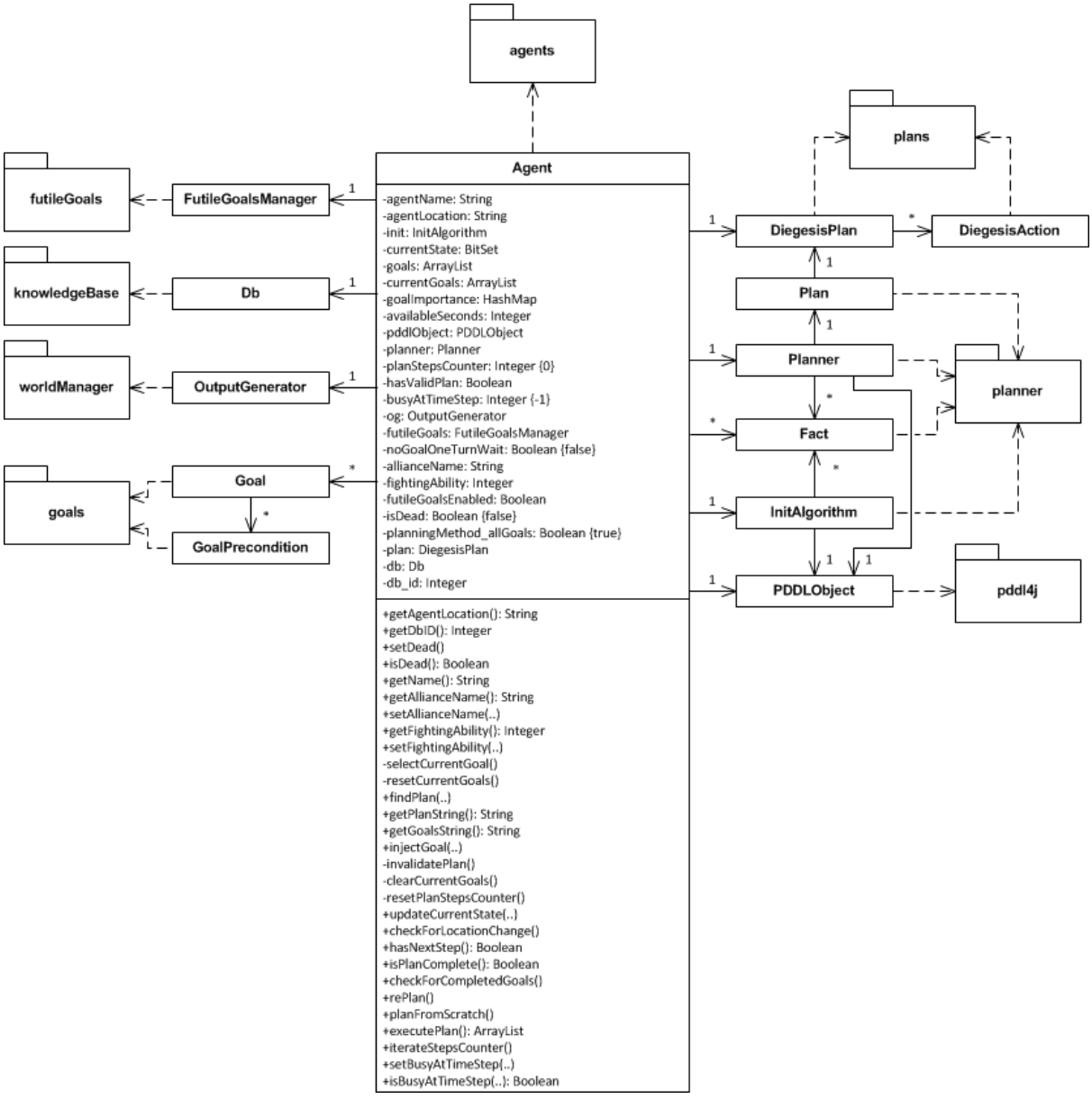

Figure 94: Agent class diagram

The Agent class may include several instances of the Goal class (each of them with its own set of GoalPreconditions), which represent a set of goals that a character has for a level. In case there are no goals in the list, the agent is set to accept futile goals, and the agent already waited for one turn, then it is able to requests a futile goal from an

instance of the FutileGoalsManager class. The agent is able to check the status of its goals at any time, and to re-populate its list of current goals (i.e. the list of goals to search for a plan for).

When an agent has at least a goal whose precondition is valid, then it needs to find a plan to accomplish it. To find a plan, the Agent class uses an instance of the Planner class. The Planner class is fed with an instance of the InitAlgorithm plan (i.e. the initialised algorithm based on the current status of the world – the PDDLObject class' instance), as well as a set of goals, and is requested to generate a plan for them. The plan which is returned by the Planner is represented by an instance of the Plan class, and it is then converted to an instance of the DiegesisPlan class, which contains several instances of DiegesisAction class, which represent different actions which the agent need to execute to accomplish the plan.

The Agent class is able to keep track of which parts of the plan are already executed and if the plan is complete. During the execution of a plan, the agent might be informed by the World Manager that an action set has failed. If that happens, then the Agent class sends all the relevant information to the Planner class, requesting the Planner to re-plan and provide a new valid plan based on the provided information and current state of the world.

5.16. Battle Manager

As we discussed in section 4.16, there are certain cases in the story that we are using to evaluate DIEGESIS, in which we need battles to occur. The Battle Manager (BM) component is responsible of dealing with all the battles in the system. The implementation of the BM is illustrated in Figure 95.

The BattleManager class uses an instance of the Parser class as well as an instance of the LevelManager class to parse and load the battle information for a new level. An instance of the Db class is used to store and retrieve battle-related information from the Knowledge Base.

It also uses an instance of the OutputGenerator to display messages to the player and to the console for debugging, and may use instances of the Agent class to retrieve

battle-related information from any agents involved in a battle (either versus another agent or versus a soldier, which is a NPC).

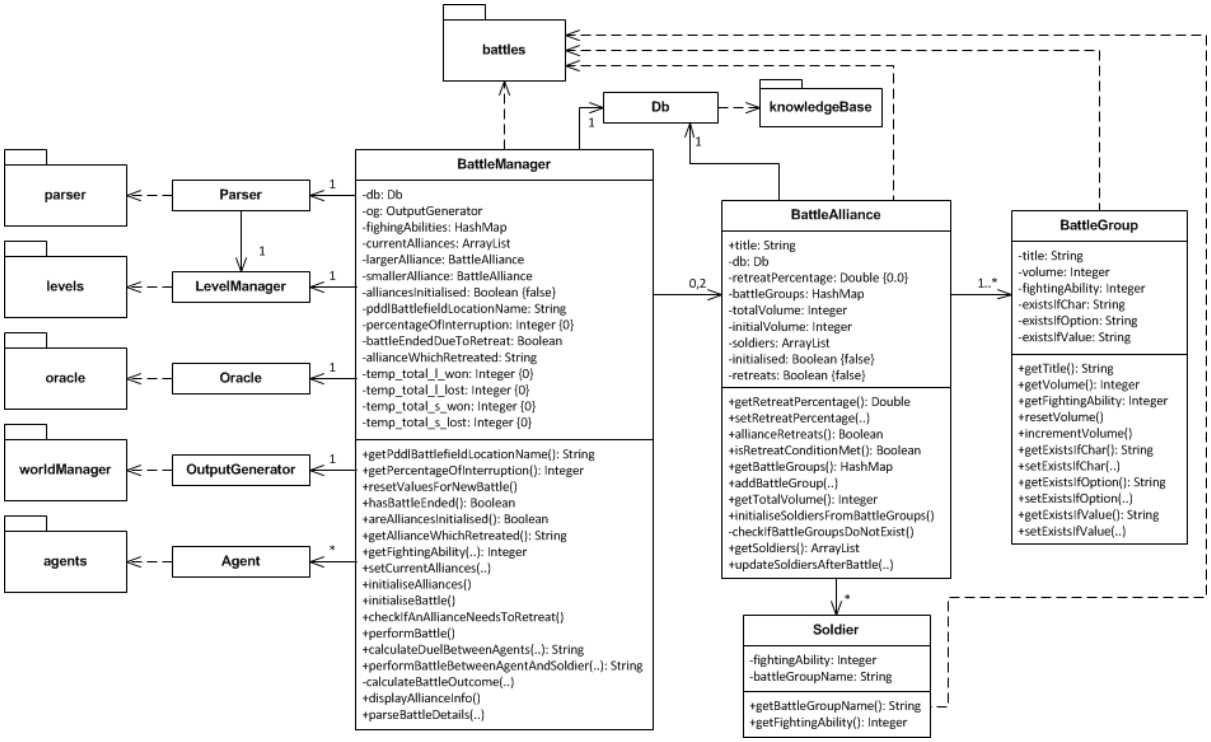

Figure 95: Battle Manager class diagram

As explained in section 4.16, in each battle level there can be up to two alliances fighting against each other. These alliances are represented by an instance of the BattleAlliance class. Each alliance may consist of different battle groups, each possessing a different volume of soldiers. Battle groups are represented by instances of the BattleGroup class, while soldiers are represented by instances of the Soldier class. To calculate the outcome of a battle, the BattleManager class uses an instance of the Oracle class.

5.17. Evaluation Monitor

As we discussed in section 4.17, we need to be able to evaluate aspects of the whole framework and the generated narrative while the story is being executed and generated, therefore we designed the Evaluation Monitor component. Its implementation is illustrated in Figure 96.

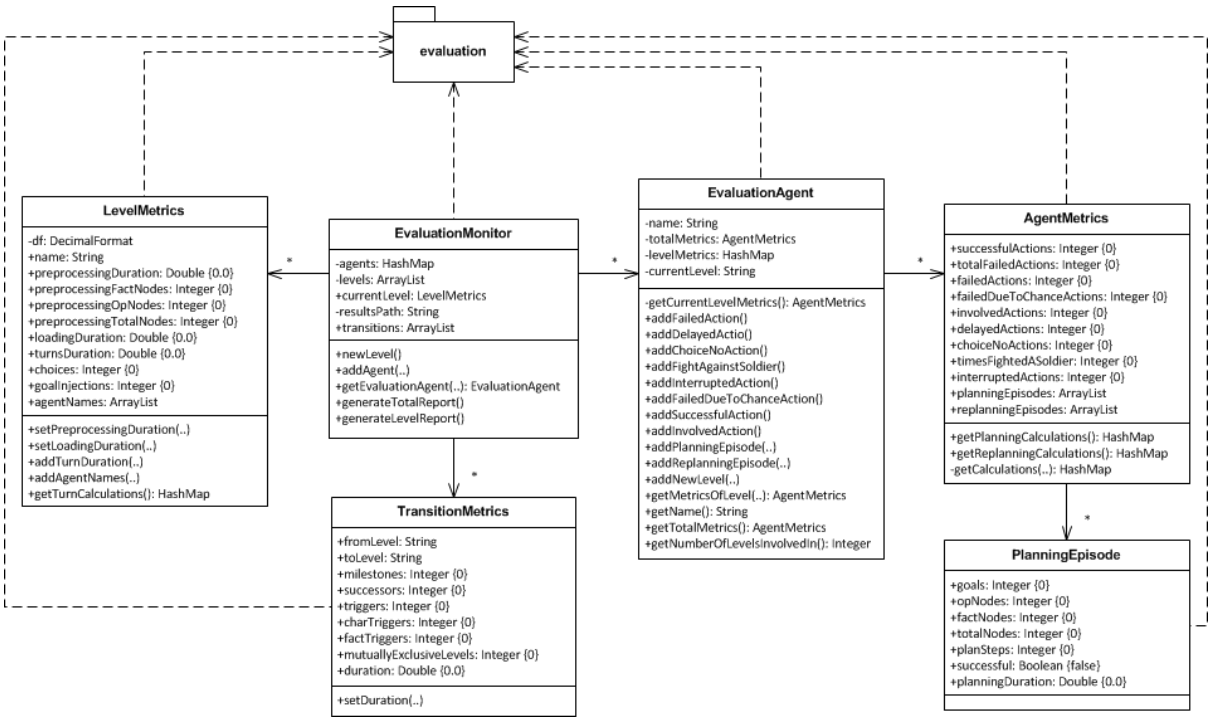

Figure 96: Evaluation Monitor class diagram

The EvaluationAgent class combines all the metrics relevant to an agent (i.e. instances of the AgentMetrics and PlanningEpisode classes) for each level the agent participates in. The LevelMetrics and TransitionMetrics classes track the level and transition metrics respectively. The EvaluationMonitor class keeps track of everything, and when requested can produce the two types of report discussed in section 4.17 and save them to text files.

In this chapter, we documented all the details about the implementation of the multi-agent DIS framework we designed in the previous chapter. In the next chapter, we will discuss about our evaluation methodology, and we will document a number of evaluations for the different components of our framework

6 EVALUATION

In this chapter, we provide detailed information about the evaluation scenario that we modelled, showing its potential storylines, we also discuss some of the mechanics that can have an impact on the generated story, and we specify the metrics used in our evaluations. Then, we are documenting a number of evaluations for the different components of our framework, using the evaluation scenario we presented earlier in the chapter.

6.1. EVALUATION SCENARIOS

As we discussed in section 3.4, to aid us to decide which planning algorithm to use as a base for our solution, we needed to perform an evaluation of planning algorithms with a DIS perspective in mind. To benchmark the performance of the algorithms, we used an approach discussed in (Barros and Musse, 2007b), therefore we decided to use the same scenario used in their experiment to be able to compare our results to the published results. The evaluation scenario is called Ugh's Story first act, and is described in (Barros and Musse, 2007b).

For the evaluation scenario of DIEGESIS though, we needed a story much bigger than the Ugh's Story scenario, which would be rich in relations between characters, actions, choices, possible constraints and outcomes, making it a suitable setup for rich emerging narratives to be developed in a DIS system.

After investigating some potential candidate stories, we decided to use a scenario based on the story presented in the 2004 film "Troy" (Petersen, 2004), which is based on Homer's epic poem called "Iliad" (Homer, 2003).

In the scenario we modelled, *Paris*, the Prince of *Troy* (brother of *Hector*; son of *Priam*) falls in love with *Helen*, the Queen of *Sparta*, seduces her and convinces her to join him and flee together to *Troy*. *Menelaos*, the King of *Sparta* and husband of *Helen*, along with his brother *Agamemnon* (King of *Mycenae*) decide to gather an army comprising of the elite of *Greek* warriors (such as *Achilles* and Odysseus) to attack and conquer

Troy, each of them for their own reason. *Menelaos* because he's angered that *Helen* left him and *Agamemnon* because he always wanted to find a reason to conquer *Troy*.

As we will present in the next section, we believe that the Troy scenario is a suitable setup for our needs, since it contains several characters with rich relations between them, and a high number of possible actions and choices, which can provide different outcomes.

6.2. Potential Storylines

As we already explained in section 4.1, the game world is created before the execution of a story, by a storyteller and is organised in multiple levels which can represent possible parts of a story (typically a broad area where a number of events in a story may occur).

We modelled the Troy scenario in the following 27 levels. The potential transitions between them are illustrated in Figure 97.

1. *Sparta palace*: In the initial level, the scene takes place in the palace of *Sparta*, where *Paris* and *Hector* (the Princes of *Troy*) are visiting *Menelaos* (the King of *Sparta*) to negotiate a peace treaty between *Sparta* and *Troy*. While this negotiation takes place, *Paris* seduces *Helen* (the Queen of *Sparta*) and tries to convince her to flee with him to *Troy*.
2. *Troy palace*: In this level, the *Trojan* princes have returned from their visit to *Sparta* and *Hector* is informing *Priam*, his father and king of *Troy*, of the negotiation outcome. If *Paris* succeeded in his seduction and *Helen* is with him, *Priam* and *Hector* are trying to convince them to return Helen back to *Sparta* to prevent the upcoming war resulting of that action.
3. *Mycenae palace*: Given that *Helen* fled with *Paris*, Menelaos visits his brother Agamemnon (King of *Mycenae*) to ask him to join the war against Troy.
4. *Sparta palace (peace)*: If the negotiation between *Troy* and *Sparta* succeeded and *Paris* did not convince *Helen* to flee with him, then there are no grounds for a war between *Sparta* and *Troy*, and the story would end there. To still provide a chance for the war to occur, we modelled this level where *Agamemnon* (who is depicted in the movie as a warmonger who always

wanted to capture *Troy*) decides that he might declare war against *Troy* and visits *Menelaos* in Sparta to try and convince him to join the war. If *Agamemnon* fails to convince *Menelaos* to join the war, he has to decide if he will still declare war by himself. If he decides against that, then the story ends here.

5. *Ithaca*: This level is located in *Ithaca*, where *Menelaos* and *Agamemnon* visit *Odysseus* (the King of *Ithaca*) to convince him and his army to join the war against *Troy*.
6. *Fthia*: This level is located in *Fthia*, the hometown of one of the greatest Greek warriors called *Achilles*, where *Odysseus*, *Menelaos*, and *Agamemon* (the latter only as a backup plan in case that both *Odysseus* or *Menelaos* are not part of the war) are visiting *Achilles* to convince him and his army of *Myrmidons* to join the war against *Troy*. In the event they fail to do so, they try to convince *Patroclus* (*Achilles'* cousin and best friend) to persuade *Achilles* and change his mind.
7. *Sparta palace (Helen returns)*: This is a level resulting from level 2 (*Troy* palace), where *Helen* was convinced to return back to *Sparta*, and tries to convince *Menelaos* (and *Agamemnon* if he joined the war) to call of the war. If she succeeds, the story ends there.
8. *Troy beach Myrmidons landing*: In this level, the *Greek* army is approaching a beach in the outskirts of *Troy*, and the ship containing *Achilles*, *Patroclus*, and the rest of the *Myrmidons* are landing first, and fight a small army of *Trojans* who are there to protect the beach and the nearby temple of the *Greek* and *Roman* god of light and sun, called *Apollo*. If they win the fight, then the rest of the *Greek* army lands without a further fight, and *Achilles* has to decide if he will visit the temple or not.
9. *Troy beach landing*: In case *Myrmidons* lost the fight of the previously described level or *Achilles* decided in level 6 (*Fthia*) that he won't join the war, then the whole of the *Greek* army lands to the *Trojan* beach and fights the previously mentioned small army of *Trojans*. If they win the fight, they establish a *Greek* camp in the beach. If *Achilles* is present, he still has to decide if he will visit *Apollo's* temple. If they lose the fight, they return back to *Greece* and the story ends.

10. *Apollo temple*: In this level, *Achilles* explores the temple of *Apollo*, located closely to the *Greek* camp. If he decides to plunder the temple, he discovers *Briseis* (priestess of *Apollo*; *Hector* and *Paris'* cousin) and decides if he will capture her and bring her back to the *Greek* camp.
11. *Greek camp (Briseis)*: When *Achilles* brings *Briseis* to the *Greek* camp she is taken from him by *Agamemnon*, and given to some *Greek* soldiers. *Achilles* confronts *Agamemnon*, and asks him to return *Briseis*. If *Agamemnon* refuses, then *Achilles* decides that he and his *Myrmidons* will no longer fight in the war, then finds the soldiers who have *Briseis* and takes her back.
12. *Troy palace gardens*: While Troy is under siege by the *Greek* forces, *Helen* (considering she's still in *Troy*) decides to leave *Troy*, head to the *Greek* camp and try to convince *Menelaos* to stop the war. While trying to leave, *Hector* stumbles upon her, and tries to talk her out of her decision.
13. *Greek camp (Helen returns)*: If *Hector* failed to convince *Helen* to leave *Troy* in the previously described level, *Helen* finds *Menelaos* and Tries to convince him to stop the war. If she succeeds, then *Menelaos* is trying to convince *Agamemnon* (considering he's part of the war) as well. If everyone is convinced, the *Greek* forces withdraw, and the story ends.
14. *Troy palace (Paris wants duel)*: Considering that *Helen* is still in Troy, *Paris* tries to convince his father (*Priam*) and his brother (*Hector*) to allow him to challenge *Menelaos* to a duel to prevent a further conflict between the *Greek* and *Trojan* forces.
15. *Troy battlefield (Paris vs Menelaos duel)*: Just before the first large-scale battle in the battlefield just outside of *Troy*, the leaders of each side are discussing. Initially, *Agamemnon* (if he's present in the war) proposes to *Hector* to surrender. If *Hector* agrees, the *Greeks* are capturing *Troy* and the story ends. Alternatively, considering that *Paris* duel proposal discussed in the previous level is valid, *Paris* proposes to duel *Menelaos*, while *Agamemnon* is trying to convince *Menelaos* not to duel. In case of a duel, if *Paris* wins he kills *Menelaos*. If *Menelaos* wins, *Paris* has to decide if he will stay and die in honour or not. If he doesn't stay to die, then there is a chance that a further duel between *Hector* and *Menelaos* will take place (if *Menelaos* tries to pursue him).

16. *Troy battlefield (first battle)*: After any duels have occurred in previous levels, the first large-scale battle between the *Greek* and *Trojan* forces starts, and everyone from the *Greek* forces who is present at the war takes part of it, except from *Achilles* and his *Myrmidons* if he previously decided to withdraw from the war due to his conflict with *Agamemnon* over *Briseis*. If *Greeks* win the battle, they capture *Troy* and the story ends. If not, they retreat back to the *Greek* camp to regroup.
17. *Greek camp (Achilles won't fight)*: If *Achilles* decided to withdraw from the war, then *Odysseus* (considering he is part of the war and still alive), *Patroclus*, or *Eudorus* (another *Myrmidon*, in case both of the others are unavailable) try to convince him to join the war again. If they fail, *Patroclus* (who wants to fight) has to decide if he will impersonate *Achilles* leading the *Myrmidons* in the next battle.
18. *Troy palace (retaliation)*: *Priam* tries to decide if the *Trojans* will attack the *Greeks* in their camp, or if they will wait for them to attack again. *Hector* wants to wait, while the *Trojan* high-priest wants to attack since he believes *Apollo* will favour *Troy* because *Achilles* decided to plunder the temple in a previous level. If *Achilles* did not plunder the temple, then the high-priest provides no opinion.
19. *Troy battlefield (final battle)*: In this level, the *Trojans* decided to wait for the *Greeks* to attack. The *Greeks* gather their whole army, and perform a final attack which will decide the outcome of the war: if they win they capture *Troy*; if they lose they withdraw and return back to *Greece*. Either way, the story ends.
20. *Greek camp (Trojans attack)*: In this level, the *Trojans* decided to attack the *Greeks* in their camp. If they win the battle, the Greeks withdraw and the story ends. If they lose, then withdraw back to *Troy* to regroup. If *Hector* and *Achilles* are both part of this battle, *Hector* will try to locate and duel him. In case that *Patroclus* decided to impersonate *Achilles*, he will be part of this duel instead.
21. *Greek camp (Priam wants ceasefire)*: In case either *Paris* or *Hector* were killed in the aforementioned battle at the *Greek* camp, *Priam* will send a messenger to the *Greek* camp asking for a twelve-day ceasefire to mourn and honour the memory of the princes. In the meantime, he performs a funeral for them.

22. *Greek camp (Patroclus died)*: In case that *Patroclus* impersonated Achilles and was killed by *Hector*, *Eudorus* informs *Achilles* of what happened. Then, *Achilles* has to decide if he will resume fighting for the *Greek* forces and if he will seek revenge for *Patroclus'* death. If he decides in favour of revenge, *Briseis* (considering she is part of the story) will try to convince him against it.
23. *Troy battlefield (Achilles vs Hector duel)*: *Achilles* wants to take revenge for *Patroclus'* death by the hands of *Hector*, so he goes alone outside of *Troy*, and challenges *Hector* to a duel. If *Hector* agrees, he says his goodbyes to his father, his brother, and his wife *Andromache* (informing her of a secret exit from *Troy* in case it is captured by the *Greeks*) and confronts *Achilles*. After the duel, if *Achilles* is killed *Hector* takes his body back to the *Greek* camp. If *Achilles* manages to kill *Hector*, he has to decide if he will leave his body, or take it with him to further disgrace him. At the end of the level, *Achilles* attends *Patroclus'* funeral.
24. *Greek camp (Priam asks for Hector's body)*: In case that *Achilles* killed *Hector* and took his body with him, *Priam* disguises himself and infiltrates the *Greek* camp. There, he locates *Achilles* and tries to convince him to allow *Priam* to retrieve *Hector's* body to give him a proper funeral. If *Briseis* is present, she tries to persuade *Achilles* over that direction as well. If *Achilles* agrees, *Priam* takes *Hector's* body back to *Troy* where every *Trojan* character attends his funeral. Also, if *Briseis* wants it, *Achilles* has to decide if he will allow her to return back to Troy as well.
25. *Greek camp (attack or Trojan horse)*: If *Odysseus* is still present in the war, he comes up with the idea of the *Trojan horse*: to build a huge hollow wooden horse and fill it with himself and other *Greek* fighters to be able to infiltrate *Troy* while the rest of the *Greek* army would seemingly depart from the war. If the *Greek* leaders decide against it, they will just perform a final battle instead.
26. *Troy beach (Trojan horse)*: *Greeks* pretend that they withdrew from the war, and left behind the *Trojan Horse Odysseus* invented. A *Trojan* scout brings the news to *Priam*, and *Trojans* need to decide if they will accept it or not. *Hector* and *Paris* (if they are still present) are against the idea of accepting the "gift", while the *Trojan* high-priest is in favour of the idea if the *Apollo* temple was previously plundered. If *Priam* decides against accepting the *Trojan horse*, he

gives the order to torch it, therefore killing everyone inside it, causing the *Greeks* to lose the war and the story to end.

27. *Inside Troy*: In this level, the *Trojans* accepted the *Trojan horse*, and placed it inside the walls of *Troy*, near its main entrance. When the time is right, the *Greek* fighters emerge from it, the open the entrance for the rest of the army to enter *Troy*, and they capture and destroy the city. *Agamemnon* heads to the *Trojan* throne room, where he finds *Priam* and tries to kill him. If *Briseis* ended up in *Troy*, is in the throne room as well, and is captured. If *Agamemnon* manages to kill *Priam*, *Briseis* tries to kill him via a sneak attack. If she's captured, *Achilles* tries to find her and save her. In the meantime, if *Hector* is still part of the story, he informs *Andromache* of the secret exit from *Troy* and sends her there to flee. *Andromache* also tries to find *Helen* and informs her of the secret exit as well. In case *Menelaos* is still part of the story tries to locate *Paris* and duel him. *Helen* tries to convince *Paris* to flee with him from *Troy*. Even if she succeeds or not, she will try to flee. In some cases, *Paris* and/or *Hector* will try to duel and kill *Achilles*. If *Briseis* is present, she will try to prevent them from doing so. This is the last level of the story, and after it's complete, the story ends.

Apart from the aforementioned levels, during the course of our research we created some other levels as well, either modified versions of the above to be used in evaluations, or modelling scenes from the Troy film which we ended-up not using in the finalised scenario for various reasons. As an example, one of these levels included *Achilles* meeting the *Greek* goddess *Athena* in the Pantheon, to take advice about going to war against *Troy*.

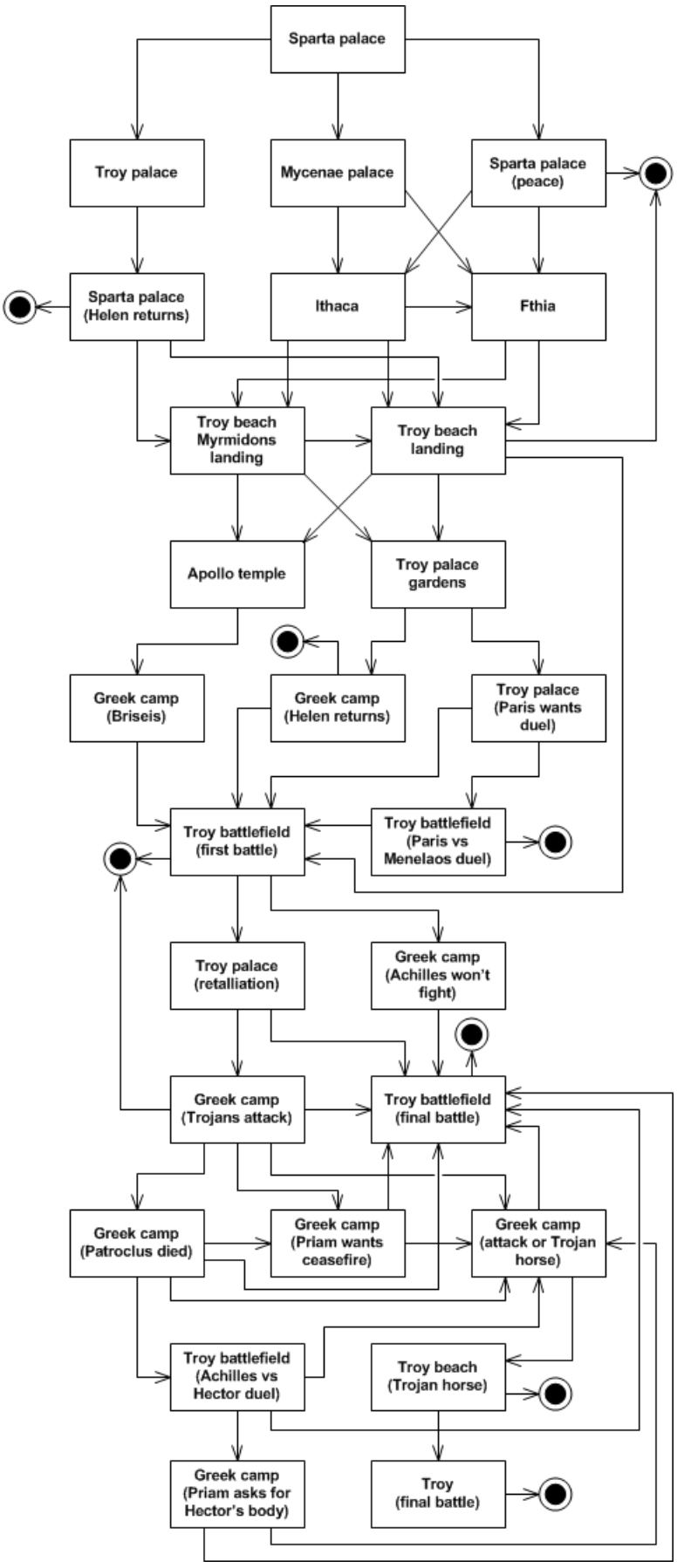

Figure 97: Troy scenario levels and potential transitions

6.3. Story Mechanics

As we explained in previous chapters, DIEGESIS implements different mechanics which have the potential to affect the agent's plans, thus the outcome of the story. First of all, based on actions that may occur during the generation of the story, some goals might be injected to agents (discussed in section 4.8), altering their behaviour in a level, and potentially affecting other agents as well. For example, in level 15 (*Paris* vs *Menelaos* duel) considering that *Paris* loses the duel and does not stay to die in honour, a goal is injected to *Hector* to protect him by duelling *Menelaos*.

The above can have serious implications to the story, since some of the main characters can die. This is caused by the implementation of both duels and battles. Duels (discussed in section 4.5) can occur between two characters, their outcome is decided by the Oracle (discussed in section 4.10), but the death of the character who lost the duel is not definite; after the end of a duel, there are goal injections which dictate if a character will be killed or not.

Battles (discussed in section 4.16) are large-scale, between two alliances, each may containing several battle groups consisting of thousands of soldiers. Soldiers are NPCs and their only purpose is to fight. As in duels, the outcome of a battle between a soldier and another soldier or a character is determined by the Oracle. Apart of the death of a character during a battle which can have serious impact on the story (we will discuss the situation where a character is no longer part of the story in a bit), the actual outcome of a large-scale battle can be a game-changer as well. For example, if the *Greek* forces lose any of the battles in the landing level, the final battle level, or the level where the *Trojans* attack the *Greek* camp, they will lose the war, and the story will end.

The most powerful mechanic though, that may occur in any level and most of the characters have the opportunity to use it while the story progresses, are the choices (discussed in section 4.6) made either by the player or the system. As an example, considering that in *Fthia* level *Achilles* decided against joining the war, 7 of the future levels will not be possible to execute since *Achilles* was an integral part of them, and further 5 levels can be executed, but they will be affected since *Achilles* will not be there. This situation is illustrated in Figure 98.

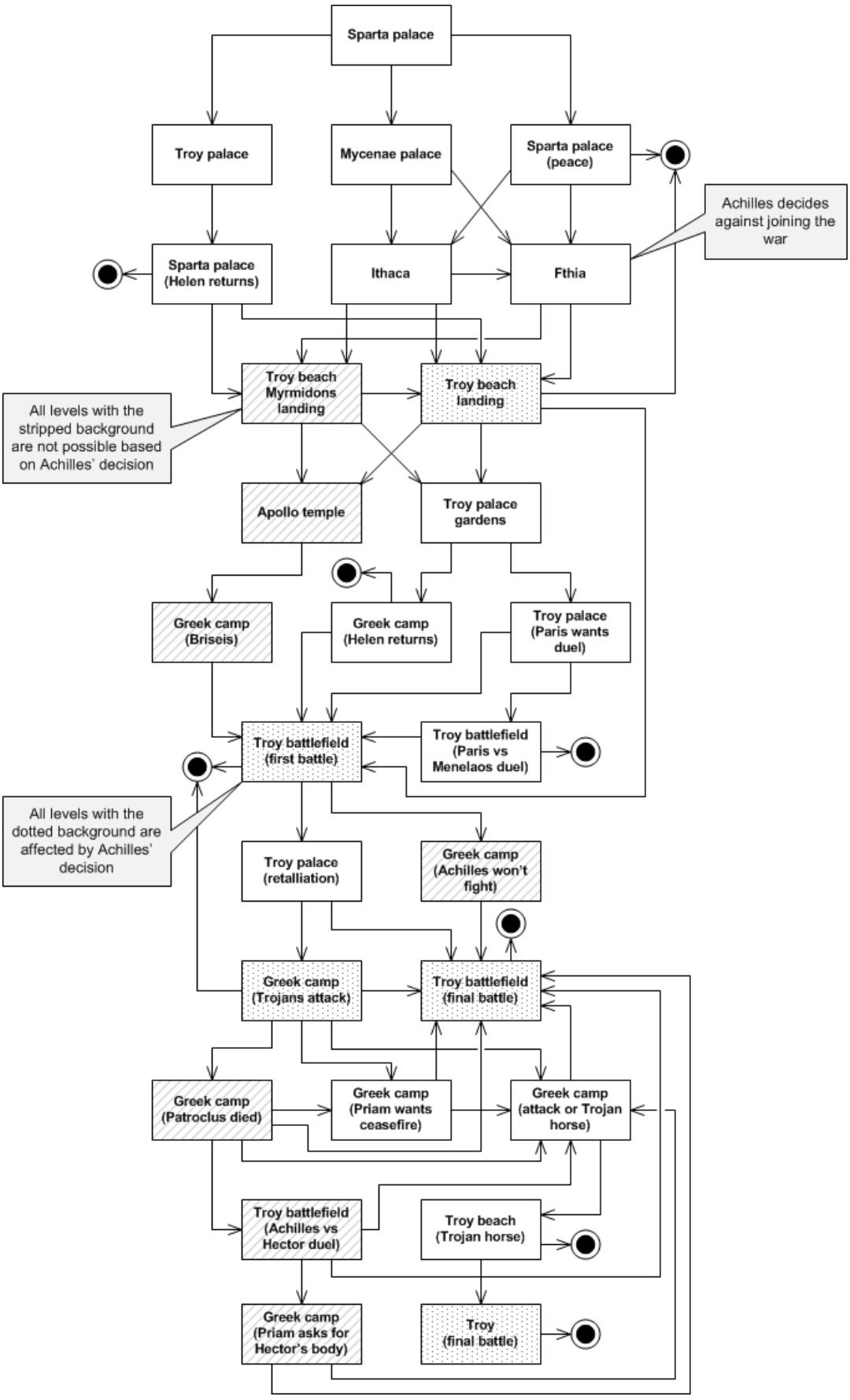

Figure 98: Levels affected if Achilles decides not to join the war

Similar to that choice, if *Achilles* does go to war but dies in the *Myrmidons* landing, it will have the almost the same effect in the story. The levels that *Achilles* is an integral part will not be executed, and the rest will be affected by *Achilles* not being there, but the difference is that the rest of the *Myrmidons* will be there and they can choose to fight and be led by another *Myrmidon* instead.

Another choice example which can have a huge impact on the story is if *Helen* decided against fleeing with *Paris* in the very first level of the story. This single choice can cause 6 levels not to be able to be executed, and one to be affected. This situation is illustrated in Figure 99.

Choices don't have to have such dramatic effects to the story though; they can just flavour it a bit. For example, if *Achilles* decides to plunder *Apollo's* temple, the *Trojan* high-priest will be in favour of attacking the *Greeks* to their camp, as well as accepting the *Trojan horse* and will try to persuade *Priam* towards that direction.

A couple of other mechanics which can potentially have some impact in the outcome of the story are the futile goals (discussed in section 4.9) and the uncertain actions (discussed in section 4.11). An uncertain action has the potential to delay the execution of a character's plan if it fails, but the character will try to execute it again in a later turn.

In the case of futile goals, the most common impact is that they can delay a character's actions when they involve another character who is executing a plan for a futile goal. For example, in the first level *Helen* has no initial goals, and one of the first goals of *Paris* is to be involved in a discussion with her, which can be delayed to be executed if *Helen* is busy executing a futile goal.

There is a final mechanism which although does not have any impact on the actual generation of the story outcome, it has an impact on the perception of the story by the player. When the vantage point (discussed in section 4.12) of a character is selected, the story will continue to be generated normally, but the player will only view content which is relevant to the selected character.

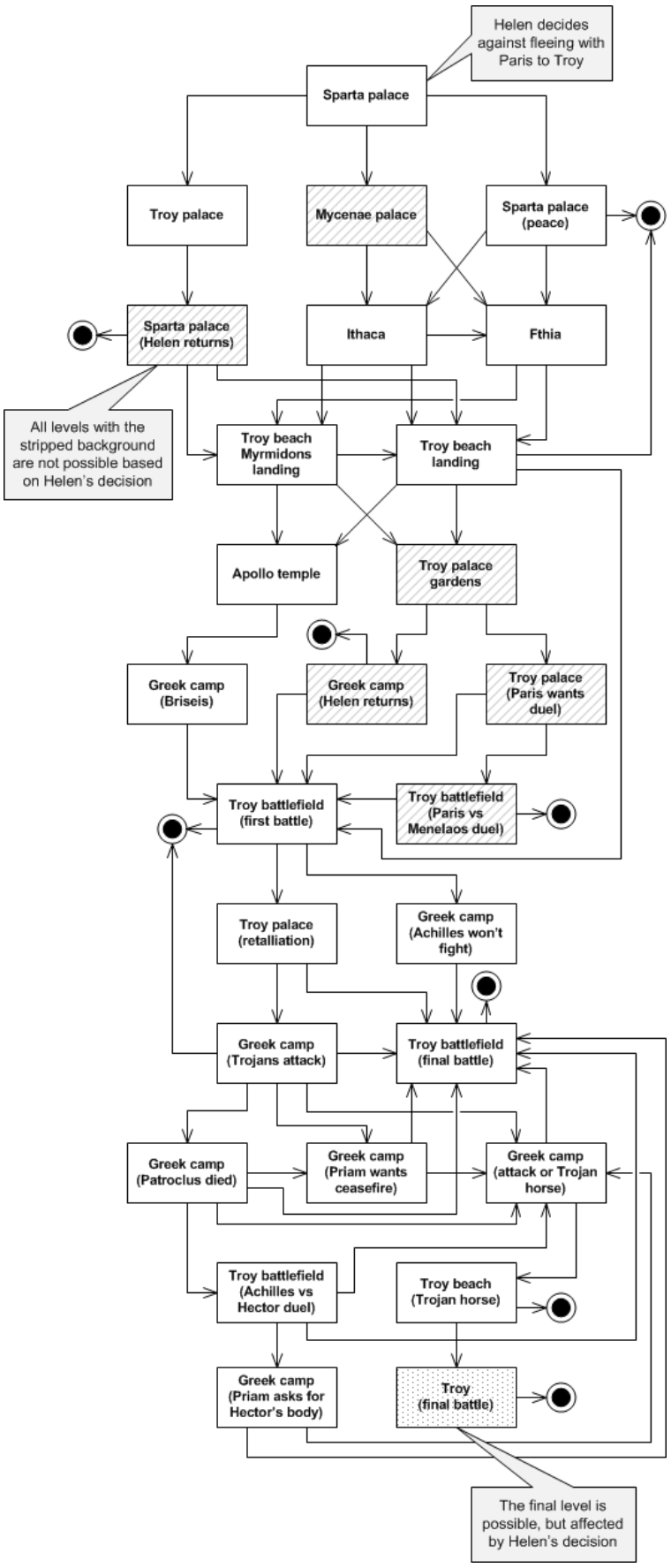

Figure 99: Levels affected if Helen decides not to flee with Paris

In (Goudoulakis et al., 2014) we documented as example of how some of the story mechanics are operating together to generate a story. To do that, we used a level which includes characters' action sequence, agents' coordination and injection of goals. In the finalised version of our scenario, some of the events of the level described here are used in separate levels. The level involves the following:

- Characters: *Achilles*, *Odysseus*, *Patroclus*, *Hector*, *Paris*, and *Priam*;
- Locations: *Achilles'* tent, *Greek* Camp, *Greek* Camp's Beach, Battlefield, and *Troy* Palace;
- Actions: go to, pick up, drop, ask for, allow to take, wear, talk to, tell, kill, prepare for funeral, perform funeral, attend funeral, find, wait.

As we have already explained, the characters, locations, and actions vary in each level of our complete scenario, depending on the nature and context of the level, which can help managing the complexity of the scenario, for example by reducing the number of actions considered in the planning and re-planning phases.

Achilles and *Priam* are initially located at *Achilles'* tent and *Troy* palace respectively. *Odysseus*, *Patroclus*, *Hector*, and *Paris* are located at the battlefield, and they are in the middle of the battle.

At some point, *Hector* kills *Patroclus*. This triggers a goal injection for *Odysseus*, which now has the following goals: To bring *Patroclus'* body back to the Greek camp; and to inform *Achilles* that *Patroclus* was killed by Hector. So, Odysseus leaves the battle, picks up *Patroclus'* body, and brings it back to the *Greek* camp. Afterwards, he goes to the tent where *Achilles* is located, and shares the news about *Patroclus'* death.

Achilles now has the following goals: To prepare for battle, to find and kill *Hector* as well as to defile his body by taking it with him to take revenge for *Patroclus'* death, and, finally, to perform *Patroclus'* funeral. He gears up, goes to the battlefield where he joins the on-going battle (*Odysseus* does the same as well), and kills *Hector*. When he does, that triggers the end of the battle, and he takes *Hector's* body with him, and returns to the camp (*Odysseus* returns as well) to prepare for *Patroclus'* funeral. When the preparation starts, the goal to attend the funeral is injected to *Odysseus*.

The death of *Hector* triggers a goal injection for *Paris*. He has the task to return to *Troy* and give the bad news to *Priam*. When he does, *Priam* decides to find *Achilles* and ask him to allow for the body of *Hector* to return to *Troy* to have a proper funeral. So, he wears a disguise to be able to infiltrate the *Greek* camp, and goes there. He finds where *Achilles'* tent is, but since *Achilles* is at *Patroclus'* funeral, he waits there until it's over.

When *Achilles* returns, the two men have a discussion, and *Priam* convinces *Achilles* to let him take *Hector's* body with him. *Priam* does that, returns to *Troy*, and performs a funeral for *Hector*, with *Paris* attending it. As an example, Figure 100 displays the complete combined plan of Achilles.

```
PICK-UP ACHILLES SWORD ACHILLES-TENT
PICK-UP ACHILLES SHIELD ACHILLES-TENT
WEAR ACHILLES ARMOUR ACHILLES-TENT
GO-TO ACHILLES ACHILLES-TENT BATTLEFIELD
ATTACK ACHILLES HECTOR BATTLEFIELD
KILL ACHILLES HECTOR BATTLEFIELD
PICK-UP ACHILLES HECTOR BATTLEFIELD
GO-TO ACHILLES BATTLEFIELD GREEK-CAMP
DROP ACHILLES HECTOR GREEK-CAMP
GO-TO ACHILLES GREEK-CAMP BEACH
PREPARE-FOR-FUNERAL GREEK ACHILLES BEACH
PERFORM-FUNERAL GREEK ACHILLES BEACH
GO-TO ACHILLES BEACH ACHILLES-TENT
ALLOW-TO-TAKE ACHILLES PRIAM HECTOR
```

Figure 100: Achilles' combined plan

Figure 101 illustrates all the interactions, coordination, and goal injections between the characters involved in the level.

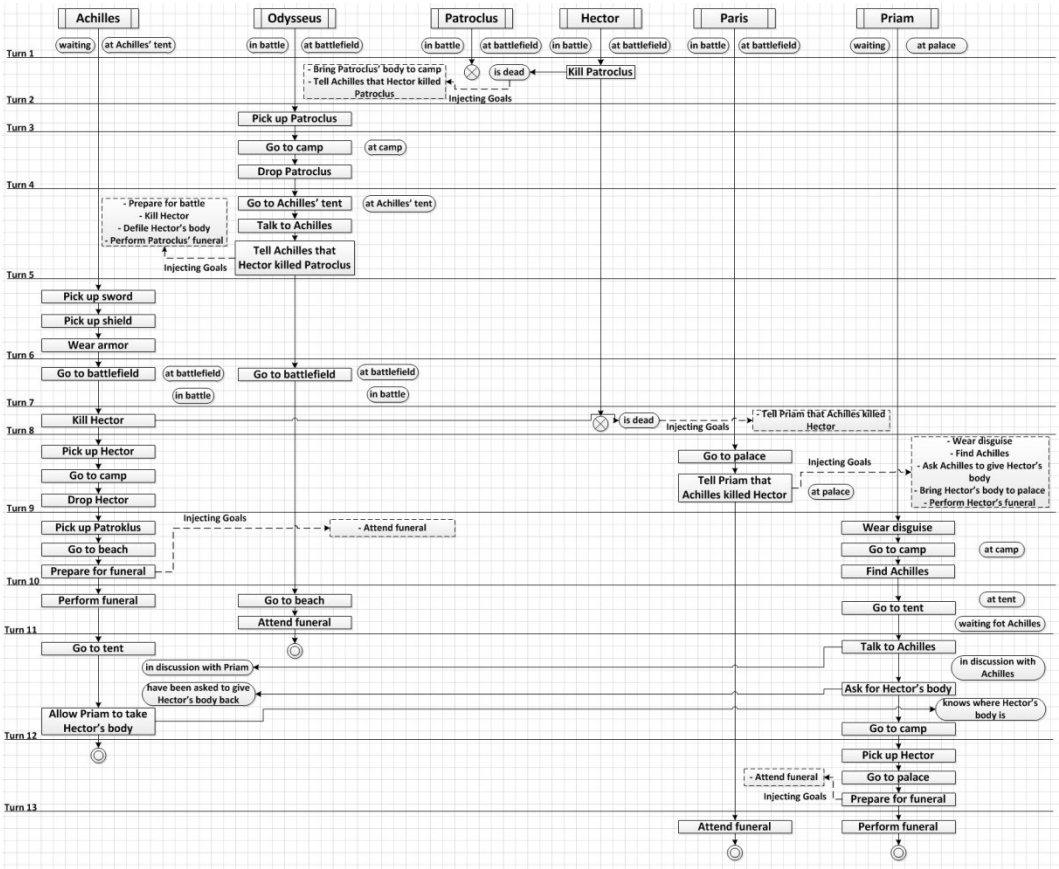

Figure 101: Troy characters coordination

6.4. Evaluation Metrics

To successfully evaluate several important parts of DIEGESIS, we needed to identify and decide on what to measure, in which way to measure it, and why to measure it.

As we discussed in section 3.4, to aid us decide on the base planning algorithms we would use to create our own planning solution, we performed an evaluation of planning algorithms with a Digital Interactive Storytelling (DIS) perspective in mind. The evaluation approach was to benchmark different planning algorithms testing their performance to solve a specific problem in a specific domain and to compare their feature sets which we considered valuable to DIS applications, which are explained in detail in section 3.4 and are the following: Support for extra language requirements; capability to generate partial-order plans; optimality; support for actions with costs; support for numeric variables.

Early in the implementation of our framework we discovered that the pre-processing of the planning domain (explained in detail in section 4.14) is a computationally intensive process which could be a potential bottleneck for the performance of the

whole system, depending on the design of our agents. We initially considered two different approaches in the design of our agents: In the first approach, we assumed that agents may be aware of different environments, each represented as different domain. Therefore, each agent who is aware of a different environment would have to perform its own individual pre-processing (i.e. to generate facts, operators etc.). In the second approach, we assumed that all the agents are aware of the same base environment; therefore they could share a pre-processed planning domain.

Since the early prototypes of DIEGESIS, we were using the second approach, but to make sure that the approach is scalable (it is a specification we specified in section 3.2) and to have an answer if the first approach is feasible in case we want to consider it in the future, we decided to perform a scalability evaluation, measuring the performance of the two approaches.

Another consideration that we had was that we considered two different planning methods (we discussed that in section 4.15). In the first one, each agent has a list of goals and generates a plan for all the goals at the same time, whilst in the second method a plan is generated only for the most important goal (the concept of goal importance was discussed in section 4.14), and when the goal is complete (or not applicable any more), a plan for the next goal is generated, and so forth. The storyteller is able to choose the way each agent will operate. We implemented and used both in DIEGESIS, but to identify which of the two approaches is better for our needs, we performed an evaluation measuring the performance of each approach and we also observed if one of them seemed to affect the generated story in any way, by measuring the generated actions.

Apart from our planning solution, our Planner includes a new re-planning solution as well. We evaluated its performance measuring the time needed to generate a valid plan after re-planning and we compared it to the approach of creating a new plan from scratch.

Finally, after we completed the design and implementation of DIEGESIS, as well as the modelling of the evaluation scenario we discussed in previous sections, we decided to evaluate the whole framework using that. Using DIEGESIS, we generated a number of different storylines and we measured generic data results such as the number of

levels, characters, choices, turns, actions, etc. We also focused on the performance aspects both of the planner and of the rest of the framework, and we finally observed some data related to vantage points.

6.5. EVALUATIONS

In this section, we are documenting a number of evaluations for the different components of our framework, using the evaluation scenarios we presented in sections 6.1 and 6.2.

6.5.1. PLANNING ALGORITHMS EVALUATION

As we discussed in section 3.4, to aid us to decide which planning algorithm to use as a base for our solution, we performed an evaluation of planning algorithms with a DIS perspective in mind. The approach we used to perform this evaluation was to benchmark different planning algorithms testing the speed they solved a specific problem in a specific domain and to compare their feature sets with DIS applications in mind.

The list of algorithms we benchmarked includes the following algorithms: FF (Hoffmann, 2001), Graphplan (Blum and Furst, 1997), LPG-TD (Gerevini et al., 2004), Marvin (Coles and Smith, 2004), Metric-FF (Hoffmann, 2003), SatPlan (Kautz and Selman, 1992), HSP (Bonet and Geffner, 2001), and JSHOP2 (Ilghami and Nau, 2003).

The feature sets considered valuable to DIS applications are the following, and were discussed in detail is section 3.4: Support for extra language requirements; capability to generate partial-order plans; optimality; support for actions with costs; support for numeric variables. Table 2 shows the list of features each algorithm implements.

Algorithm	TH	EO	NP	CE	EP	Opt	POP	AC	NV
FF	✓	✓	✓	✓	✓	×	×	×	×
Graphplan	×	×	×	×	×	✓	✓	×	×
LPG-TD	✓	✓	✓	×	✓	×	✓	✓	✓
Marvin	✓	✓	✓	✓	✓	×	✓	×	×
Metric-FF	✓	✓	✓	✓	✓	×	×	×	✓
SatPlan	✓	×	×	×	×	✓	✓	×	×
HSP	✓	✓	×	×	×	×	×	×	×
JSHOP2	×	×	✓	✓	×	×	✓	×	✓

Table 2: Algorithms' feature sets

The abbreviations used in the table are the following: Type hierarchies (TH); equality operator (EO); negative preconditions (NP); conditional effects (CE); existential preconditions (EP); optimal (Opt); partial-order planner (POP); action costs (AC); and numeric variables (NV).

As we explained in section 6.1, to benchmark the performance of the algorithms, the same test problem (i.e. Ugh's Story first act) as in (Barros and Musse, 2007b) was used. Since most of the algorithms tested could not handle all the language features needed for the test problem, the test problem was "ported" for each algorithm (e.g. defining an "eq" predicate for algorithms without a built-in equality operator). We performed the porting for the JSHOP2 algorithm, as it was not evaluated in the original paper.

The times reported are an average of three runs in a Virtual Machine (Ubuntu Linux on Oracle VirtualBox with 1024 MB of RAM and 1 core of the Quad-core CPU @ 2.40 GHz); the results of the benchmark are listed in Table 3.

Graphplan is an optimal and partial-order planner, but its input language is quite limited since none of the extra requirements described earlier is supported.

SatPlan is also optimal and is capable of generating partial-order plans. Its input language is quite limited since the only extra requirement it supports is type hierarchies.

HSP is a total-order non-optimal planner and its input language supports type hierarchies and has a built-in equality operator. It lacks all other features discussed earlier.

LPG-TD supports all the extra language requirements discussed before, except conditional effects. It generates partial-order plans, supports action costs and numeric variables but it is not optimal.

The FF (Fast-Forward) algorithm is non-optimal, produces total-order plans and its input language supports all extra requirements described before. It cannot handle numeric variables and actions with costs.

Metric-FF is an extension of FF which provides support for numeric variables. Also, although actions with costs are not supported explicitly, they can be easily "emulated"

with numeric variables, since Metric-FF allows for the definition of an optimality metric.

Marvin is based on FF's search strategy and supports the same input language as FF, but it is capable of generating partial-order plans.

JSHOP2, is a planner based on ordered task decomposition, a type of Hierarchical Task Network (HTN) planning. It supports conditional effects, negative preconditions, partial ordered plans, and numeric variables. Its language, although it incorporates many PDDL features, has a unique syntax. The problem and domain descriptions must be compiled into Java code (by the system) before execution therefore there might be a problem with potential use with real-time planning systems.

Algorithm	**Barros**	**Time**
FF	0.015 s	0.023 s
Graphplan	0.138 s	0.240 s
LPG-TD	0.169 s	0.463 s
Marvin	0.017 s	0.040 s
Metric-FF	0.018 s	0.036 s
SatPlan	0.247 – 0.512 s	0.630 s
HSP	0.031 s	-
JSHOP2	-	0.021 s

Table 3: Algorithms' execution time comparison

The times produced by our evaluation comply with the times produced by the original evaluation. JSHOP2 produced the fastest solution, with FF, Metric-FF and Marvin coming very close. Graphplan, LPG-TD and SatPlan on the other hand showed an increase in the solution time needed.

By examining the above analysis, it can be seen that there is no planning algorithm that combines all the characteristics described before. Therefore, we can conclude, and agree with (Barros and Musse, 2007b) that no planning algorithm can be considered ideal for DIS applications. Based on the available planning algorithms, and considering that each DIS system has its own goals, the final choice of algorithm must be done based on the unique requirements of each DIS system (the same way as it is done until now).

6.5.2. DIEGESIS SCALABILITY EVALUATION

As we discussed in section 3.2, while designing and prototyping DIEGESIS it was a necessity that the performance of the system and the algorithms would be good enough to accommodate large stories with several characters (represented by agents). Therefore, during the implementation of our system, we had to constantly evaluate its performance, to ensure that the system stays responsive and usable even when using large stories.

Early in the implementation of our system we discovered that the initialisation (pre-processing) of the planning domain (explained in section 4.14) is a computationally intensive process which could be a potential bottleneck for the performance of the whole system, depending on the design of our agents. We initially considered two different approaches in the design of our agents: In the first approach, we assumed that agents may be are aware of different environments, each represented as different domain. Therefore, each agent who is aware of a different environment would have to perform its own individual pre-processing (i.e. to generate facts, operators etc.). In the second approach, we assumed that all the agents are aware of the same base environment; therefore they could share a pre-processed planning domain.

So, the first evaluation that we performed (Goudoulakis et al., 2012b) was to test the performance and scalability of DIEGESIS with an increasing number of active agents in the system, was to measure the agent initialisation time (including the pre-processing time of our planning algorithm), as well as the plan generation time.

As a planning domain, we used one of the levels of the Troy scenario we presented in section 6.2, which -at that point- was still early in its design and modelling. In that level, each agent represents the character called *Paris* with a goal of convincing character *Helen* to come with him to the city of *Troy*. The valid plan consists of 4 actions and is displayed in Figure 102.

```
GOTO PARIS TROY-PALACE SPARTA-PALACE
SEDUCE PARIS HELEN SPARTA-PALACE
CONVINCE-TO-FOLLOW PARIS HELEN SPARTA-PALACE
TAKE-SOMEONE-FROM-TO PARIS HELEN SPARTA-PALACE TROY
```

Figure 102: The generated valid plan

To perform our tests, a varying number of agents are activated, and are trying to generate the valid plan of Figure 102 using an instance of the Planner component. To be able to do that, as we already discussed in section 4.14, the Planner needs to translate the domain into facts (e.g. “at paris troy-palace”), operators (e.g. “seduce paris helen sparta-palace”), mutexes between them, etc. to be used in the graph expansion and extraction phases.

After a successful pre-processing, the evaluation domain contains a total of 758 facts and 3398 operators. The times reported in this evaluation are an average of five runs for each case in a PC with 4GB of RAM and a Quad-core CPU @ 2.83 GHz, running Windows 7.

We evaluated both of the approaches mentioned earlier in this section, and our results are documented in Table 4, and illustrated in Figure 103. We initially tested for one agent, and then for 10, 50, and 100 agents simultaneously.

Agent execution environment		**Number of agents**			
		1 agent	**10 agents**	**50 agents**	**100 agents**
Different domain	Initialisation time	0.62s	5.19s	26.12s	51.7s
	Planning time	0.05s	0.22s	1.35s	5.84s
Same Domain	Initialisation time	0.62s	0.63s	0.63s	0.63s
	Planning time	0.05s	0.23s	1.11s	3.41s
Multi-threaded	Total time	0.68s	2.11s	8.98s	20.65s

Table 4: Pre-processing and planning times vs number of agents

If we compare the planning times between the two approaches (i.e. different domain & same domain), we can see a slight increase in the different domain approach while the number of agents increases, but it is clear that the real potential bottleneck is the initialisation (pre-processing) time.

To solve the obvious scalability issue of the first approach (i.e. each agent is aware of a different domain), we performed a third test with the same principle. In that, we altered our system’s implementation so it can initialise the agents in a multi-threaded fashion instead of the linear initialisation of the first approach, to improve its performance. The results of this approach reveal a remarkable improvement compared to the results of the first approach (20.65 seconds against 57.54 seconds for 100 agents).

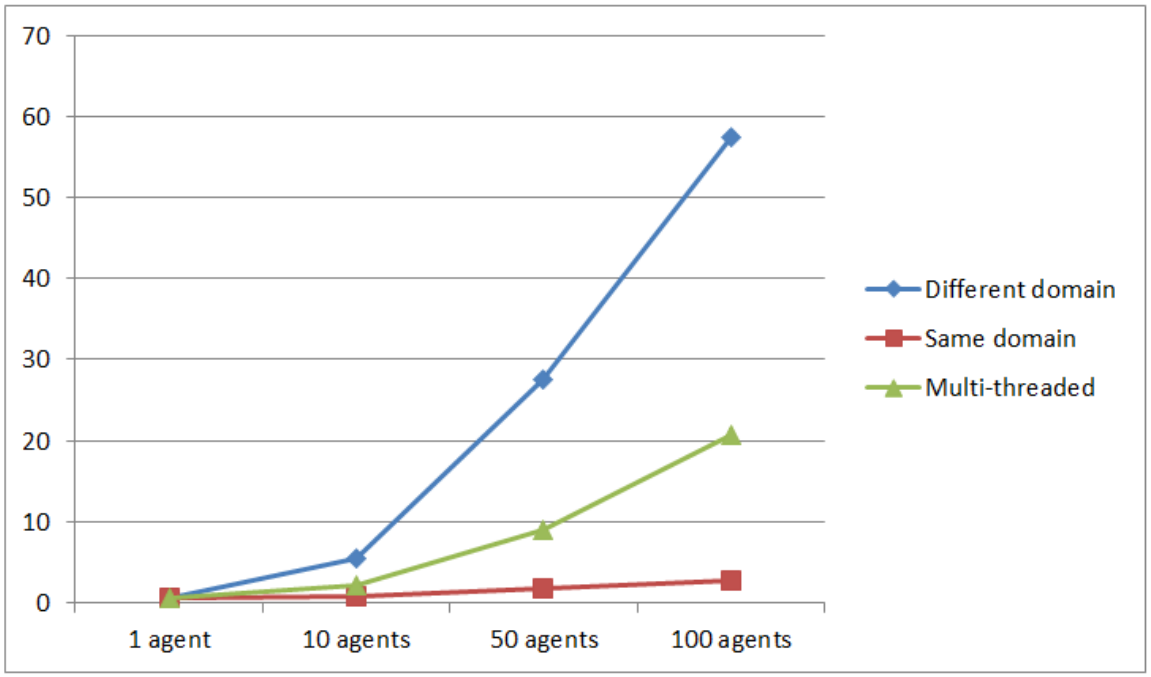

Figure 103: Total times vs number of agents

After we progressed with the implementation of the system and we expanded the Troy scenario, we decided to perform a similar scalability evaluation to further test the performance of DIEGESIS, which we published in (Goudoulakis et al., 2012a). In the same manner as before, we decided to test the performance and scalability the framework with an increasing number of active agents in the system by measuring the agent initialisation and plan generation time.

As a planning domain, we used an extended version of the level used in the previously discussed scalability evaluation, which is part of the Troy scenario discussed in section 6.1. Again, each agent represents the character called "Paris" with a goal of seducing and convincing character "Helen" to flee with him back to the city of "Troy". This time, the valid plan of each agent consists of 8 actions (instead of 4 actions that the previous plan included) and is displayed in Figure 104.

```
WALK-TO PARIS TROY-THRONE-ROOM TROY-DOCK TROY
TRAVEL-TO PARIS TROY-DOCK SPARTA-DOCK SHIPS SEA
WALK-TO PARIS SPARTA-DOCK SPARTA-THRONE-ROOM SPARTA
TALK-TO PARIS HELEN SPARTA-THRONE-ROOM
SEDUCE PARIS HELEN SPARTA-THRONE-ROOM
CONVINCE-TO-FOLLOW PARIS HELEN SPARTA-THRONE-ROOM
WALK-TO-WITH PARIS HELEN SPARTA-THRONE-ROOM SPARTA-DOCK SPARTA
TRAVEL-TO-WITH PARIS HELEN SPARTA-DOCK TROY-DOCK SHIPS SEA
```

Figure 104: The generated valid plan

Similarly to the previous scalability evaluation, a varying number of agents are activated, and are trying to generate the valid plan of Figure 104 using an instance of the Planner component. After a successful pre-processing, the evaluation domain contains a total of 816 facts and 7696 operators (accordingly, 758 and 3398 in the previous evaluation). The times reported in this evaluation are an average of five runs for each case in a PC with 4GB of RAM and a Quad-core CPU @ 2.83 GHz, running Windows 7, the same infrastructure used in the previous evaluation.

We evaluated the same two approaches as in the previous evaluation (i.e. agents have individual domains, or share a domain). Our results are documented in Table 5, and illustrated in Figure 105. We initially tested for one agent, and then for 10, 50, and 100 agents simultaneously.

Table 5 displays the times needed for the all agents to be initialised (including the pre-processing time) and the times needed for all the agents to generate a valid plan.

Agent execution environment		**Number of agents**			
		1 agent	**10 agents**	**50 agents**	**100 agents**
Different domain	Initialisation time	2.159s	13.914s	76.909s	163.930s
	Planning time	0.085s	0.522s	2.818s	8.780s
Same Domain	Initialisation time	2.159s	2.160s	2.162s	2.181s
	Planning time	0.085s	0.520s	2.803s	8.771s
Multi-threaded	Total time	1.355s	6.419s	32.248s	65.318s

Table 5: Pre-processing and planning times vs number of agents

As we already identified, the difference between the plan generation time of each approach is minimal, therefore the main impact derives from the initialisation time. Similarly to the previous evaluation, the different domain approach shows a scalability issue, therefore we tested the same domain with the multi-threaded execution approach which have implemented as a solution during our previous evaluation, which (as expected) showed a significant improvement compared to the results of the first approach.

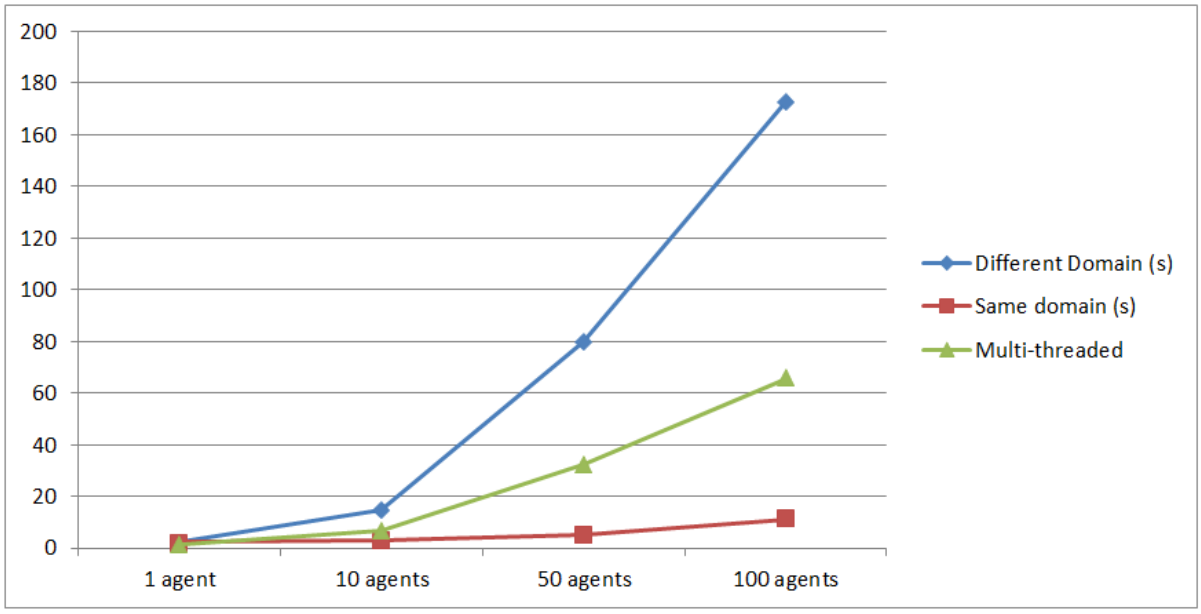

Figure 105: Total times vs number of agents

When we published the results of our first evaluation in (Goudoulakis et al., 2012b), we concluded that we would continue to use the second approach (i.e. all agents are aware of the same domain), since the likelihood of all the characters that need to generate a plan at the same time being aware of a different domain is extremely low. The same conclusion was reached after the second evaluation as well. Ultimately, as we continued to implement the system and grow the scenario we decided that this is the approach that fits best our needs, and therefore, as we documented in chapter 4, this is the approach which DIEGESIS is designed to use.

6.5.3. Planning Algorithm Performance Evaluation

As we already discussed in section 4.15, we have considered two different planning methods. In the first one, each agent has a list of goals and generates a plan for all the goals at the same time, whilst in the second method a plan is generated only for the most important goal (the concept of goal importance was discussed in section 4.14), and when the goal is complete (or not applicable any more), a plan for the next goal is generated, and so forth. The storyteller is able to choose the way each agent will operate.

To identify which of the two approaches is better for our needs, we performed an evaluation, which we published in (Goudoulakis et al., 2014), using a modified part of the scenario we discussed in section 6.2, which includes the following elements:

- Characters: *Paris*, *Hector*, *Menelaos*, *Helen*, and *Agamemnon*;
- Locations: throne room, guest room, royal bedroom, kitchen, armoury, dock, stable, and garden (all located in *Sparta*), as well as *Troy's* dock;
- Actions: talk to, walk to, travel to, pick up, drop, ask to give, give, kill, seduce, convince to follow, board, negotiate peace, drink, and eat.

We are using *Paris* and *Hector* as our main characters and the rest as dummy characters. By the term "dummy characters" we mean that they don't have any initial goals associated with them; the system injects goals to them during the execution based on other agents' actions, or just futile goals.

Hector has the following three goals: first, to give a present brought from *Troy* to *Menelaos* (since he is his guest), then to negotiate peace between *Troy* and *Sparta*, and finally, to return back to *Troy*. Figure 106 displays a possible plan for Hector which achieves all his goals.

```
WALK-TO HECTOR GUEST-ROOM THRONE-ROOM
TALK-TO HECTOR MENELAOS THRONE-ROOM
GIVE HECTOR MENELAOS HECTOR-TROY-GIFT THRONE-ROOM
NEGOTIATE-PEACE HECTOR TROY MENELAOS SPARTA THRONE-ROOM
WALK-TO HECTOR THRONE-ROOM DOCK
BOARD HECTOR SHIPS DOCK
TRAVEL-TO HECTOR DOCK TROY SHIPS SEA
```

Figure 106: Hector's plan

Paris has five goals, which are to give a present to *Menelaos* as well, to seduce *Helen*, convince her to follow him back to *Troy*, engage in small talk with *Agamemnon* who is around at the palace, and finally return to *Troy*. Figure 107 displays a possible plan for *Paris* which achieves all his goals.

We measured the following, for both agents for both of the approaches: the total planning time needed (in seconds), the average time needed to generate a single plan, the total number of nodes in the planning graph, the average number of nodes for each plan, and the number of successful, failed, and wait actions during the scenario execution.

```
WALK-TO PARIS GUEST-ROOM GARDEN
TALK-TO PARIS AGAMEMNON GARDEN
WALK-TO PARIS GARDEN ROYAL-BEDROOM
TALK-TO PARIS HELEN ROYAL-BEDROOM
CONVINCE-TO-FOLLOW PARIS HELEN ROYAL-BEDROOM TROY
WALK-TO PARIS ROYAL-BEDROOM THRONE-ROOM
TALK-TO PARIS MENELAOS THRONE-ROOM
GIVE PARIS MENELAOS PARIS-TROY-GIFT THRONE-ROOM
WALK-TO PARIS THRONE-ROOM DOCK
BOARD PARIS SHIPS DOCK
TRAVEL-TO PARIS DOCK TROY SHIPS SEA
```

Figure 107: Paris' plan

The data is an average of five runs for each case in a PC featuring 4GB of RAM, and a Quad-core CPU @ 2.83 GHz, running Windows 7.

The total planning time for each approach, for each agent, as well as the average planning time for each plan are displayed in Table 6, and illustrated in Figure 108. (a) is the case where a plan is generated for all the goals at the same time, where (b) the case where a plan is generated for only one goal at a time.

Planning methods & characters		**Planning times**		**Number of nodes**	
		Average	**Total**	**Average**	**Total**
Case (a) (all goals)	Paris	1.3783s	4.135s	6476	19428
	Hector	1.1265s	2.253s	9459.5	18919
Case (b) (one goal)	Paris	0.1902s	0.951s	784.2	3921
	Hector	0.1504s	0.752s	748.8	3744

Table 6: Planning evaluation results

The combined total planning time for both of the agents in case (a) was 6.388 seconds, where in case (b) was only 1.703 seconds, a 73.34% decrease. In case (a), the highest planning time that appeared was 2.216 seconds, and the lowest 0.735 seconds. In contrast, in case (b), the highest was 0.486 seconds, and the lowest 0.017 seconds.

The total average for both of the agents in case (a) was 1.2524 seconds, and in case (b) 0.1703 seconds, a decrease of 86.4%. Based on the above numbers, it's clear that there is a significant performance increase using the case (b).

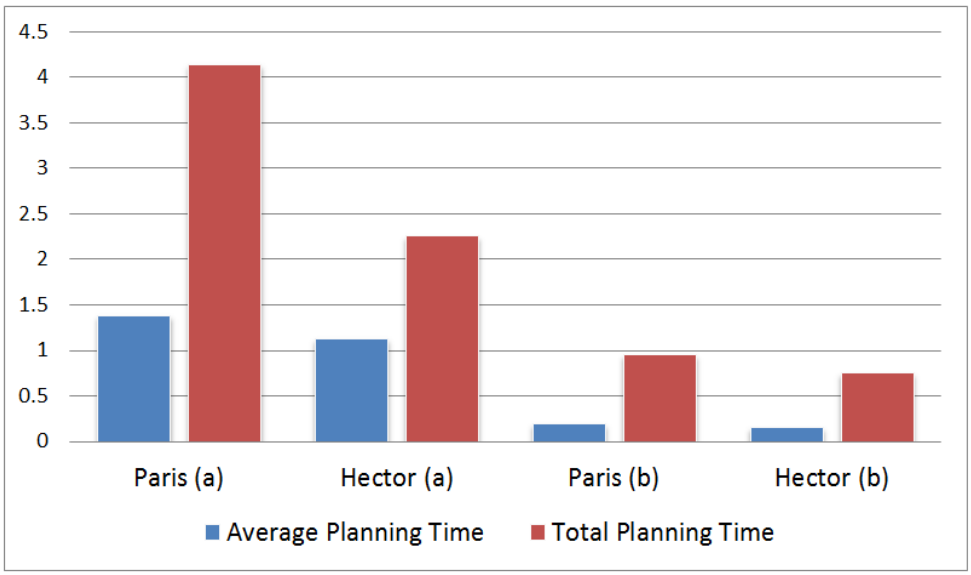

Figure 108: Total and average planning times

Figure 109 illustrates the total and average number of nodes generated in the planning graph while planning, for both agents, in both cases. The number of nodes is displayed in Table 6.

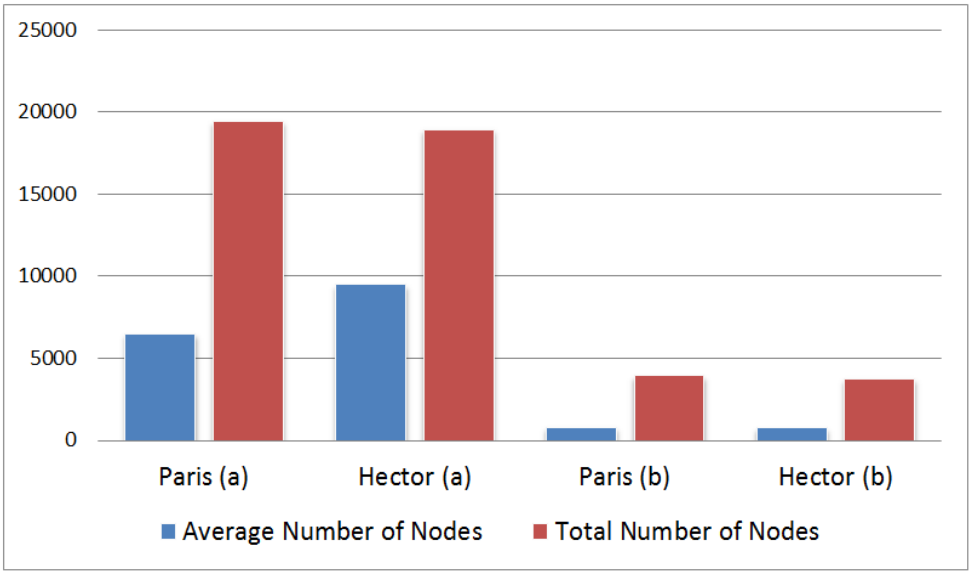

Figure 109: Total and average number of nodes in planning graph

The reduction in the number of nodes is consistent with the reduction in time. More specifically, the total number of nodes in case (b) is 80% less than in case (a), and the average number of nodes of case (a) are showing a 90.38% reduction comparing them with the average of case (b).

Table 7 displays (and Figure 110 illustrates) the successful actions execution, the failed actions, the number of times each agent was instructed to wait, and the total number

of plans generated by each agent throughout the previously mentioned scenario. In case (a) the number consists of the initial plan plus the re-planning because of failures, and in case (b) consists of the plans for each individual goal, plus the re-planning because of actions' failure.

Planning methods & characters		Successful actions	Failed actions	Instructed to wait	Total plans
Case (a) (all goals)	Paris	11	2	0	3
	Hector	8	1	2	2
Case (b) (one goal)	Paris	12	1	0	5
	Hector	9	2	2	5

Table 7: Successful and failed actions, times instructed to wait, and numbers of total generated plans

There were more plans generated in case (b), something which was expected since the agents needed to generate a plan for each individual goal, but since the complexity of each plan was quite lower than the plans of case (a), the planning times were quite lower as discussed before in Figure 108, therefore the increased number of plans was not a negative impact on the performance.

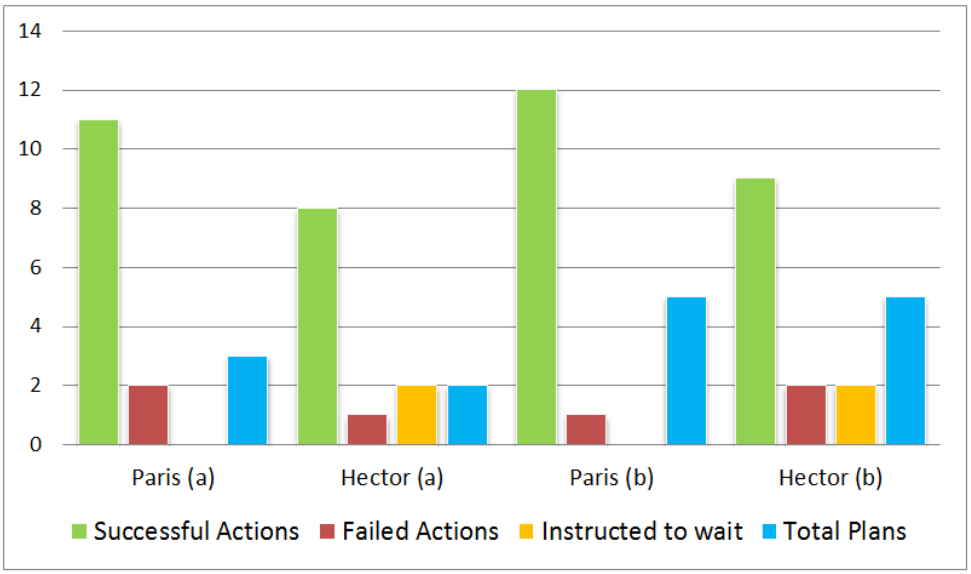

Figure 110: Successful and failed actions, times instructed to wait, and numbers of total generated plans

The rest of the metrics in Table 7 and Figure 110 are similar to each other between the two cases. The times an agent was instructed by the system to wait a turn before trying to execute its action again were identical, as well as the failed actions. There was a slight increase in the successful actions in case (b), which can potentially add to the emergence of the generated narrative.

6.5.4. Re-planning Algorithm Evaluation

To evaluate the performance of our re-planning solution, we used a part of the scenario we discussed in section 6.2, modified for the evaluation purposes. The evaluation scenario includes the following elements:

Paris as the main character, *Hector*, *Menelaos*, *Helen*, and *Agamemnon* as "dummy" characters (they exist only to interact with the main character when needed); guest room, garden, armoury, royal bedroom, throne room, kitchen, and dock as locations (all located in *Sparta*); and several actions such as talk, walk, seduce, ask for item, etc. which can be performed by the agents.

Figure 111 illustrates the complete plan of the agent which represents character *Paris* in the scenario.

```
WALK-TO PARIS GUEST-ROOM GARDEN
TALK-TO PARIS AGAMEMNON GARDEN
WALK-TO PARIS GARDEN ARMOURY
PICK-UP PARIS MAP ARMOURY
WALK-TO PARIS ARMOURY ROYAL-BEDROOM
TALK-TO PARIS HELEN ROYAL-BEDROOM
SEDUCE PARIS HELEN ROYAL-BEDROOM
CONVINCE-TO-FOLLOW PARIS HELEN ROYAL-BEDROOM TROY
WALK-TO PARIS ROYAL-BEDROOM THRONE-ROOM
TALK-TO PARIS MENELAOS THRONE-ROOM
GIVE PARIS MENELAOS PARIS-TROY-GIFT THRONE-ROOM
WALK-TO PARIS THRONE-ROOM DOCK
BOARD PARIS SHIPS DOCK
TRAVEL-TO PARIS DOCK TROY SHIPS SEA
```

Figure 111: Paris' complete plan

Paris is initially located in the guest room in *Sparta*, and he plans to go to the gardens and chat with *Agamemnon*, then to pick up a map located in the armoury, then to find *Helen* in the royal bedroom, seduce her, and convince her to follow him back to *Troy*. Afterwards, to find *Menelaos* at the throne room and give him a gift, and finally, to walk to the docks and board to a ship to return to *Troy*.

While the plan is executed, it is scheduled to fail three times in the following situations:

1. *Hector* has taken the map, so *Paris* needs to find him and ask for the map;
2. *Helen* is no longer located at the royal bedroom;

3. *Menelaos* is no longer located at the throne room.

We evaluated two cases: In the first one, when a plan fails, the re-planning is performed from scratch. The previously generated plan is discarded, and a new plan is generated from the current state in which an action has failed. In the second case, the re-planning algorithm documented in section 4.14 is used, generating a new partial plan only for the failed action(s), which is then merged with the rest of the existing plan.

Figure 112 illustrates the re-planning process and the action execution sequence of the second case, for the scenario used in the evaluation. When an action fails, a partial plan is generated, which accomplishes the effects of the failed action, as well as the effects of the next action (if exists). It is then merged with the initial plan, and the execution of the actions continues as usual.

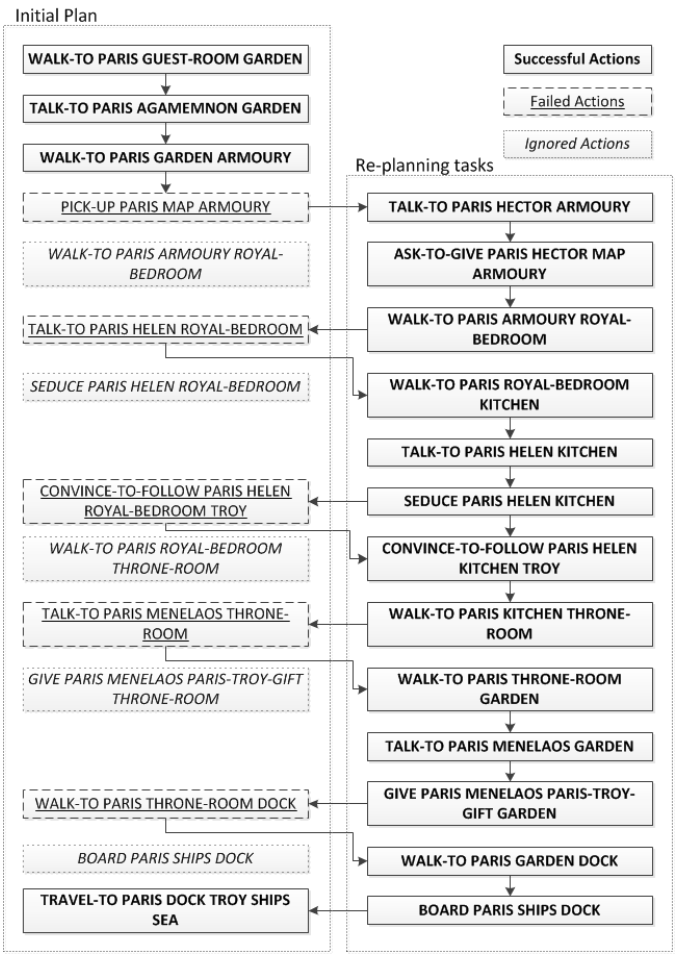

Figure 112: Re-planning process and action execution of the new re-planning solution

We measured the total re-planning time needed (in seconds), the average re-planning time needed for each plan, the total number of nodes in the planning graph, the

average number of nodes for each plan, and the number of re-plans occurred during the scenario execution.

The data is an average of five runs for each case in a PC featuring 4GB of RAM, and a Quad-core CPU @ 2.83 GHz, running Windows 7.

Figure 113 illustrates the number of the re-planning tasks occurred during the execution of the plans. (a) is the case where a plan was generated from scratch, whilst (b) is the case where a partial plan was generated.

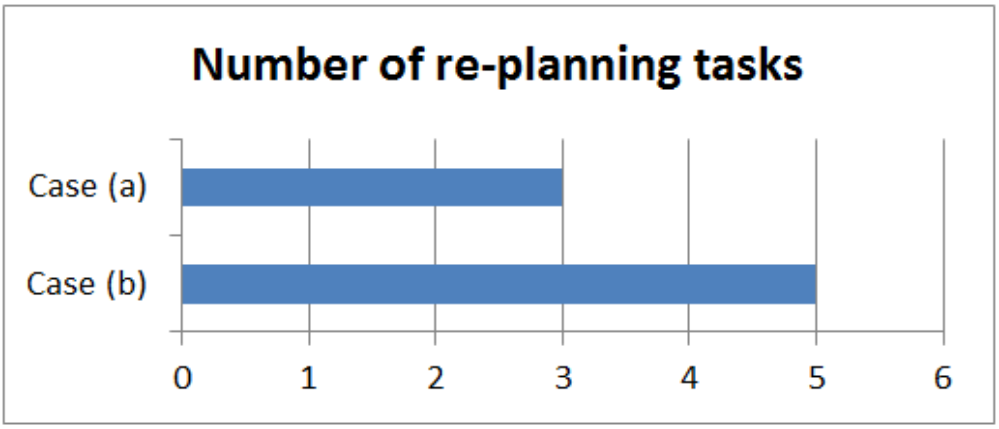

Figure 113: Number of re-planning tasks

Case (b) has an increased number of re-planning tasks, due to the following: The re-planning solution generates a partial plan which connects the failed actions with the next step of actions. But, most of the actions in our scenario are modelled in a way that binds them to locations. During the evaluation, we noticed 2 cases where a second partial plan needed to be generated to complete a long sequence of actions in a specific location which was different during the initial plan generation.

Table 8 contains the total and average planning times for each of the two re-planning method cases (illustrated in Figure 114), as well as the total and average number of nodes of each case (illustrated in Figure 116).

Re-planning methods	Planning times		Number of nodes	
	Average	Total	Average	Total
Case (a) (new plan from scratch)	0.065s	0.194s	1274.67	3824
Case (b) (partial plan)	0.014s	0.070s	314.25	1257

Table 8: Planning times and number of nodes

The total planning time in case (a) was 0.194 seconds, where in case (b) was only 0.07 seconds, a 63.92% decrease.

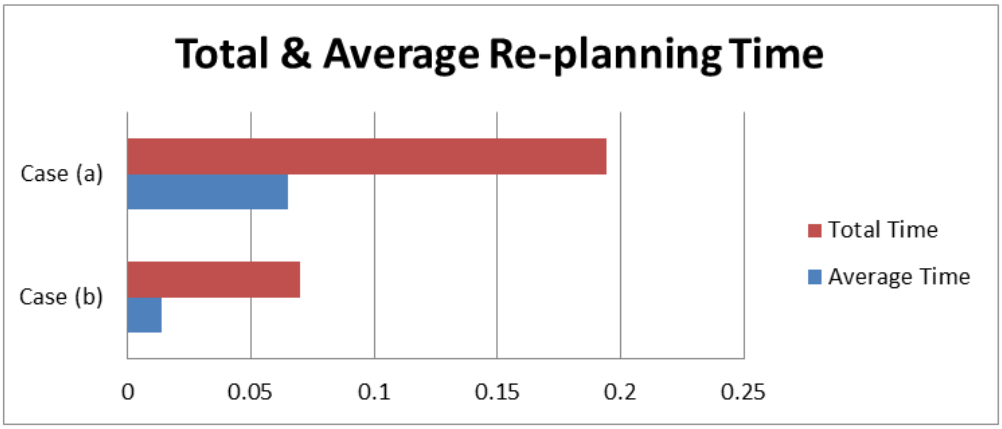

Figure 114: Total and average re-planning times

The individual planning times and number of nodes for each individual re-planning task in each case are documented in Table 9, and illustrated in Figure 115 and Figure 117.

Re-planning methods		Planning tasks				
		1	**2**	**3**	**4**	**5**
Case (a) (from scratch)	Times	0.107s	0.056s	0.031s	-	-
	Nodes	1635	1384	805	-	-
Case (b) (partial plan)	Times	0.019s	0.014s	0.011s	0.016s	0.010s
	Nodes	259	273	203	328	194

Table 9: Planning tasks results

In case (a), the highest re-planning time that appeared was 0.107 seconds, and the lowest 0.031 seconds. In contrast, in case (b), the highest was 0.019 seconds, and the lowest 0.01 seconds.

The average re-planning time for each re-planning task in case (a) was 0.065 seconds, and in case (b) 0.014 seconds, a decrease of 78.46%.

Based on the above numbers, it is clear that there is a significant performance increase in case (b).

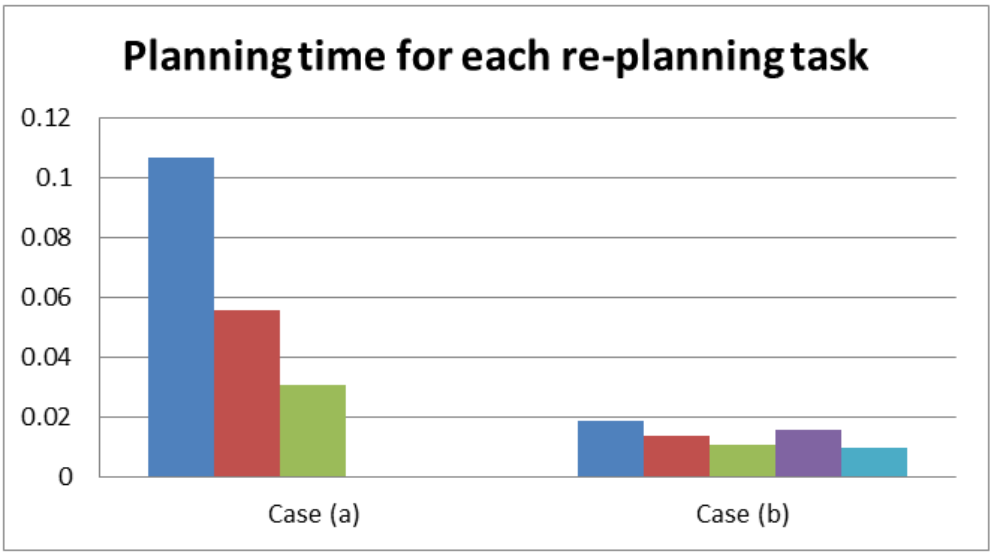

Figure 115: Individual planning time for each re-planning task

The reduction in the number of nodes is consistent with the reduction in time. More specifically, the total number of nodes in case (b) is 67.13% less than in case (a), and the average number of nodes of case (a) are showing a 80.28% reduction comparing them with the average of case (b).

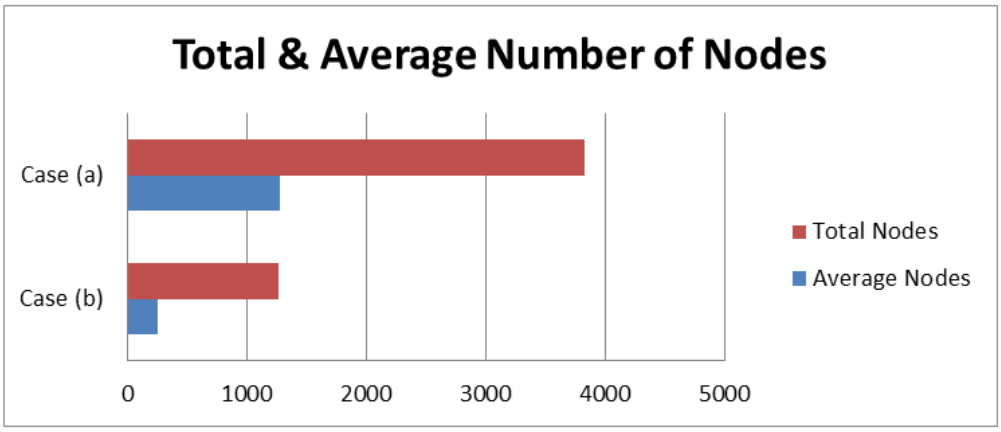

Figure 116: Total and average number of nodes

Based on the results of the evaluation, the approach to generate a partial plan which solves the failed actions and to connect it with the rest of the valid plan has a significantly better performance from the approach to re-plan from scratch. The increased number of re-planning tasks we observed and discussed earlier in this section does not have any impact of the overall performance of the solution.

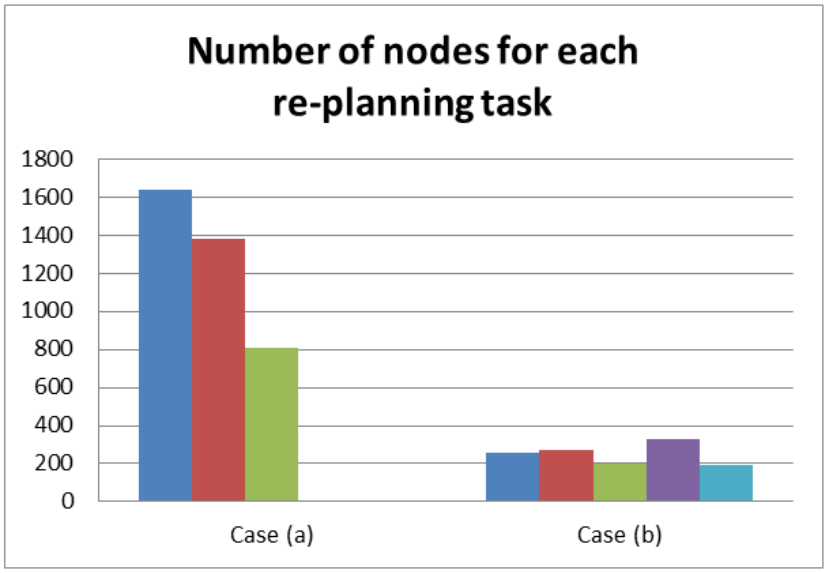

Figure 117: Individual number of nodes for each re-planning task

Finally, apart from a better performance, our re-planning solution did not affect in a negative way the outcome of the generated story compared to the other solution.

6.5.5. SUMMARISATION EVALUATION

To evaluate the whole framework with the Troy evaluation scenario (discussed in section 6.2) we modelled for this purpose, we used DIEGESIS to generate 4 different storylines and measured a number of metrics, which we will document in this section.

In the first storyline (from now on we will refer to it as "story 1"), we intentionally used the appropriate choices, and made some minor modifications (merely to the fighting abilities of some key characters and the volumes of some battle groups) to the modelling of the scenario to produce an outcome consistent with the Troy movie in which we based our scenario. The 21 levels generated and executed are illustrated in Figure 118, are marked in a blue-shaded background and their execution order is mentioned before their title (the same principles apply to all the illustrated outcomes).

In the rest three storylines, the modifications we made for the first story were removed and all the choices were made by the Oracle component (discussed in section 4.10), producing a relatively random outcome in each storyline. In the second storyline (i.e. "story 2"), 10 levels were executed, which are illustrated in Figure 119. In the third storyline (i.e. "story 3"), 6 levels were executed (illustrated in Figure 120), while in the fourth and final storyline, 13 levels were executed which are illustrated in Figure 121.

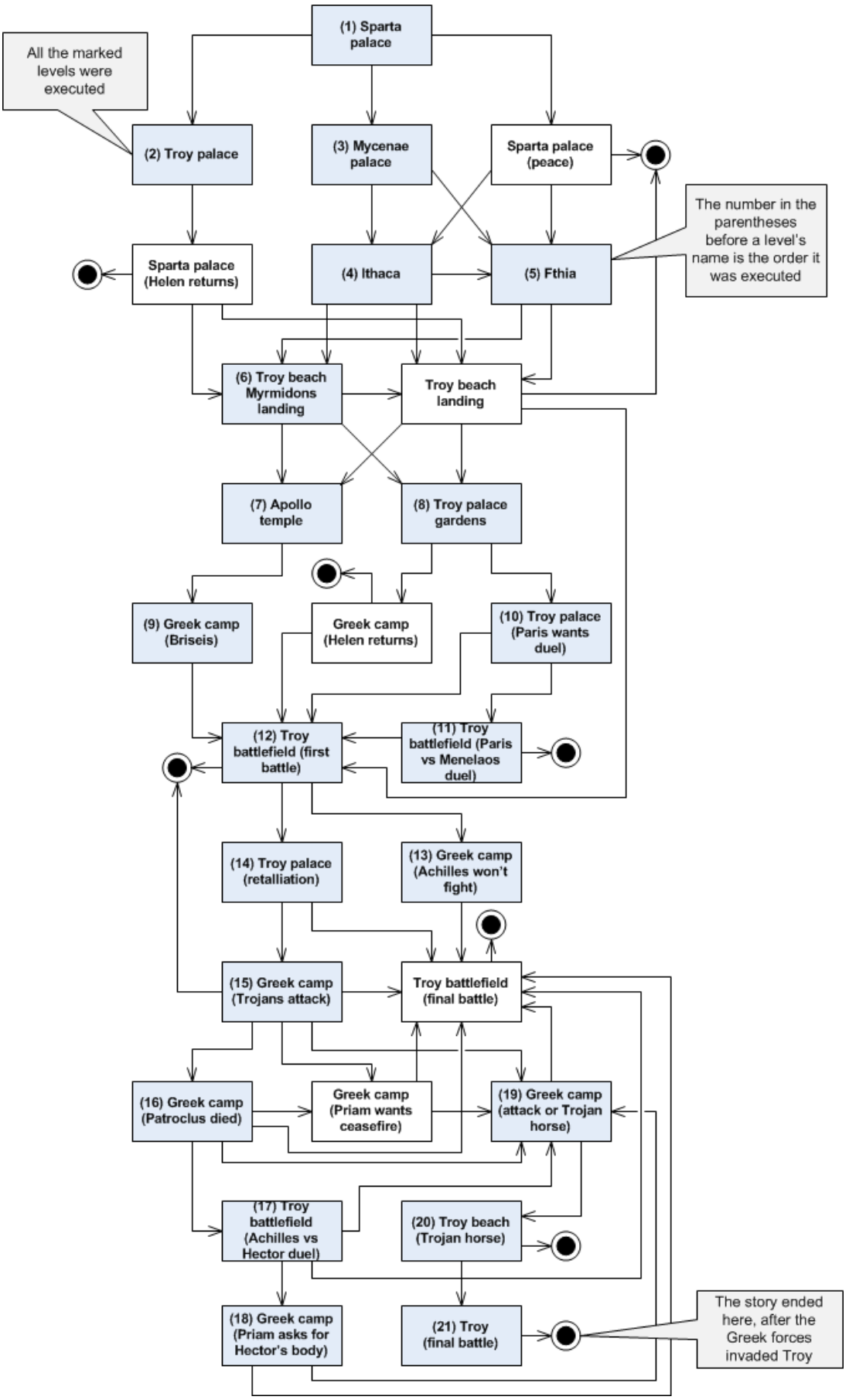

Figure 118: Execution order of levels in Story 1

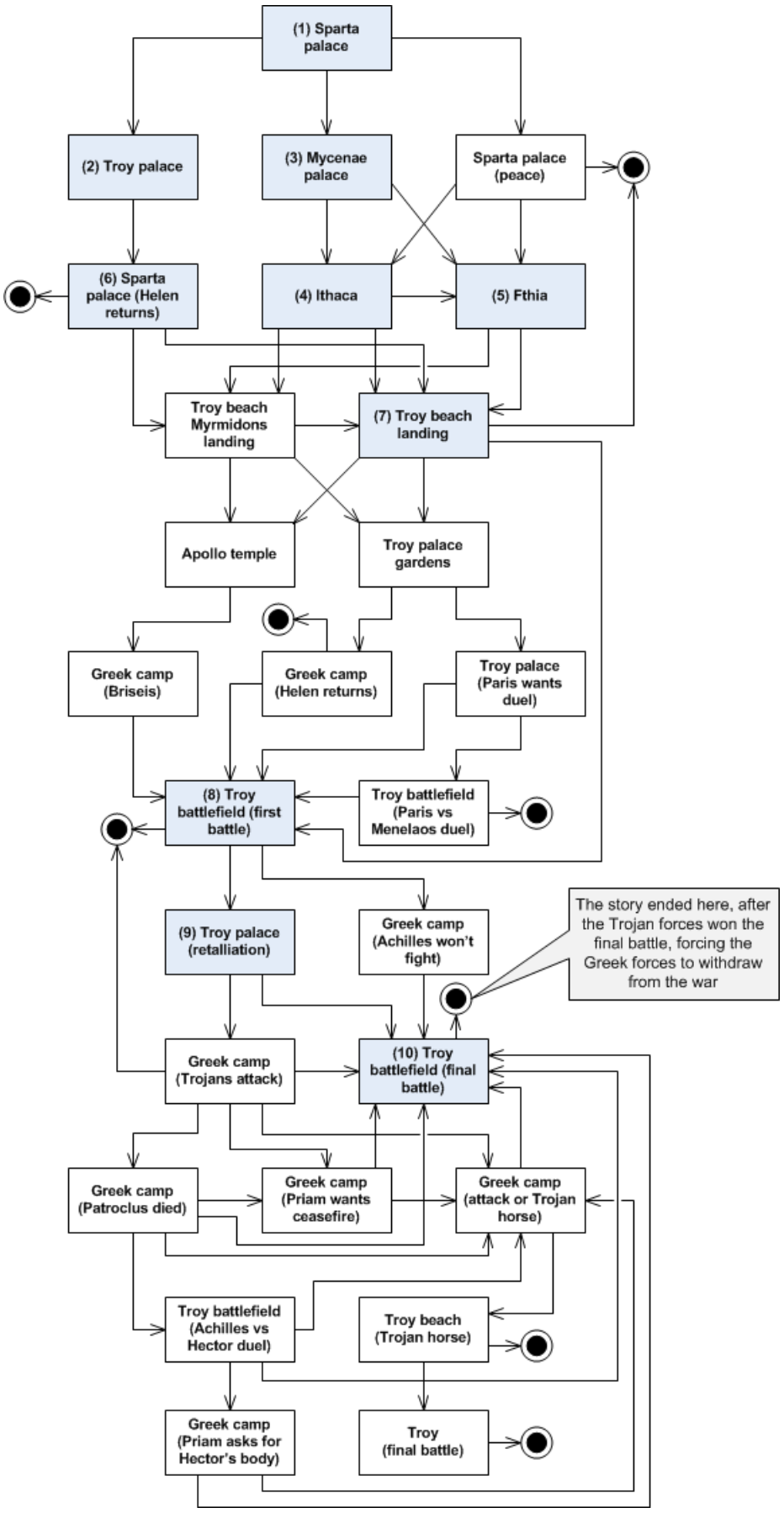

Figure 119: Execution order of levels in Story 2

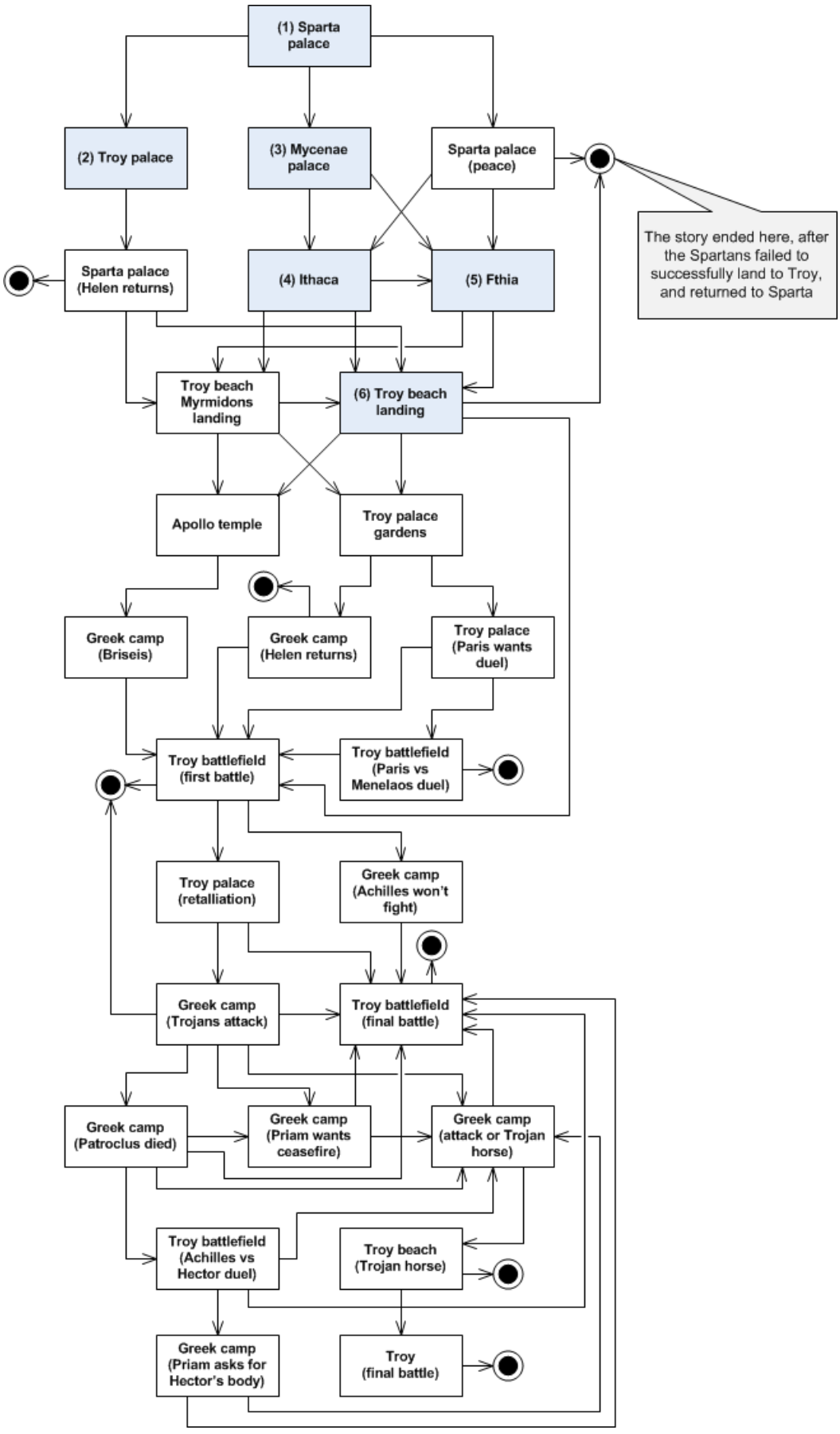

Figure 120: Execution order of levels in Story 3

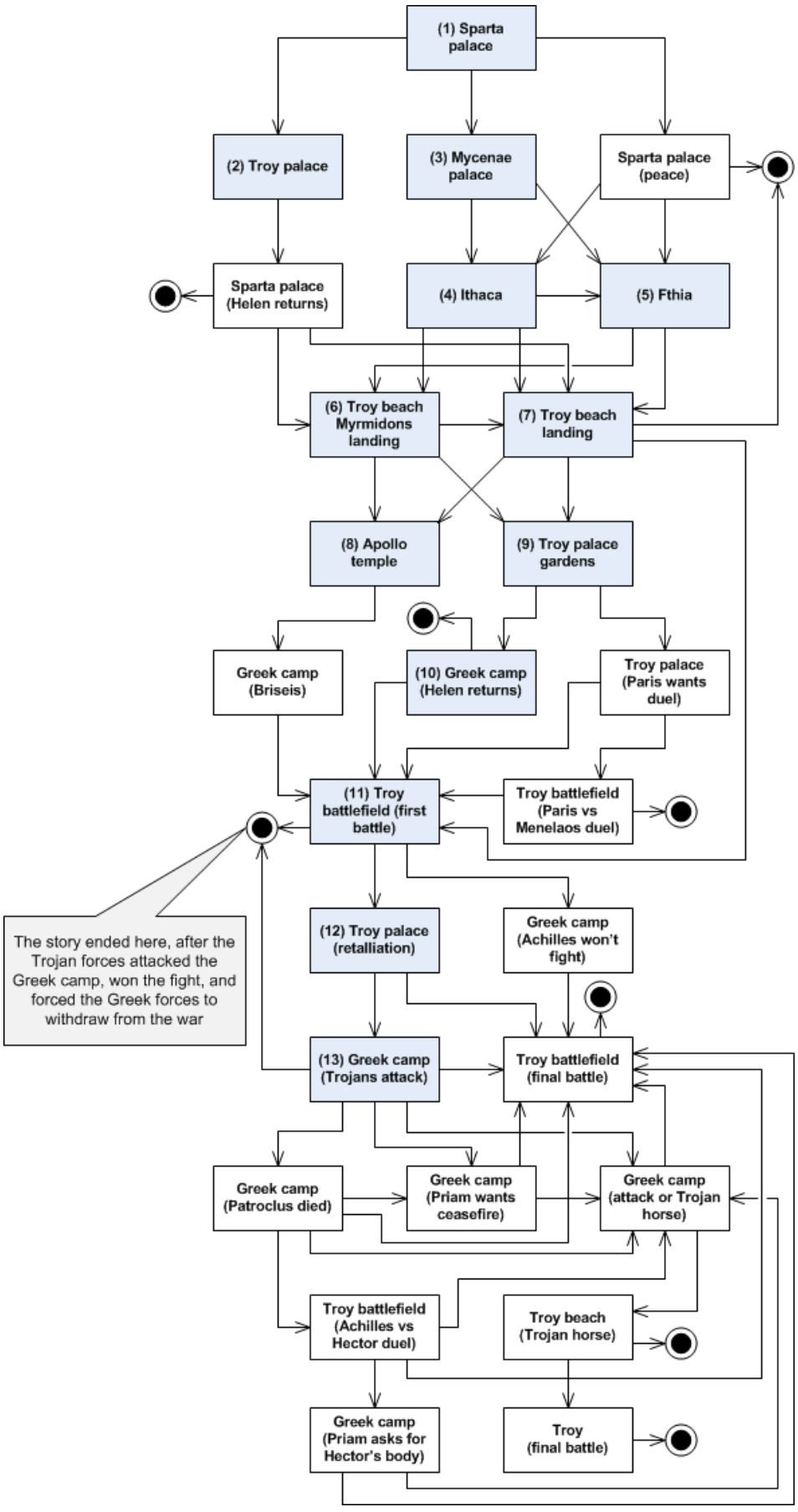

Figure 121: Execution order of levels in Story 4

To gather the data of this evaluation, we run DIEGESIS in a PC featuring 4GB of RAM, and a Quad-core CPU @ 2.83 GHz, running Windows 7.

Table 10 contains the following data for each storyline in total:

- The volume of generated and executed levels;
- the volume of turns;
- the volume of character occurrences in these levels (in this metric, if a character appeared in 3 levels, it will count as 3 characters);
- the volume of unique characters who appeared at least once in the story (in contrast to the previous metric, in this one if a character appeared in 3 levels, it will count as 1 character);
- the volume of choices made (either by the framework or by the player);
- the total successful actions for all the characters;
- the total volume of successful character actions which were not planned by the character, but initiated by another character instead (these actions are not included in the previously mentioned successful actions);
- the volume of failed character actions;
- the total volume of actions (including successful, initiated, and failed actions).

Story data	**Storylines**			
	Story 1	**Story 2**	**Story 3**	**Story 4**
Levels	21	10	6	13
Turns	167	63	41	84
Character occurrences	85	33	16	45
Unique characters	18	10	9	11
Choices	40	11	9	19
Successful actions	173	55	35	80
Initiated actions	93	22	15	36
Failed actions	32	23	7	41
Total actions	298	100	57	157

Table 10: Generic story data results

As we can observe, as a story grows in levels and turns, the number of unique characters and how many time they appear in the executed levels grow as well, something which was expected. The growth is not steady, since levels are different from each other, for example they include different amounts of characters, possible actions, etc. A similar growth can be observed in the volume of choices and actions.

As discussed in section 4.14, facts are potential states of the game world, for example that *Paris* is located in *Troy's* throne room, and operators are potential actions that can be executed resulting to these states, for example that *Paris* can walk from the throne room to his own room. As we also explained in section 4.14, during the loading phase of each level our planner pre-processes the possible facts and operators, creating and storing all their possible combinations (in relation to the characters, locations, items, etc. which are present in a level) and uses this information when performing a planning episode.

Table 11 contains the number of pre-processed facts and operators (we will call them "nodes") for each of the storylines, as well as for all the levels present in the evaluation scenario. These data (which are further illustrated in Figure 122) are an indicator of the complexity of each story, and as we observed before, the complexity rises when a story includes more levels, characters, etc. More specifically, story 1 used 66.14% of the available nodes, story 2 used 15.74% of the available nodes, story 3 used 10.84% of the available nodes, and story 4 used 21.36% of the available nodes.

Pre-processing nodes	**Storylines**				
	All levels	**Story 1**	**Story 2**	**Story 3**	**Story 4**
Fact nodes	4758	3320	1092	701	1451
Operator nodes	17464	11377	2406	1708	3295
Total nodes	22222	14697	3498	2409	4746

Table 11: Pre-processing nodes

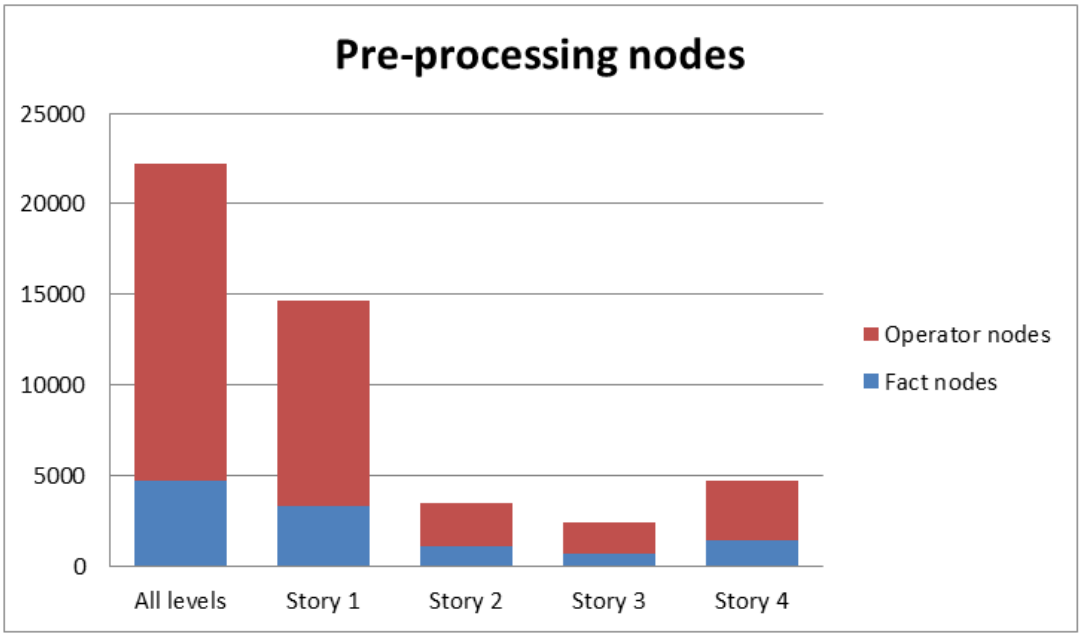

Figure 122: Pre-processing nodes

Table 12 contains information about the planning episodes occurred in each story. The same pattern appeared here as well, where a larger story had more planning episodes than a smaller one. The table also contains the amount of nodes (facts and operators) the planner considered while constructing the planning graph to find a valid plan. In terms of performance, the results are showing that our planner is able to generate a large number of plans in a short amount of time. As an example, there were 102 planning episodes the largest story (1), and the planned needed a total of 0.249 seconds to generate all of them.

Planning data	**Storylines**			
	Story 1	**Story 2**	**Story 3**	**Story 4**
Planning episodes	102	30	18	49
Total planning duration	0.249 sec	0.065 sec	0.055 sec	0.078 sec
Fact nodes	1239	248	165	401
Operator nodes	6514	1699	1262	2522
Total nodes	7753	1947	1427	2923

Table 12: Planning data results

Continuing the discussion about the performance, Table 13 contains further performance data, more specifically:

- The total pre-processing duration, i.e. the combined time the planner needed to generate the nodes we mentioned before, as well as other information (discussed in detail in section 4.14) such as the mutexes between the operators, etc.;
- the total loading duration, i.e. the combined time the World Manager (WM) component needed to initialise a new level (the process in discussed in section 4.5), including the pre-processing time;
- and the total turns duration, i.e. the combined time the WM needed to execute all the turns in the story (including the time the planner needed to generate each of the planning episodes we mentioned earlier).

Once more, the larger a story gets, the more time it needs to be pre-processed, to be loaded, and to be executed.

Performance data	Storylines			
	Story 1	**Story 2**	**Story 3**	**Story 4**
Total pre-processing duration	1.177 sec	0.368 sec	0.255 sec	0.331 sec
Total loading duration	5.676 sec	3.543 sec	2.548	3.734 sec
Total turns duration	7.056 sec	2.367 sec	1.345 sec	4.323 sec

Table 13: Performance data results

Before continuing with the performance discussion we will have a look at the transitioning data included in Table 14. The transitioning process (i.e. moving from a level which ended to a new level that makes sense based on what happened in previously executed levels) is explained in detail in section 4.7. The transitioning data includes the number of transitions successfully performed during the generation of each story, the total time needed to be calculated, the total potential successors that were investigated for their suitability, and the mutual exclusions between them which were calculated.

Transitioning data	Storylines			
	Story 1	**Story 2**	**Story 3**	**Story 4**
Transitions	20	10	6	13
Total transitions duration	3.394 sec	1.858 sec	1.162 sec	2.506 sec
Potential successors	44	24	14	32
Mutual exclusions	39	14	3	17

Table 14: Transitioning data results

Table 15 includes the following information for each of the storylines:

- The average and max level loading duration, i.e. the time the WM component needed to initialise a new level, including the pre-processing time of the Planner;
- the average and max turn execution duration, i.e. the time the WM needed to execute a turn in the story (including the time the planner needed to generate each of the planning episodes we mentioned earlier);
- and the average and max transition duration, i.e. the time the Transitioning Manager (TM) component needed to perform a transition from a level to another.

The maximum combination of durations the player has to wait during the execution of a story is when she instructs the framework to execute the next turn, the turn is

executed and it is the last of a level, so a transition has to be performed, then a loading of the next level, and the execution of the new level's each turn.

Interactivity performance data	Storylines			
	Story 1	Story 2	Story 3	Story 4
Average level loading duration	0.270 sec	0.354 sec	0.425 sec	0.287 sec
Max level loading duration	1.872 sec	2.218 sec	1.888 sec	2.031 sec
Average turn execution duration	0.042 sec	0.038 sec	0.033 sec	0.051 sec
Max turn execution duration	0.235 sec	0.149 sec	0.127 sec	0.240 sec
Average transition duration	0.170 sec	0.186 sec	0.194 sec	0.193 sec
Max transition duration	0.411 sec	0.336 sec	0.267 sec	0.316 sec

Table 15: Interactivity performance data

Therefore, the waiting duration ($d_{waiting}$) before the framework becomes available for interaction again can be calculated as a combination of these durations and can be expressed as the following equation (1):

$$d_{waiting} = 2d_{turn} + d_{loading} + d_{transitioning}$$

Equation 1: Calculation of maximum waiting duration

Using this equation and considering as a worst case scenario that the maximum durations from Table 15 occurred sequentially, we can calculate that the waiting duration that the player had to wait before the system becomes interactive again would be 2.753 seconds for story 1, 2.852 seconds for story 2, 2.409 seconds for story 3, and 2.827 seconds for story 4. These results are displayed in Table 16.

Storylines	Maximum waiting duration
Story 1	2.753 sec
Story 2	2.852 sec
Story 3	2.409 sec
Story 4	2.827 sec

Table 16: Maximum waiting durations

Taking into account that these durations include loading durations (something that gamers are accustomed to) and they represent the worst case scenario, we consider them an indicator that DIEGESIS has a good interactivity performance. The average values of Table 15 shows that as well.

Extending the above, if we combine the total loading, turns, and transitions durations from Table 13 and Table 14, we can calculate the total story duration, represented by equation 2.

$$d_{total} = d_{total-turn} + d_{total-loading} + d_{total-transitioning}$$

Equation 2: Calculation of total story duration

Using that equation, we can calculate that story 1 needed a total of 16.126 seconds to be generated and executed, story 2 a total of 7.768 seconds, story 3 a total of 5.055 seconds, and story 4 a total of 10.563 seconds. These calculations are illustrated in Figure 123.

Based on these results, we believe that DIEGESIS is capable of generating and executing a large and complex story containing several characters (as a reminder, story 1 contains all the main characters, interactions and important scenes appeared in the Troy film) in a very short amount of time, making the framework suitable to be used for the purpose of digital interactive storytelling.

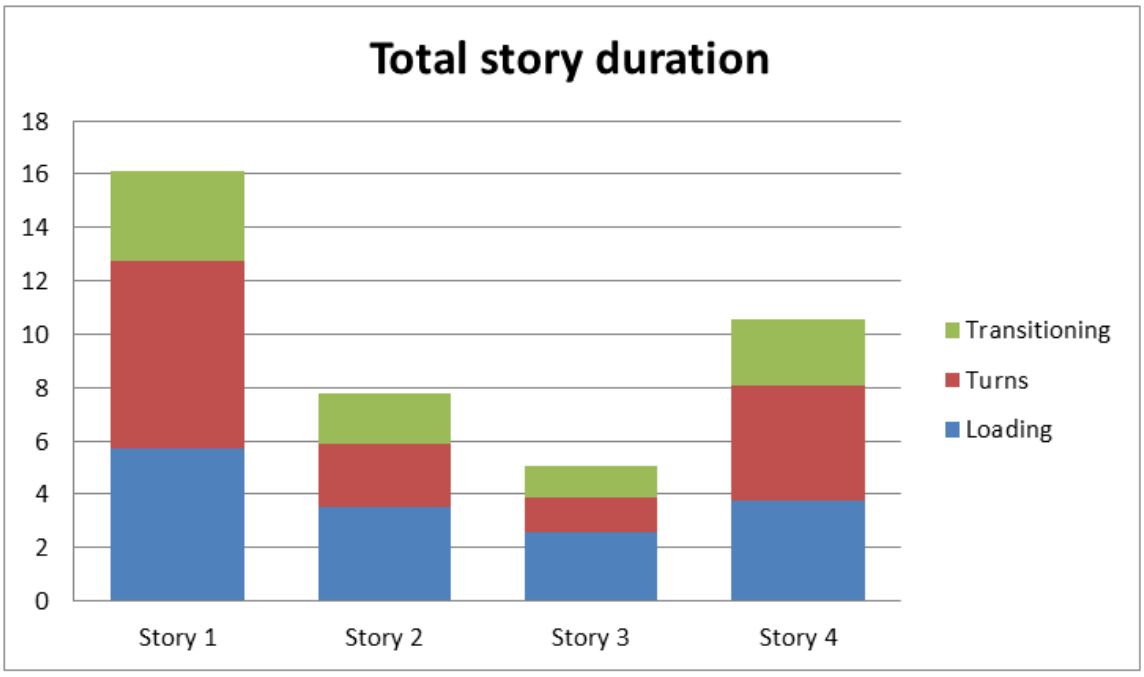

Figure 123: Total story duration

Apart from the combined data for each story, it is worth reviewing some data for a few individual characters of a story, to investigate what happens when a vantage point is selected by the user. As we discussed in section 4.12, the player is able to choose between different vantage points or return to a full story view freely during run-time,

and choosing a vantage point does not affect the outcome of the story, which continues normally –yet invisible if it's unrelated to the chosen character- in the background.

Table 17 includes data for 4 of the characters (*Achilles*, *Hector*, *Odysseus*, and *Helen*), as measured during the generation of story 1. The data includes the following:

- The volume of levels the character appeared in;
- the total successful actions of the character;
- the total volume of successful character actions which were not planned by the character, but initiated by another character instead (these actions are not included in the previously mentioned successful actions);
- the volume of failed character actions;
- and the total volume of actions (including successful, initiated, and failed actions).

Vantage points data	**Characters (Story 1)**			
	Achilles	**Hector**	**Odysseus**	**Helen**
Levels	11	9	7	4
Successful actions	33	38	11	5
Initiated actions	14	8	2	7
Failed actions	3	4	4	1
Total actions	50	50	17	13

Table 17: Vantage points data results

If the player chooses to view the story from the "eyes" of *Achilles*, DIEGESIS will show her 50 actions involving *Achilles* scattered across 11 levels plus any battles which *Achilles* is involved in. In contrast, if the player chooses *Helen's* vantage point, she will only see 13 actions in 4 levels. The levels which will visible in the event that *Achilles'* vantage point is active are illustrated in Figure 124, and those of *Helen's* vantage point in Figure 125.

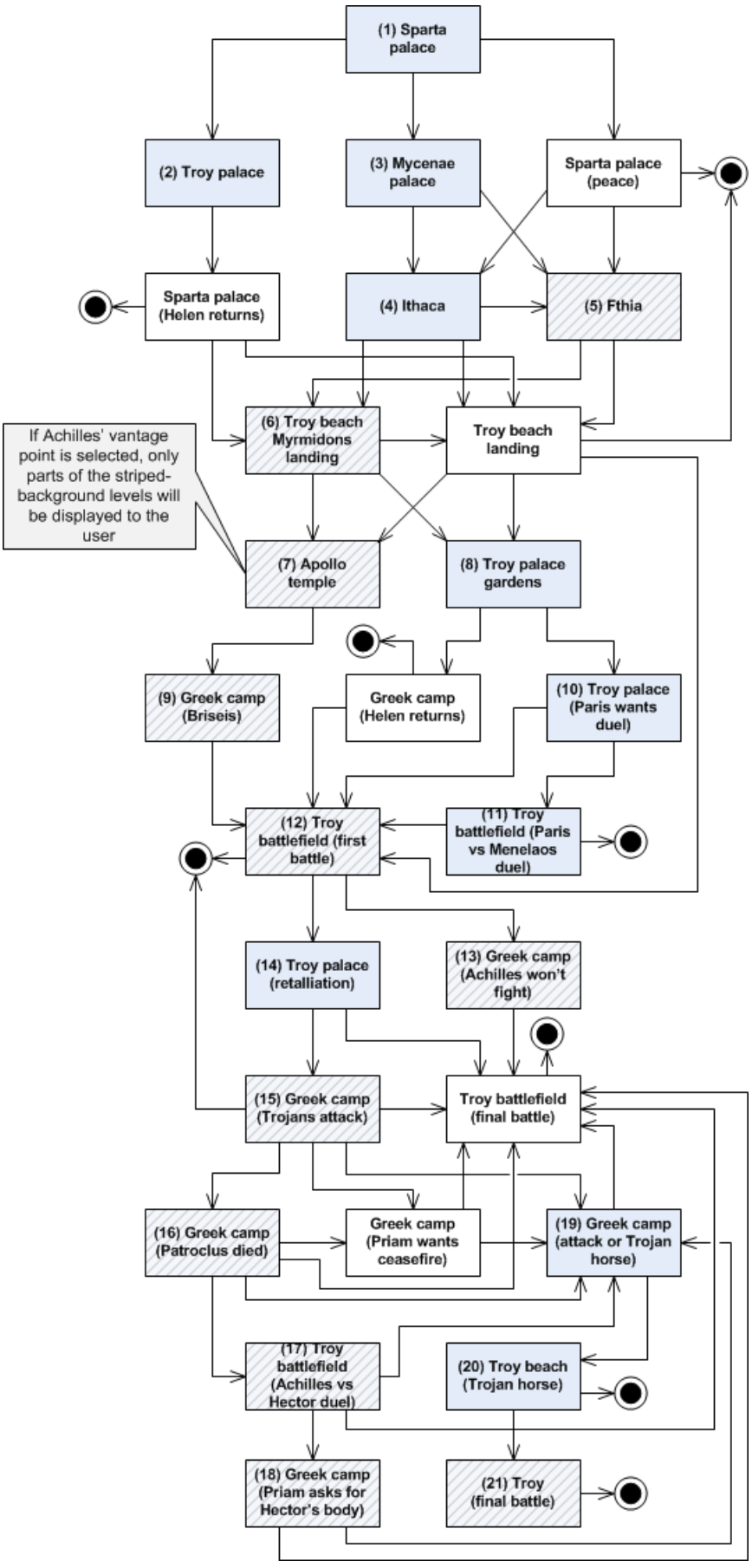

Figure 124: Levels of Achilles' vantage point

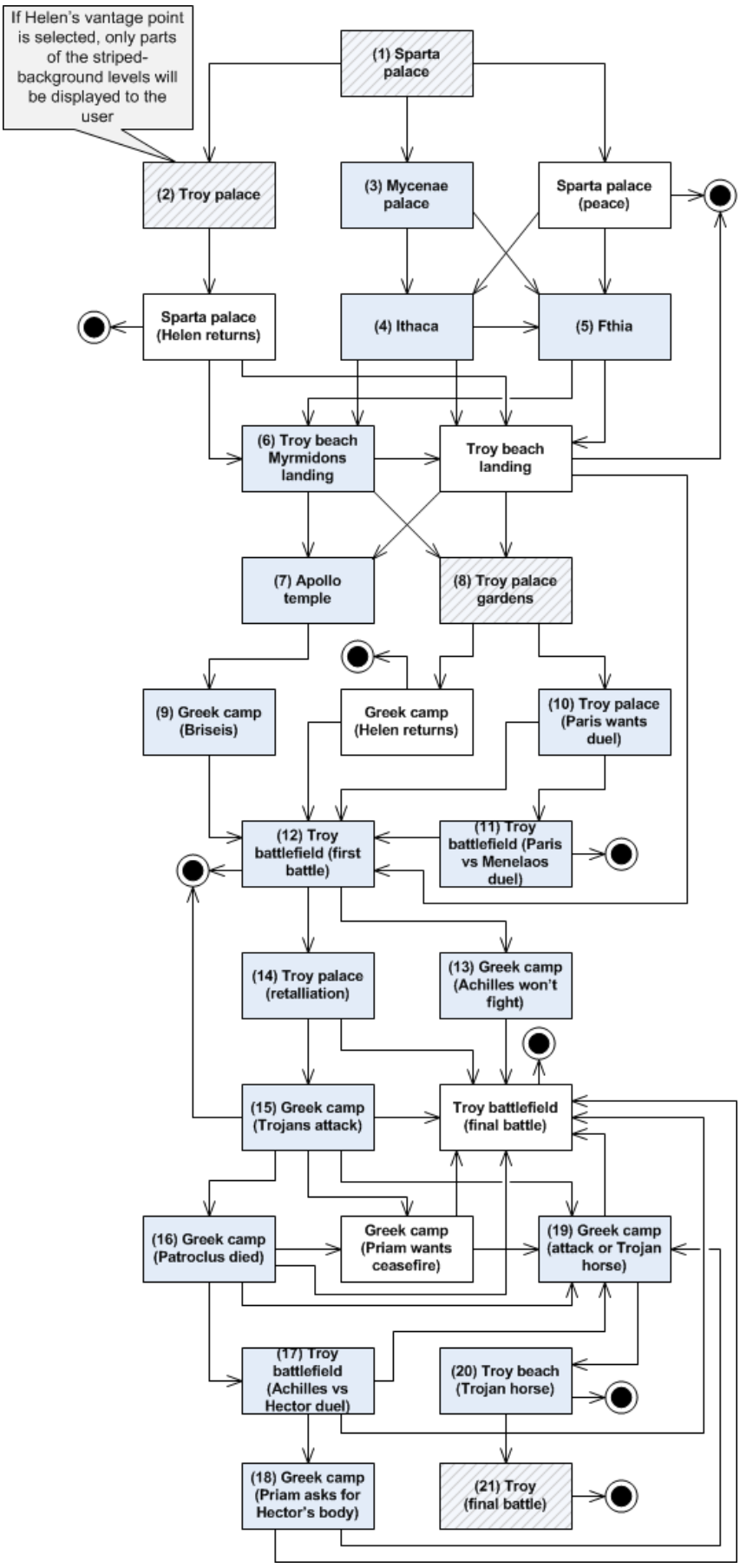

Figure 125: Levels of Helen's vantage point

In this chapter, we provided detailed information about the evaluation scenario that we modelled, showing its potential storylines. We also discussed some of the mechanics that can have an impact on the generated story, and we specified the metrics used in our evaluations. Then, we documented a number of evaluations for the different components of our framework, using the evaluation scenario we presented earlier in previous chapter. In the next chapter, we conclude the thesis and provide some future work ideas

7 CONCLUSION & FUTURE WORK

In this chapter we will conclude the thesis, and provide some future work ideas.

7.1. RESEARCH SUMMARY

The goal of this research was to investigate AI planning and re-planning algorithms and exploit their potential for the field of Digital Interactive Storytelling (DIS), to evaluate their suitability for such systems and to develop new algorithms and storytelling mechanisms to improve them. To this end and to also provide more solutions to DIS research, we decided to specify, design, implement, and evaluate a multi-agent DIS framework, which we titled DIEGESIS.

Apart from a review of the planning and re-planning solutions used by DIS systems, we surveyed and critically assessed a number of DIS systems related to DIEGESIS. This process gave us the opportunity to specify the requirements and the specifications of our multi-agent DIS framework. Based on this set of requirements and specifications, we designed DIEGESIS to be multi-agent, scalable, interactive, modular, and to utilise planning and re-planning techniques to dynamically generate and narrate a story.

Compared to related DIS systems, DIEGESIS' design took many different approaches in key aspects of the framework, which were all governed by our specified needs. For example, although DIS systems traditionally perform either centralised or decentralised planning, DIEGESIS' approach borrows ideas from both approaches, creating a hybrid approach; on the plan generation level it performs decentralised planning, but on the plan execution level our approach borrows the control and coordination concepts from the centralised planning approach. DIEGESIS' agent architecture follows a hybrid approach as well, including elements of reactive, deliberative, and BDI agents.

While many DIS systems allow the player to control and/or make decisions for only one character in the story, in DIEGESIS the player can observe and make choices for any character. Player choices and a number of other mechanics we devised such as a

goal injection mechanism, a battle mechanism, a futile goals module, and the concept of uncertain actions can have an impact on the generated outcome, changing the course of the story.

We also identified that DIS systems utilise either a first-person or a third-person perspective, including vantage points, to present their stories to the player. In its default mode, DIEGESIS presents the generated story as a whole, allowing the player to observe and interact (when is required) with any of the agents present in the story. While these abilities constitute a third-person perspective, we also want to provide the player with a first-person perspective. That's the reason we created the concept of vantage points. If the player selects to view the story from the vantage point of an agent she will view only the story outcome which is related to the chosen agent, and will be available to interact with the story (i.e. make choices) only when an action is related to the story agent. The rest of the story (which is unrelated to the selected agent) will continue normally in the background. The player is able to choose between different vantage points or return to a full story view freely during run-time.

DIEGESIS uses a hybrid story modelling approach, combining both plot-based and character-based elements. The game world is organised in multiple relatively abstract levels which can represent possible parts of a story, and DIEGESIS can perform transitions between them based on a set of rules and what happened previously in the story, to form a valid an interesting story, concept which constitutes a plot-based model. In the level execution though, DIEGESIS implements a character-based model, since each agent has some initial intentions, but is able to operate autonomously and opportunistically to achieve them.

DIEGESIS includes a new planning solution created based on the needs of the DIS field, able to generate plans of actions based on each agent's state and context, considering both the current world state and the available resources. The planning algorithm is based on Graphplan (expanded to include support for several language requirements that we consider valuable for DIS) for solutions expansion, and a backtracking heuristic search for plan extraction, enriched with constraints satisfaction and dynamic opportunistic restart when required. The planning algorithm is also aware of the

available time (duration) an agent/character has for a plan when it is asked to generate one.

DIEGESIS deals with the execution of the agents' plans in a higher level, and when a part of a plan fails, instructs the agent to re-plan based on its current knowledge of the state of the world. Considering that we modelled each agent to act as a real person in the way they generate and try to execute plan, it does not make sense (in our context) to predict and prevent plan failures as some related DIS systems do, since a plan can fail either due to user intervention (which cannot be predicted), or intervention by other characters, or –in some cases– pure chance. In any case, failed plans due to "unpredicted" reasons are realistic and have the potential to enrich a generated narrative.

In our re-planning solution, as we interleave plan generation and plan execution, when a plan fails, we discard the already completed actions and we only re-plan for the failed (and some of the pending) actions of the plan, merging the new partial plan with the unexecuted portion of the original plan.

To evaluate DIEGESIS and its components, we created a large-scale evaluation scenario, where we modelled the story presented in the 2004 film "Troy" (Petersen, 2004), which is based on Homer's epic poem called "Iliad" (Homer, 2003), and using that performed a number of evaluations. Also, since there are no widely accepted metrics to evaluate DIS systems that we could use, we specified some of them based on what consider valuable for the DIS field, a collection which could be potentially used by other DIS systems in the future. These requirements are the following: performance of planning and re-planning solutions; suitability of planning algorithms' features for DIS, performance-based interactivity of the framework; and summarisation metrics. Finally, we disseminated the outcomes of our research, via publishing a number of papers.

7.2. Thesis Summary

In *Chapter 1 (Introduction)*, we discussed the motivation behind our research, the aims and objectives of it, and the research methodology we followed. We also listed a number of contributions to knowledge that our research made.

In *Chapter 2 (Background & Related Work)*, we presented the background of our research area, where we discussed about the fields of Digital Interactive Storytelling and multi-agent systems, we presented some agent architectures, and we discussed about DIS-related as well as multi-agent-related planning and re-planning. We also presented some of the planning algorithms which are typically used in DIS systems along with some of the representation languages used by them. Finally, we presented some examples of re-planning outside of the DIS field, and we surveyed and critically assessed a number of DIS systems, stating their relation to our own work.

In *Chapter 3 (DIEGESIS DIS Framework)*, we documented the requirements and specifications of our multi-agent DIS framework, and then, in *Chapter 4 (Design of the Framework)*, we discussed in detail the design aspect of every component of our framework.

In *Chapter 5 (Implementation)*, we documented all the details about the implementation of the multi-agent DIS framework we discussed in the previous chapters.

In *Chapter 6 (Evaluation)*, we provided detailed information about the evaluation scenario that we modelled, discussed its potential storylines. We also discussed some of the mechanics that can have an impact on the generated story, presented a coordination example, and we specified the metrics used in our evaluations. Finally, we documented a number of evaluations for the different components of our framework, which were performed using the evaluation scenario we presented earlier in the chapter.

7.3. Possible Applications

Apart from using DIEGESIS on its own as described in this thesis, we are also considering alternative routes for it, such as utilising its framework capabilities to connect it to other components and game engines. In this section we will briefly discuss two potential applications involving DIEGESIS as a storytelling framework.

7.3.1. 3D Visualisation of Stories

One potential route for DIEGESIS to be used as a framework would be to be connected with a 3D engine that will enable us to visualise the generated stories and improve the interactivity with the end-user. The resulting application would be able to use DIEGESIS to generate a story as described in previous chapters, and then the 3D engine would be able to visualise the outcome of the generated narrative, presenting it to the user, and allowing her to interact with the system and the story as well. As an example, Figure 126 depicts a scene from Total War: Rome II (SEGA, 2013) video game, showing a Spartan warrior inside Troy after the city was set on fire.

Figure 126: Spartan warrior

For the game engine and implementation development aspect we will use the Homura Game Engine and development Framework (El Rhalibi et al., 2009), developed by our research team. The Homura project's game development framework was created to provide an Open Source (LGPL-Licensed) API for the creation of Java and Open GL based hardware-accelerated 3D games applications, which support cross platform, cross-browser deployment using Java Web Start (JWS) and Next-generation Applet technologies. The framework bundles together several example applications and technical demos, which demonstrate and explain how to implement common games functionality in our applications; An application template, which acts as a great starting point for developing research applications and Homura related games; The APIs of both Homura and the key open-source projects it builds upon including the Java Monkey Engine scenegraph API, jME Physics Library, MD5 Model Importer, GBUI User Interface Libraries and many more; External Tools for the creation of Font Assets, Particle Effects and Levels for the games.

The game engine can be used to create the different 3D scenes to animate and visualise the story as it is generated by the DIEGESIS framework. A member of our research team created a visualisation sample; as an example, Figure 127 depicts a 3D scene involving a panoramic view of the Parthenon of Goddess Athena.

Figure 127: Parthenon of Greek Goddess Athena

Figure 128, describes a scene where Achilles watches the ships of the Greek army preparing to sail to Troy.

Figure 128: Achilles Watching Ships Preparing for War

Figure 129 depicts a scene of Achilles meeting Greek Goddess Athena before leaving to Troy, with some of the Greek army ships preparing to depart on the background.

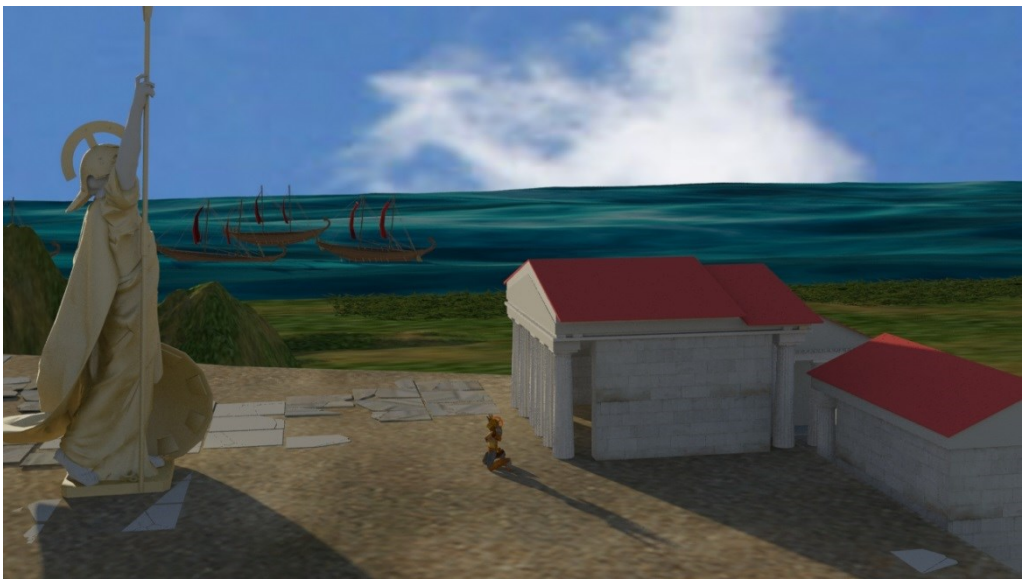

Figure 129: Achilles meeting Greek Goddess Athena

7.3.2. VIRTUAL STORYTELLER

Another potential route for our system is to interface it with a 3D character animation framework to create a conversational avatar framework for Digital Interactive Storytelling, i.e. 3D facial model animation which will act as a narrator for the stories which our framework produces. We designed, documented, and proposed this application in (Duarte et al., 2013). In the proposed system, DIEGESIS will be interfaced with Charisma (El Rhalibi et al., 2010), an MPEG-4 based facial animation framework, which will enable a 3D character to narrate the generated story (Figure 130), or it could also be used for in-story character dialogues.

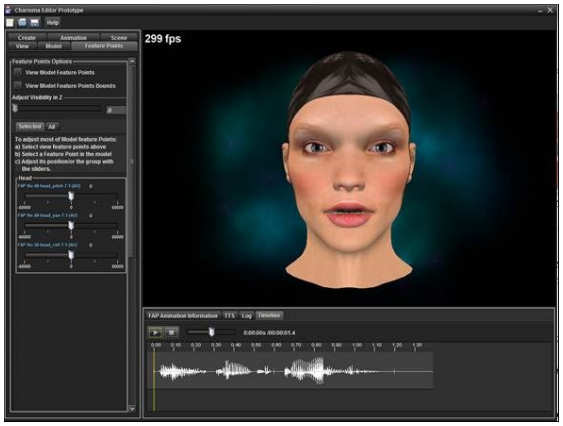

Figure 130: The Charisma interface

DIEGESIS is capable of generating interactive, dynamic, emergent, and consistent narratives, however the language of the outcome story presented to the user is not

rich and expressive. Therefore, it seems as a good fit to use an intermediate system that generates natural language, such as those introduced in (Theune et al., 2006, Theune et al., 2007), and presents it to the user by text and speech, using the coarticulation system in Charisma, providing a more natural interface to the user.

Using the abilities of DIEGESIS, the proposed framework generates a narrative (Figure 131 and Figure 132), based on the initial story model provided by the storyteller, the interaction between the agents, and the choices of the user.

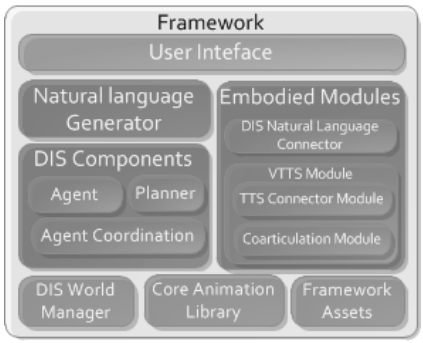

Figure 131: Framework layers for facial animation and DIS

Once a part of the narrative is generated, it will be converted to natural language, and handed to the coarticulation embodied module. Further details about the coarticulation module (which are out of the scope of this thesis) can be found in (Duarte et al., 2013).

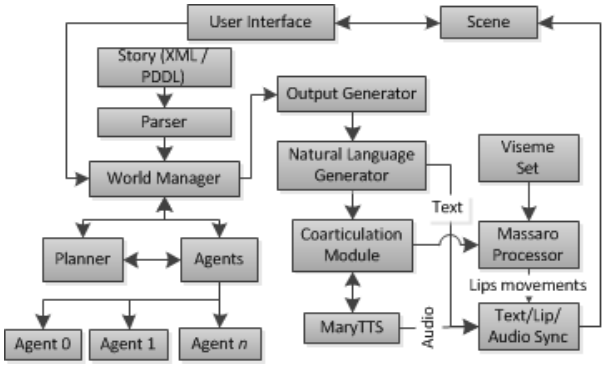

Figure 132: The flow of a story used as input

The current framework provides a richer user experience, by using a 3D narrator to narrate the generated story, and by generating a richer and expressive language for the narrative, when compared with the previous outcome of the framework.

Figure 133 illustrates an example of an agent's possible plan, based on the scenario described in section 6.2. The agent represents the character Achilles in the story. Using a dialogue generation solution, we will generate an expressive narration based on the outcome which DIEGESIS is generating, so it can be imported to Charisma.

```
PICK-UP ACHILLES SWORD ACHILLES-TENT
PICK-UP ACHILLES SHIELD ACHILLES-TENT
WEAR ACHILLES ARMOUR ACHILLES-TENT
GO-TO ACHILLES ACHILLES-TENT BATTLEFIELD
ATTACK ACHILLES HECTOR BATTLEFIELD
KILL ACHILLES HECTOR BATTLEFIELD
PICK-UP ACHILLES HECTOR BATTLEFIELD
GO-TO ACHILLES BATTLEFIELD GREEK-CAMP
DROP ACHILLES HECTOR GREEK-CAMP
PICK-UP ACHILLES PATROCLUS GREEK-CAMP
GO-TO ACHILLES BEACH
PREPARE-FOR-FUNERAL GREEK ACHILLES BEACH
PERFORM-FUNERAL GREEK ACHILLES BEACH
GO-TO ACHILLES BEACH ACHILLES-TENT
ALLOW-TO-TAKE ACHILLES PRIAM HECTOR
```

Figure 133: Achilles' combined plan

An example of the narrative that could be generated by a dialog generation solution based on the outcome illustrated in Figure 133, is the following: "*Achilles picked up his sword and shield, and wore his armour. Afterwards, he left his tent, heading for the battlefield. There, he attacked Hector and killed him. Achilles picked up Hector's body, returned to the Greek camp and dropped Hector's body. He picked up Patroklus' body and walked to the beach. There, Achilles prepared for Patroclus' funeral, and then performed it. He returned to his tent, where Priam was waiting and, after a discussion, Achilles allowed Priam to take Hector's body.*"

As illustrated in Figure 134, once the natural language based story is generated using the tools available in DIEGESIS in conjunction with a dialogue generation solution, it is then sent to Charisma for further processing and to be presented to the user. A further evaluation of the proposed application can be found in (Duarte et al., 2013).

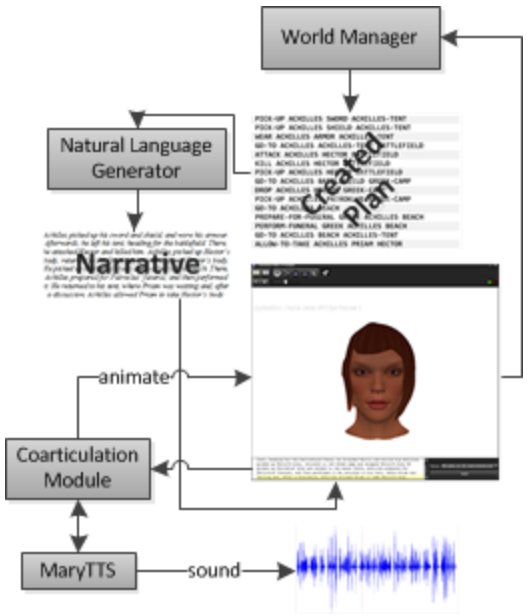

Figure 134: Information flow to create a narrative in the framework

7.4. Future Work

There are several avenues to extend this research and explore some areas that we didn't explore, and to add some features that were not necessary to implement during our research.

7.4.1. Authoring Tools

As we discussed in section 4.18, to make the authoring process easier for the storyteller, we are using a PDDL editor created by (Cooper, 2011), and we designed an XML editor as an extension to the PDDL one, so the storyteller can be able to create easier all the XML nodes needed to complete the modelling of a story. Since implanting authoring tools wasn't the aim of our research we didn't implement the XML editor we designed, but implementing it in the future would benefit the storytellers who will use DIEGESIS.

7.4.2. Emotions Manager Component

A potential feature that we considered at some point during our research was to design and implement an emotions manager component, which will closely work with our framework's agents. A potential implementation could be to add emotions to the agents in related to other agents, and then model the story's actions to affect certain

emotions positively or negatively. Then, these emotion values can be used to affect the choices that agents make (when they are made by the framework and not the player) and to trigger goal injection rules based on emotion thresholds.

We didn't go into that route because we decided that it would require a lot of micro-management to get it working in an acceptable way and it would derail us from our research aims and objective. It has the potential to make the generated narrative and the relations between the characters richer though, so it should be considered as a potential extension of the framework.

7.4.3. Improvement of Planning Algorithm's Pre-processing

During the implementation phase of our research, we have identified that the bottleneck of DIEGESIS' planner is its pre-processing process, i.e. the time needed by the planner to pre-generate nodes that are later used in the actual planning and re-planning process.

Our planner only needs to pre-process the information of a level once and then the pre-processed information can be reused in any planning and re-planning episode of that level, an approach which has a good performance as we discovered in the evaluation we documented in section 6.5.2. Therefore, any delay due to the pre-processing will only affect the loading time of the level, and not the experience of the player while executing the level.

Furthermore, as we discovered in the evaluation we performed and documented in section 6.5.5, DIEGESIS is capable of generating and executing a large and complex story containing several characters in a very short amount of time. Nevertheless, it is something that we could investigate further in the future and try to improve the performance of the algorithm.

7.4.4. Durative Actions

As we discussed in section 4.14, for simplicity reasons since it was easier to evaluate the generated plans of each character in the story and investigate how they affect each other, we modelled the actions that an agent can execute to be atomic and

unable to overlap each other during the course of several turns. Therefore, we decided that instead of using PDDL's durative actions which are more complex for the storyteller to construct, to simply specify the total duration of an action (in seconds) in XML. We consider the incorporation of durative actions into the plan execution phase as an interesting prospect for our framework, especially since we have already modelled our new planning solution to take into account the duration of an action and discard goals that are not such important if a generated plan does not meet its time constraints.

7.4.5. Possible Applications

As we discussed in section 4.13, we initially decided that based on the focus of our research, a graphical user interface (GUI) would not be developed, as being out of scope of this project. Although we did end up creating a GUI, it is mainly focused on debugging, providing relevant information about what is happening in the framework and displaying the generated story in a visually limited way. Therefore, in section 7.3 we described two possible applications utilising DIEGESIS as a DIS framework which will be visually pleasing and easy to use by the player.

7.4.6. Application Programming Interface (API)

Although we have designed DIEGESIS as a framework and –as we discussed in the previous section– we want it to be able to connect with other systems to form different DIS applications, we didn't implement an API for it, something which will make it easier to be integrated to other systems. The de-coupled nature of our design though will make it easy to replace the User Manager component with an appropriate API if we decide to implement one in the future.

7.4.7. Further Framework Evaluations

Apart from the evaluation metrics we discussed in section 6.4 used in the evaluations we documented in section 6.5, there are a couple of other metrics and potential evaluations that can be performed in the future:

- *Plan quality over optimality*: Optimality of a plan is not necessary in DIS. The ultimate goal of a DIS system is to create a good story so (even if it's hard to measure it) the quality of the narrative is the most important metric in such a system. A possible way to measure the quality of the generated narrative is to present a story created by the system to a group of people and create a questionnaire that they will complete to evaluate the quality of the story.
- *Minimising Disruption in Re-planning*: When performing re-planning to an existing plan, the disruption made to it should be kept to a minimum. A metric to measure is the number of changes performed in an existing plan and the extent of these changes, to measure the disruption in it, quantified by a disruption metric.

7.4.8. Story Modelling

As the specifications of 3.2 dictated, we aimed to design the framework in the most abstract way we could, to be able to be used with any kind of story instead of being highly coupled with one. We believe that we achieved that, since with the exception of the Battle Manager component (which can be turned off) all the other components of the system and the modelling rules of it are not highly coupled to a specific kind of story. But, since we did not have a chance to evaluate that notion by building another type of story (e.g. a science fiction-based scenario), it would be interesting to do that in the future.

Also, another interesting route would be to replicate stories produced for other DIS systems, after they are ported into our modelling requirements, to investigate what kind of outcome DIEGESIS is capable of producing compared to these systems, based on the same story model.

REFERENCES

ARINBJARNAR, M., BARBER, H. & KUDENKO, D. 2009. A Critical Review of Interactive Drama Systems. *AISB 2009 Symposium. AI & Games.*

BARBER, H. & KUDENKO, D. Dynamic Generation of Dilemma-based Interactive Narratives. 3rd Conference on Artificial Intelligence and Interactive Digital Entertainment, 2007.

BARBER, H. & KUDENKO, D. 2009. Generation of Adaptive Dilemma Based Interactive Narratives. *IEEE Transactions on Computational Intelligence and AI in Games.*

BARROS, L. M. & MUSSE, S. R. 2005. Introducing Narrative Principles Into Planning-Based Interactive Storytelling. *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology, ACE '05.*

BARROS, L. M. & MUSSE, S. R. 2007a. Improving Narrative Consistency in Planning-Based Interactive Storytelling. *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 07).*

BARROS, L. M. & MUSSE, S. R. 2007b. Planning Algorithms for Interactive Storytelling. *ACM Computers in Entertainment,* 5.

BARTÁK, R., SALIDO, M. A. & ROSSI, F. 2010. New Trends in Constraint Satisfaction, Planning, and Scheduling: A survey. *The Knowledge Engineering Review,* 25**,** 249-279.

BARTOLD, T. & DURFEE, E. 2003. Limiting Disruption in Multi-agent Re-planning. *Proceedings of the second international joint conference on Autonomous agents and multi-agent systems, AAMAS '03.*

BLUM, A. & FURST, M. 1997. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence,* 90**,** 281-300.

BONET, B. & GEFFNER, H. 2001. Planning as heuristic search. *Artificial Intelligence,* 129**,** 5-33.

CARMICHAEL, G. & MOULD, D. 2014. A Framework for Coherent Emergent Stories. *Foundations of Digital Games.*

CAVAZZA, M., CHARLES, F. & MEAD, S. J. 2002. Interacting with Virtual Characters in Interactive Storytelling. *Proceedings of the first international joint conference on Autonomous agents and multi-agent systems, AAMAS '02.*

CHANG, H.-M. & SOO, V. W. Simulation-based narrative generation with a theory of mind. 4th Conference on Artificial Intelligence and Interactive Digital Entertainment, 2008. 16-21.

CHANG, H.-M. & SOO, V. W. 2009. Planning-Based Narrative Generation in Simulated Game Universes. *IEEE Transactions on Computational Intelligence and AI in Games.*

CHARLES, F., LOZANO, M., MEAD, S. J., BISQUERRA, A. F. & CAVAZZA, M. 2003. Planning Formalisms and Authoring in Interactive Storytelling. *Proceedings of the 1st International Conference on Technologies for Interactive Digital Storytelling and Entertainment.*

COLES, A. I. & SMITH, A. J. 2004. Marvin: Macro-actions from reduced versions of the instance. *Fourth International Planning Competition Booklet (IPS'04), ICAPS.*

COOPER, S. 2011. *DISE: A Game Technology-based Digital Interactive Storytelling Framework.* PhD, Liverpool John Moores University.

DOYLE, J. 1996. Toward Rational Planning and Re-planning. *Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative.*

DUARTE, R., GOUDOULAKIS, E., EL RHALIBI, A. & MERABTI, M. 2013. A Conversational Avatar Framework for Digital Interactive StoryTelling. *The 14th Annual PostGraduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet 2013).*

EL RHALIBI, A., CARTER, C., COOPER, S., MERABTI, M. & PRICE, M. 2010. Charisma: High Performance Web Based MPEG-4 Compliant Animation Framework. *ACM Computers in Entertainment,* 8.

EL RHALIBI, A., GOUDOULAKIS, E. & MERABTI, M. 2012. DIS Planning Algorithms Evaluation. *4th IEEE International Workshop on Digital Entertainment, Networked Virtual Environments, and Creative Technology.*

EL RHALIBI, A., MERABTI, M., CARTER, C., DENNETT, C., COOPER, S., SABRI, M. A. & FERGUS, P. 2009. 3D Java Web-Based Games Development and Deployment. *International Journal on Information and Communication Technologies,* 2**,** 221-230.

FOWLER, M. 2003. *UML Distilled Third Edition a Brief Guide to the Standard Object Modeling Language*, Addison-Wesley.

FOX, M., GEREVINI, A., LONG, D. & SERINA, I. 2006. Plan Stability: Replanning versus Plan Repair. *International Conference on AI Planning and Scheduling (ICAPS).*

FOX, M. & LONG, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research, JAIR,* 20**,** 61-124.

GEREVINI, A. & LONG, D. 2005. BNF Description of PDDL 3.0.

GEREVINI, A., SAETTI, A., SERINA, I. & TONINELLI, P. 2004. Planning in PDDL2.2 domains with LPG-TD. *Fourth International Planning Competition Booklet (IPS'04), ICAPS.*

GHALLAB, M., HOWE, A., KNOBLOCK, C., MCDERMOTT, D., RAM, A., VELOSO, M., WELD, D. & WILKINS, D. 1998. PDDL - The Planning Domain Definition Language. *Tech Report DCS TR-1165.* Yale Center for Computational Vision and Control.

GOUDOULAKIS, E., EL RHALIBI, A., MERABTI, M. & TALEB-BENDIAB, A. 2011. Evaluation of Planning Algorithms for Digital Interactive Storytelling. *The 12th Annual PostGraduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet 2011).*

GOUDOULAKIS, E., EL RHALIBI, A., MERABTI, M. & TALEB-BENDIAB, A. 2012a. Framework for multi-agent planning and coordination in DIS. *Proceedings of the Workshop at SIGGRAPH Asia (WASA '12).*

GOUDOULAKIS, E., EL RHALIBI, A., MERABTI, M. & TALEB-BENDIAB, A. 2012b. Opportunistic Multi-Agent Digital Interactive Storytelling System. *The 13th Annual PostGraduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet 2012).*

GOUDOULAKIS, E., EL RHALIBI, A., MERABTI, M. & TALEB-BENDIAB, A. 2013. Re-planning in Digital Interactive Storytelling. *The 14th Annual PostGraduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet 2013).*

GOUDOULAKIS, E., EL RHALIBI, A., MERABTI, M. & TALEB-BENDIAB, A. 2014. DIEGESIS - A Novel Multi-Agent Planning System for Digital Interactive Storytelling. *ACM Journal Computer in Entertainment,* Vol. 3 (in print).

HOFFMANN, J. 2001. FF: The Fast-Forward Planning System. *AI Maagazine,* 22.

HOFFMANN, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Artificial Intelligence Research,* 20**,** 291-341.

HOMER 2003. *The Iliad*, Penguin Books.

ILGHAMI, O. & NAU, D. S. 2003. A general approach to synthesize problem-specific planners. *Tech. Rep. CS-TR-4597, UMIACS-TR-2004-40.* University of Maryland.

KARLSSON, B., CIARLINI, A. E. M., FEIJO, B. & FURTADO, A. L. 2007. Applying the Plan-Recognition / Plan-Generation Paradigm to Interactive Storytelling: The LOGTELL Case Study. *Monografias em Ciencia da Computacao,* 24/07, September.

KAUTZ, H. & SELMAN, B. Planning as satisfiability. Tenth European Conference on Artificial Intelligence (ECAI'92), 1992. 359-363.

KLUSCH, M., GERBER, A. & SCHMIDT, M. 2005. Semantic Web Service Composition Planning with OWLS-Xplan. *AAAI Fall Symposium on Semantic Web and Agents.*

KLUSCH, M. & RENNER, K.-U. 2006. Fast Dynamic Re-planning of Composite OWL-S Services. *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology Workshops (WI-IAT 2006 Workshops).*

LEKAVY, M. & NAVRAT, P. Expressivity of STRIPS-Like and HTN-Like Planning. 1st KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications (KES-AMSTA '07), 2007. 121-130.

MATEAS, M. & STERN, A. 2003. Façade: An Experiment in Building a Fully-Realised Interactive Drama. *In Game Developers Conference Proceedings (GDC '03) (March 2003).*

MERABTI, M., EL RHALIBI, A., ALCANTARA MELENDEZ, D., PRICE, M. & SHEN, Y. 2008. Interactive Storytelling: Approaches and techniques to achieve dynamic stories. *International Journal Transactions on Entertainment I, LNCS 5080, Springer-Verlag, Berlin, Heidelberg.*

NAREYEK, A., FREUDER, E. C., FOURER, R., GIUNCHIGLIA, E., GOLDMAN, R. P., KAUTZ, H., RINTANEN, J. & TATE, A. 2005. Constraints and AI Planning. *Intelligent Systems, IEEE,* 20**,** 62-72.

NILSSON, N. J. & FILES, R. E. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence,* 2**,** 189-208.

ORKIN, J. 2006. Three States and a Plan: The A.I. of F.E.A.R. *Game Developers Conference (GDC).*

PAUL, R., CHARLES, D., MCNEILL, M. & MCSHERRY, D. MIST: An Interactive Storytelling System with Variable Character Behavior. Third joint conference on Interactive digital storytelling, 2010. Springer Berlin Heidelberg, 4-15.

PAUL, R., CHARLES, D., MCNEILL, M. & MCSHERRY, D. Adaptive Storytelling and Story Repair in a Dynamic Environment. 4th international conference on Interactive Digital Storytelling, 2011. Springer-Verlag Berlin, Heidelberg, 128-139.

PAUL, R. J. S., MCNEILL, M. D. J., CHARLES, D. K., MCSHERRY, D. M. G. & MORROW, P. J. 2009. Real-time Planning for Interactive Storytelling. *Proceedings of the 9th Irish Workshop on Computer Graphics.*

PEDNAULT, E. ADL: Exploring the middle ground between STRIPS and the situation calculus. 1st international conference on Principles of Knowledge Representation and Reasoning, 1989. 324-332.

PELLIER, D. 2009. *PDDL4J* [Online]. Available: https://github.com/gerryai/PDDL4J [Accessed 20th May 2014].
Troy, 2004. Directed by PETERSEN, W.
RIEDL, M., SARETTO, C. J. & YOUNG, R. M. Managing Interaction Between Users and Agents in a Multi-agent Storytelling Environment. Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '03), 2003.
ROBERTS, D. L. & ISBELL, C. L. 2008. A Survey and Qualitative Analysis of Recent Advances in Drama Management. *International Transactions on Systems Science and Applications, Special Issue on Agent Based Systems for Human Learning,* 3**,** 61-75.
RUSSELL, S. & NORVIG, P. 2010. *Artificial Intelligence: A Modern Approach, Third Edition*, Pearson Education Inc.
SEGA. 2013. *Total War: Rome II* [Online]. Available: https://www.totalwar.com/en_gb/rome2 [Accessed 1st June 2014].
SHEN, Y. 2006. *Exploiting Timed Coloured Petri net in Control and Analysis of Graphplan.* Liverpool John Moores University.
SPIERLING, U. 2009. Conceiving Interactive Story Events. *Proceedings of the 2nd Joint International Conference on Interactive Digital Storytelling, ICIDS '09.*
TEESSIDEUNIVERSITY. 2010. *IRIS Wiki: IS Systems* [Online]. Available: http://tecfalabs.unige.ch/mediawiki-narrative/index.php/IS_Systems (mirror) [Accessed 25th May 2014].
THEUNE, M., MEIJS, K., HEYLEN, D. & ORDELMAN, R. 2006. Generating expressive speech for storytelling applications. *IEEE Transactions on Audio, Speech, and Language Processing,* 14.
THEUNE, M., SLABBERS, N. & HIELKEMA. The automatic generation of narratives. 17th Meeting of Computational Linguistics in the Netherlands (CLIN17), 2007. 131-146.
THUE, D., BULITKO, V., SPETCH, M. & WASYLISHEN, E. 2007. Interactive Storytelling: A Player Modelling Approach. *Proceedings of the third Artificial Intelligence and Interactive Digital Entertainment conference (AIIDE07).*
VAN DER KROGT, R. & DE WEERDT, M. 2005. Plan Repair as an Extension of Planning. *International Conference on Automated Planning and Scheduling.*
VLACHAVAS, I., KEFALAS, P., VASILIADIS, N., KOKKORAS, F. & SAKELLARIOU, E. 2005. *Artificial Intelligence (Τεχνητή Νοημοσύνη), Second Edition (Β Έκδοση)*, Gartaganis (Γαρταγάνης).

YOUNG, R. M. & RIEDL, M. O. Towards an architecture for intelligent control of narrative in interactive virtual worlds. ACM International Conference on Intelligent User Interfaces, 2003. 310-313.

ZHANG, J. F., NGUYEN, X. T. & KOWALCZYC, R. 2007. Graph-based Multi-agent Replanning Algorithm. *6th international joint conference on Autonomous agents and multiagent systems (AAMAS '07).*

APPENDICES

APPENDIX A: USE OF UML IN THE THESIS

In this thesis, we use a set of UML diagrams to describe use cases in the form of use case diagrams (in chapter 3), processes of various components of the framework in the form of activity diagrams (in chapter 4), classes and operations of the framework in the form of class diagrams (in chapter 5), and interactions between the components of the framework in the form of sequence diagrams (in chapter 4).

To design these diagrams, we used the notation and recommendations made by (Fowler, 2003). When a diagram does not clearly states that is a use case, activity, sequence, or class diagram, then it is not a UML diagram.

The conventions that we have made to the standard UML notation are the following:

- In some class diagrams, when a class has been described in detail as part of a previous class diagram and appears again in another located after the one it was described into, we are omitting its description (i.e. its properties and operations) and keep only its name for the sake of simplicity.
- Although most of the framework's components consist of a number of classes, in all of the sequence diagrams located in the design chapter, we consider each component as one entity, and represent them as one instance of the component. That happens for the sake of simplicity, but also because the actual brake down of each component's classes is located in the implementation chapter (which is located after the design chapter) and it was suitable to describe the interactions and message flow between the components as part of the design process.