Generador de historias basado en agentes

Iván Manuel Laclaustra Yebes
Jose Luis Ledesma López

Grado en ingeniería informática, Universidad Complutense de Madrid

Trabajo de fin de grado en computación

Madrid, 23 de junio de 2014

Director: Gonzalo Méndez Pozo
Codirector: Pablo Gervás Gómez-Navarro

Página de autorización

Por el presente texto se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código fuente generado y/o el prototipo desarrollado.

Madrid, 23 de Junio de 2014

Fdo.: Iván Manuel Laclaustra Yebes

Fdo.: Jose Luis Ledesma López

Agradecimientos

Este proyecto ha sido parcialmente respaldado por los proyectos WHIM 611560 y PROSECCO 600653 fundados por la comisión Europea, Programa Marco de Trabajo 7, el tema ICT y el programa FET de Futuras Tecnologías Emergentes.

Índice

ÍNDICE .... **VII**

ÍNDICE DE FIGURAS .... **IX**

ABSTRACT .... **XI**

KEYWORDS XI

RESUMEN .... **XII**

PALABRAS CLAVE XII

CAPÍTULO 1: INTRODUCCIÓN **1**

1.1 INTRODUCTION 1
1.2 INTRODUCCIÓN 2
1.3 MOTIVACIONES 3
1.4 OBJETIVOS 3

CAPÍTULO 2: TRABAJO RELACIONADO **5**

2.1 AGENTES SOFTWARE 5
Estándares FIPA *7*
2.2 PLANIFICACIÓN 8
2.3 SISTEMAS 9
Novel Writer *9*
Talespin *10*
Author *11*
Universe *12*
Minstrel *13*
Mexica *14*
Virtual Storyteller *15*
Fabulist *16*
Thespian *18*
Narrador en nn *19*
Party Quirks *19*

CAPÍTULO 3: ARQUITECTURA **21**

3.1 MUNDO 22
3.2 SISTEMA MULTIAGENTE 22
3.3 PLANIFICACIÓN 22
3.4 LOGGER 23

CAPÍTULO 4: DISEÑO E IMPLEMENTACIÓN **25**

4.1 SISTEMA MULTIAGENTE 25
4.1.1 Plataforma *25*
4.1.1.1 Comportamientos 25
4.1.1.2 Estados de los agentes 25
4.1.1.3 Mensajes 26
4.1.2 Agentes *27*
4.1.2.1 Agentes como personajes 27
Personajes de la historia 29

Caballero ... 30
Dragón ... 31
Princesa ... 32
Rey ... 33
4.1.2.2 Agentes conductores ... 35
Agente director ... 35
Agente mundo ... 36
4.1.3 Ficheros de configuración ... 38
4.1.3.1 Objetivos de los personajes ... 38
4.1.3.2 Localización y Mapa ... 39
4.2 MUNDO ... 40
4.2.1 Estado ... 41
4.2.2 Objetos ... 41
4.3 PLANIFICADOR ... 42
4.3.1 JavaFF ... 42
4.3.2.1 Heurística ... 43
4.3.2.2 Búsqueda ... 43
4.3.2 Acciones ... 44
4.4 LOGGER ... 46
4.5 FLUJO DE EJECUCIÓN ... 46
CAPÍTULO 5: RESULTADOS ... 49
Ejemplo 1 ... 49
Ejemplo 2 ... 49
Ejemplo 3 ... 50
5.1 CONCLUSIONES ... 51
5.2 DISCUSIÓN ... 52
5.3 TRABAJO FUTURO ... 53
Añadir paralelismo ... 53
Cambio de dominio ... 54
Expandir el mapa ... 54
Expandir personajes ... 55
Interactividad ... 55
Expansión del agente director ... 55
5.4 APORTACIONES DE LOS MIEMBROS ... 56
5.4.1 Parte Jose Luís Ledesma López ... 56
5.4.2 Parte de Iván Manuel Laclaustra Yebes ... 58
5.5 CONCLUSIONS ... 60
APÉNDICE A: ARTÍCULO PUBLICADO ... 61
BIBLIOGRAFÍA ... 67

Índice de figuras

FIGURA 1: ESTRUCTURA DE UNA PLATAFORMA MULTIAGENTE 8
FIGURA 2: EJEMPLO DE PLANIFICACIÓN 8
FIGURA 3: PLAN PARA UNA HISTORIA DE FABULIST 18
FIGURA 4: ARQUITECTURA DEL PROYECTO 21
FIGURA 5: CICLO DE VIDA DE UN AGENTE SEGÚN LOS ESTÁNDARES FIPA 26
FIGURA 6: DIAGRAMA DE CLASES DE LOS PERSONAJES 28
FIGURA 7: DIAGRAMA DE CLASES DEL CABALLERO 31
FIGURA 8: DIAGRAMA DE CLASES DEL DRAGÓN 32
FIGURA 9: DIAGRAMA DE CLASES DE LA PRINCESA 33
FIGURA 10: DIAGRAMA DE TRANSICIÓN DEL AUTÓMATA DEL REY. 34
FIGURA 11: DIAGRAMA DE CLASES DEL REY 35
FIGURA 12: DIAGRAMA DE CLASES DEL AGENTE DIRECTOR 36
FIGURA 13: DIAGRAMA DE CLASES DEL AGENTE MUNDO 38
FIGURA 14: DIAGRAMA DE ACTIVIDAD DE UNA HISTORIA BÁSICA 47

Abstract

In this project we discuss about how we can generate computational creativity. This is a complex work, as the concept of creativity itself is ambiguous, inasmuch as there are things such as talent involved in it, which are difficult to explain themselves. Following the domain of our application, we concentrate in the scope of narrative and story generation. We'll see how we can generate stories automatically, in addition to the different storytellers that we've come across, in which our project relies to a greater or lesser extent. Relying on story generation based on intelligent agents an planification, we've implemented our own storyteller. These agents use an external planner to know what to do whenever they want, which leads to interactions between them. We use this interactions between agents as the content of the stories generated, so that, in some way, these agents will make part of the work for us. Once we've collected the data produced by the agents, we'd have finished the "create" phase, and there will only be the "narrate" phase remaining. For this, by now, we've predefined text in between the code itself, although our initial idea was to use an external program, that had the log with the interactions occurred as an input. With all this, we've achieved to implement a basic storyteller, capable of generating not only different, but independent stories.

Keywords

Intelligent Agents
Computational Creativity
Storyteller
Planification
Simulation

Resumen

En este proyecto discutiremos cómo podemos generar creatividad computacional. Esto es una tarea compleja, ya que el propio concepto de lo que es la creatividad es ambiguo, puesto que en él intervienen cosas como el talento, que son de por sí difíciles de explicar. Siguiendo el dominio de nuestra aplicación, nos concentraremos en el ámbito de la narrativa y la creación de historias. Veremos cómo podemos generar historias automáticamente, además de los diferentes generadores con los que nos hemos encontrado, y que nos han servido como fuente de inspiración en mayor o menor medida. Basándonos en la generación de historias por simulación y planificación, hemos implementado un generador de historias basado en agentes inteligentes para conseguirlo. Estos agentes utilizarán un planificador externo para saber qué es lo que tienen que hacer en cada momento, lo que dará lugar a interacciones entre los mismos. Utilizaremos estas interacciones entre agentes como contenido de las historias a generar, de forma que en cierto modo, serán estos agentes los que nos hagan parte del trabajo. Recogiendo los datos producidos por estos agentes, habremos terminado la fase de "crear", y ya sólo nos quedará la fase de "narrar". Para ello, por el momento, tendremos textos predefinidos dentro del propio código, aunque la idea inicial era utilizar un programa externo que tomase como entrada un log con las interacciones producidas. Con todo esto, hemos logrado implementar un generador de historias básico, capaz de generar historias distintas e independientes unas de otras.

Palabras clave

Agentes inteligentes
Creatividad computacional
Generador de historias
Planificación
Simulación

CAPÍTULO 1: Introducción

1.1 Introduction

When trying to generate stories automatically, it's necessary to investigate about how real stories work. This takes us to the following question: What makes a story interesting?

A story is a highly complex intellectual product that exercises a wide range of the cognitive abilities of humans, involving as it usually does perceptions of time and space, attribution of knowledge to particular characters, identifying character goals, validating character plans to achieve the goals, attributing feelings to the characters, etc… [1]

For a story to be satisfying, there're lots of elements (characters, personality, knowledge, goals, emotions and feelings, dialogs…) that have to combine in hard to specify way, whose complexity will lead, in turn, to stories with different grades of complexity. In fact, if we stop to think about how a person in real life creates a story, we notice that there've been many attempts to identify what gives interest to a story, but there's no accord nowadays. This presents to us a serious problem to obtain creativity in the context of storytelling, in the sense that, as engineers, we can try to reproduce the patterns that take place when generating stories, but we can also use tools that allow us to have different points of view of the story that, otherwise, we won't have. For example, using processes that work in parallel, as it would help the characters be totally independent one from another to a large degree, being able to know what each of them is doing all the time in the story. However, when we talk about creating a story in real life, the creativity of the author tends to be one of the principal ingredients. The RAE defines creativity as the "ability to create", which in the case of artistic creations (like stories) implies that the creation is also "novel" in some way. This, by itself, constitutes a whole research branch in artificial intelligence (computational creativity), so that we're unable to reproduce it with total fidelity.

When doing the research work for this project, we realized that in a story, most of the times, the most important thing is not WHAT happened, but HOW happened. This represents a big challenge, as it's difficult to simulate things such as time (it's necessary, other way the stories will happen instantaneously) or conversations (they have to be fluid, spontaneous). In the same way, there are actions that have lack of interest by themselves, but may have in combination with others. Eating or sleeping are actions that may not appear in the final story, but they have to if, for example, the character meets his enemy while eating. Also, it can happen (and, in fact, it does) that there're stories that lack of interest, in which the characters barely interact between themselves, or the result of these interactions is irrelevant, but this is an inevitable part of the process.

Inside the process of storytelling, we can find very different approaches (see chapter 2). Among the most importants to take into account are: plannification based, in which there's a planner that will be responsible of, given the goals of the story, decide how it will develop, knowing the characteristics of the characters; based on rules, in which there are a set of probabilistic rules that will execute in terms of the state of the story an some probability; based on simulation, which concentrates in generation by the recreation of the facts that take place. In our case, we'll follow this last approach, as well as plannification.

1.2 Introducción

A la hora de tratar de generar historias automáticamente, es necesario investigar acerca de cómo funcionan las historias reales. Esto nos lleva a la siguiente cuestión: ¿Qué hace a una historia interesante?

Una historia es un producto intelectualmente muy complejo, que hace uso de un amplio abanico de habilidades cognitivas, como la percepción del espacio y el tiempo, la atribución de conocimiento a personajes concretos, identificación de objetivos de cada personaje, validar los planes que los personajes siguen para cumplir sus objetivos, atribuir sentimientos a los personajes, etc… [1].

Para que una historia sea satisfactoria, hay muchos elementos (personajes, personalidad, conocimiento, objetivos, emociones y sentimientos, diálogos…) que deben combinarse de una forma bastante difícil de especificar, cuya complejidad dará lugar, a su vez, a historias con diferentes grados de complejidad. De hecho, si nos paramos a pensar en cómo una persona en la vida real crea una historia, nos damos cuenta de que ha habido muchos intentos de identificar aquello que da interés a una historia, pero no hay ningún consenso actualmente. Esto nos presenta un serio problema para obtener creatividad en el contexto de la generación de historias, en el sentido de que, como ingenieros, podemos intentar reproducir los patrones que se producen a la hora de generar las historias, pero también usar herramientas que nos permitan tener distintos enfoques de la historia que de otra forma no lograríamos. Por ejemplo, usar procesos que trabajen en paralelo, ya que facilitaría en gran medida que los personajes fuesen totalmente independientes unos de otros, pudiendo saber que hace cada uno de ellos en todo momento de la historia. Sin embargo, cuando hablamos de crear una historia en la vida real, suele aparecer la creatividad del autor como parte principal del proceso. La RAE define la creatividad como la "capacidad de crear", lo que en el caso de las creaciones artísticas (como las historias), implica que lo creado sea "novedoso" en cierto sentido. Esto, por sí mismo, constituye una rama de investigación dentro de la inteligencia artificial (*computational creativity*), por lo que, actualmente, nos es imposible reproducirla con total fidelidad.

Haciendo el trabajo de investigación necesario para este proyecto, nos dimos cuenta de que en una historia la mayoría de las veces lo más importante no es el QUE sucedió, sino el CÓMO lo hizo. Esto representa un gran reto, ya que es complicado simular cosas como el tiempo (es necesario, ya que si no las cosas se producirían instantáneamente) o las conversaciones (deben ser fluidas, espontáneas). De igual forma, hay acciones que carecen de interés por sí mismas, pero que pueden obtenerlo en combinación con otras. Comer o dormir son acciones que pueden no aparecer en la historia final, pero deben hacerlo si, por ejemplo, el personaje en cuestión se encuentra con su enemigo mientras come. Además, puede ocurrir (y de hecho ocurre) que haya historias sin ningún interés, en las que los personajes apenas interactúen entre ellos, o que el resultado de esas interacciones no tenga ningún fondo, pero esto es una parte inevitable del proceso.

Dentro del proceso de generación de historias, nos encontramos con diferentes enfoques (ver capítulo 2). Entre ellos los más importantes a tener en cuenta son: basado en planificación, en el que se tiene un planificador que será el encargado de, dados los objetivos de la historia, decidir cómo se desarrollará, conociendo las características de los personajes; basado en reglas, en el que se tienen una serie de reglas probabilísticas que se ejecutarán en función del estado de la historia y de cierta probabilidad; basado en simulación, que se concentra en la generación mediante la recreación de los

hechos que se producen. En nuestro caso, seguiremos este último enfoque, además del de planificación.

1.3 Motivaciones

Nos decantamos por este trabajo debido a que nos interesa especialmente todo lo relacionado con la inteligencia artificial. Ayudó también el hecho de que durante el curso dedicado a tal tema, el apartado de agentes inteligentes fué muy corto y genérico, por lo que, al comienzo del trabajo, apenas teníamos un conocimiento básico de lo que eran los agentes. Sin embargo el tema en sí nos pareció muy interesante, por lo que decidimos que mediante este proyecto podríamos aprender más cosas sobre los agentes, y aprender a usarlos, no sólo en este proyecto, sino también en posibles aplicaciones futuras. Además nos motiva especialmente el hecho de que pueda ser un proyecto de largo recorrido, que pueda ser continuado por otros estudiantes a partir de nuestro trabajo.

1.4 Objetivos

Desarrollar una aplicación capaz de generar historias automáticamente. Para ello, utilizaremos agentes inteligentes (que harán las veces de los personajes), de forma que interactúen entre ellos. Observando las interacciones que se producen, y mediante un “traductor” externo que se nos facilitó, podremos generar dichas historias.

Estos agentes se guían por objetivos que pasaremos en un fichero de configuración, y usando un planificador obtendremos las acciones que deberán realizar. Dado que nuestro programa es a escala reducida, la mayoría de las veces sucederán las mismas cosas, pero si aumentamos las localizaciones disponibles, los personajes, los tipos de personajes, y realizamos los cambios pertinentes al dominio que pasamos al planificador, para que aborde dichos “aumentos”, podríamos observar cómo, aunque el fin fuese el mismo, la historia tendría variaciones.

Dado que el proyecto comenzará desde cero, nos marcamos como objetivo poder generar una pequeña historia, con unos pocos personajes, y en un escenario concreto. Dejamos para futuros trabajos la generación de historias más largas, el uso de personajes más complejos y un mayor número de ellos, etc...

A continuación, listaremos uno por uno los objetivos que nos propusimos al comienzo de este proyecto. Nuestros objetivos fueron:

- Diseñar un generador de historias que sea capaz de generar varias historias diferentes.
- Hacer el generador de historias configurable.
- Hacer un generador capaz de cambiar el contexto de la historia mediante un fichero sencillo de configuración.
- Hacer que los personajes utilicen la planificación, o replanificación en caso necesario.

CAPÍTULO 2: Trabajo relacionado

2.1 Agentes software

El mundo del software es uno de los que más riqueza y diversidad poseen. Existen millones de programas disponibles, ofreciendo una amplia variedad de información y servicios, todo ello dentro de un amplio dominio. Aunque la mayoría de estos programas dotan a sus usuarios de capacidades significativas cuando se usan por sí solos, está creciendo la demanda de programas que puedan "relacionarse", es decir, cambiar información y servicios con otros programas para, así, solucionar problemas que de otra manera no podrían ser solucionados [2].

De hecho, muchos de los sistemas actuales son concurrentes de facto, y se espera que interactúen con componentes y se aprovechen de servicios encontrados dinámicamente en la red. Además, los sistemas se están convirtiendo en "entidades 24 horas", que no se pueden parar, restaurar y mantener de la manera tradicional.

Parte de lo que hace esta capacidad de "relación" difícil de implementar es la heterogeneidad en el código. Diferentes personas escriben programas, en diferentes momentos, idiomas e interfaces. Todo esto se ve empeorado por el dinamismo existente en el ámbito del software. Los programas son reescritos continuamente, se añaden nuevos programas, etc…

Llamamos agente a cualquier cosa capaz de percibir su medio ambiente y actuar sobre él. Un agente humano tiene ojos, oídos y otros órganos sensoriales además de manos, piernas, boca y otras partes del cuerpo para actuar. Un agente robot recibe pulsaciones del teclado, archivos de información y paquetes vía red a modo de entradas sensoriales y actúa sobre el medio con mensajes en el monitor, escribiendo ficheros y enviando paquetes por la red [3].

Un agente inteligente es un sistema que está situado en un cierto entorno y que tiene capacidad de actuar autónomamente de forma flexible en ese entorno para satisfacer sus objetivos de diseño. Podemos definir esta flexibilidad como un equilibrio entre el comportamiento reactivo (en respuesta al estado del entorno) y proactivo (en respuesta al objetivo). Además estos agentes deben ser capaces de cooperar con otros agentes, de forma que puedan satisfacer sus propios objetivos (negociación y cooperación con agentes que persiguen distintos objetivos). Estos agentes pueden ser tan simples como subrutinas o ser de gran complejidad. Normalmente son entidades más complejas con algún tipo de control persistente (como por ejemplo, diferentes threads con un mismo espacio de direcciones, distintos procesos en una misma máquina, o distintos procesos en distintas máquinas).

Los agentes software surgen dentro del campo de la Inteligencia Artificial y, a partir de los trabajos desarrollados en el área de la inteligencia artificial distribuida, surge el concepto de sistemas multiagente. Posteriormente este concepto se extiende al resto de la ingeniería del software, planteándose actualmente como un nuevo paradigma de programación (programación orientada a agentes) altamente prometedor

Cada agente se sitúa dentro de un contexto, siendo capaz de "saber" lo que ocurre en él, actuando en consecuencia. Se rigen según sus propios objetivos, o bien por los proporcionados por otros

agentes. Estas acciones producen cambios en el contexto, y por tanto, en el comportamiento del resto de los agentes contenidos en dicho contexto. Finalmente, cada uno de ellos actúa por un periodo de tiempo determinado. Un agente software, una vez iniciado, se ejecuta hasta que él mismo decide acabar. Por supuesto, una persona podría desconectarlo manualmente, aunque en algunos casos ni siquiera ésto es posible (agentes móviles de internet).

Las características de estos nuevos sistemas "multiagente" encajan bastante bien con las dificultades que están surgiendo en la programación actual. La autonomía de dichos agentes refleja la naturaleza descentralizada de los sistemas distribuidos actuales, por lo que podrían considerarse una extensión de la modularidad y la encapsulación para sistemas de diferentes propietarios. Además, la flexibilidad con la que operan los agentes se adapta a situaciones dinámicas e impredecibles, aquellas en las que operarán dichos agentes. Finalmente, la naturaleza dinámica de las interacciones multiagente es apropiada para sistemas en los que los elementos que los constituyen (y sus interacciones) estén en constante evolución.
Algunas de las propiedades que los agentes pueden poseer, con diferentes combinaciones, son las que mostraremos a continuación [4]:

Autónomo - es capaz de actuar sin intervención externa directa. Posee cierto grado de control sobre su estado interno y sus acciones, basado en sus propias experiencias.

Interactivo - se comunica con el entorno y con otros agentes.

Adaptativo - capaz de responder a otros agentes y/o a su entorno en cierta medida. Formas más avanzadas de adaptación permiten al agente modificar su comportamiento en base a su experiencia.

Sociable - interacción que se caracteriza por la amabilidad o relaciones sociales amables, es decir, cuando el agente es afable, sociable, y amable.

Móvil - capaz de transportarse de un "medio ambiente" a otro.

Apoderado- puede actuar en nombre de algo o alguien, es decir, en interés de, como representante de, o en beneficio de otra entidad.

Proactivo - orientado por sus objetivos.

Inteligente - su estado interno está formado por conocimiento (por ejemplo, creencias, objetivos, planes, suposiciones) e interactúa con otros agentes usando un lenguaje simbólico.

Racional - capaz de escoger una acción basándose en objetivos y en el conocimiento que una acción determinada proporcionará.

Impredecible - capaz de actuar de formas que no son totalmente predecibles, incluso conociendo las condiciones iniciales. Es capaz de comportarse de forma no determinista.

Continuo en el tiempo - es un proceso en continua ejecución.

Coordinativos - capaz de realizar acciones en un entorno compartido con otros agetes. Normalmente, las actividades se coordinan mediante planes, flujos de trabajo, o algún otro mecanismo de manejo de procesos.

Cooperativos - capaces de coordinarse con otros agentes para cumplir con un objetivo común.

Competitivos - capaces de coordinarse con otros agentes excepto en el caso en el que el éxito de un agente implique el fracaso de otro.

Tosco - capaz de manejar errores y datos incompletos de forma robusta.

Estándares FIPA

FIPA es una organización de estándares del *IEEE Computer Society* que promueve la tecnología basada en agentes y la interoperabilidad de sus estándares con otras tecnologías. Según estos estándares, nos disponemos a mostrar la estructura que debe tener la plataforma [5] (ver figura 1):

- *Agent*: un proceso que implementa la funcionalidad autónoma y comunicativa de una aplicación. Los agentes se comunican usando el lenguaje de comunicación entre agentes (ACL, *Agent Communication Language*). Los agentes son los actores principales de una plataforma multiagente que combina uno o más servicios, como está publicado en la descripción del servicio.

- *Directory Facilitator* (DF): es un componente opcional de la plataforma, pero en caso de estar presente, debe estar implementado como un servicio. El DF proporciona un servicio de “páginas amarillas” para otros agentes. Los agentes registrarán sus servicios allí, o buscarán un servicio que necesiten de entre los registrados por el resto de agentes.

- *Agent Management System* (AMS): es un componente obligatorio en la plataforma. El AMS realiza la supervisión y control sobre el acceso y uso de la plataforma. Será único, y mantendrá un directorio con los AID (*Agent ID*) que contengan direcciones de transporte (entre otras cosas) para los agentes registrados en la plataforma.

- *Message Transport System* (MTS): es el método de comunicación estándar entre agentes en diferentes plataformas.

- *Agent Platform* (AP): proporciona la infraestructura física en la que desplegar los agentes. La plataforma está formada por todos los componentes anteriormente descritos.

  Es importante notar que el concepto de AP no implica que todos los agentes que residen en una AP deban residir en el mismo ordenador anfitrión.

- *Software*: describe todas las instrucciones no relacionadas con agentes que son accesibles por éstos. Los agentes pueden acceder al software para, por ejemplo, añadir nuevos servicios, adquirir nuevos protocolos de comunicación, adquirir nuevos protocolos/algoritmos

de seguridad, adquirir nuevos protocolos de negociación, acceder a herramientas que soporten migración, etc.

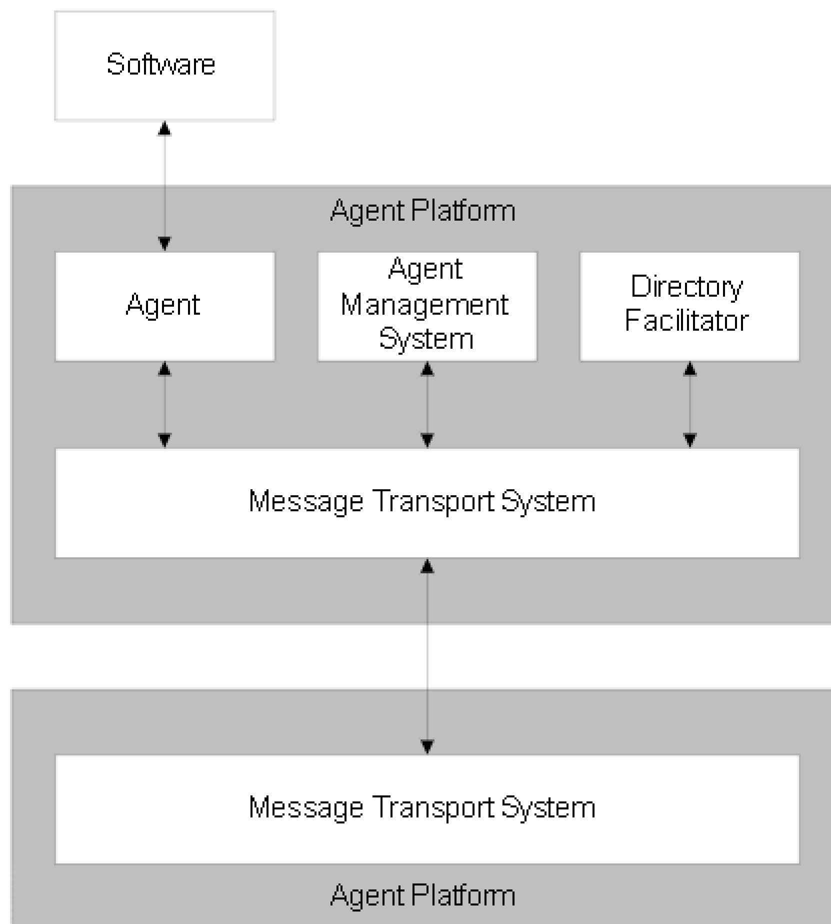

Figura 1: Estructura de una plataforma multiagente [5]

2.2 Planificación

Llamamos planificación al problema de seleccionar acciones orientadas por objetivos basándonos en una descripción de alto nivel del mundo. Podríamos escribir un libro completo con las diferentes variaciones de lo que ésto significa. Desde el punto de vista de la informática, la planificación no es más que un formalismo que describe sucintamente grandes sistemas de transiciones, similares a redes de autómatas o máquinas de Turing (ver figura 2). Trivialmente, este es un problema difícil (NP-completo). Lo que hace a la planificación especial es su proposito dentro de la inteligencia artificial.

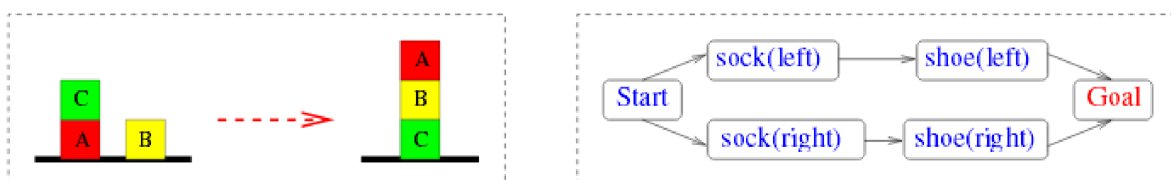

Figura 2: ejemplo de planificación [6]

Mientras que en sus comienzos, la idea de la planificación era la de conseguir la flexibilidad que un humano posee a la hora de resolver problemas, actualmente se ha adaptado una filosofía mucho más pragmática, de forma que será recomendable el uso de un planificador en alguno de los siguientes supuestos:

- **El problema a resolver cambia continuamente.** En este caso, implementar un algoritmo concreto de resolución sería muy costoso, ya que tendríamos que estar adaptándolo frecuentemente. Mediante la planificación, es suficiente con cambiar el modelo de planificación.

- **Es costoso implementar un algoritmo concreto**. A menos que el problema sea relativamente sencillo, hacer un algoritmo será costoso, mientras que escribir un modelo de planificación es, normalmente, una tarea mucho más sencilla.

El principal problema de la planificación es la dimensionalidad, es decir, el problema de la explosión de estados, aunque, gracias a los trabajos realizados durante la última década, ahora disponemos de nuevas técnicas capaces de lidiar con este problema. Gracias a eso, la planificación ha experimentado un gran avance de escalabilidad en ese periodo. Mientras que en el caso del problema SAT (*satisfiability,* problema en el que, dada una fórmula booleana, debemos encontrar si existe una cierta combinación de valores que la satisfaga) ha sido principalmente gracias a técnicas de aprendizaje, en el caso de la planificación ha sido gracias a la búsqueda heurística.

En computación, dos objetivos fundamentales son encontrar algoritmos que manejan buenos tiempos de ejecución a la vez que nos dan soluciones óptimas a los problemas. Una heurística es un algoritmo que abandona uno, o ambos objetivos. Así, se utiliza una versión simplificada del problema a resolver como heurística (por ejemplo, en el caso del problema del 8-*puzzle,* podemos tomar una versión en la que las figuras pueden moverse a cualquier posición adyacente) [6].

2.3 Sistemas

Durante la fase de investigación, pudimos ver distintos ejemplos de generadores de historias. Nos concentramos en aquellos que utilizaban agentes software para ello, pues coincide con el dominio de nuestra aplicación, además de aquellos que utilizan la planificación, en mayor o menor medida. Nuestro objetivo durante esta fase era el de hacernos una idea de cómo podíamos aproximarnos al problema. En el sentido de la generación basada en agentes inteligentes pudimos encontrar un modelo muy parecido a lo que pretendíamos conseguir. Sin embargo, no hallamos ninguno del tipo de planificación del que teníamos en mente usar.

Novel Writer

El primer generador de historias del que se tiene constancia es el Novel Writer (Sheldon Klein 1973). El Novel Writer crea historias de asesinatos en el contexto de una fiesta de fin de semana. Se apoya en el modelo de microsimulación donde el comportamiento de personajes completos y eventos son controlados por reglas probabilísticas que progresivamente cambian el estado del mundo simulado. El flujo narrativo surge de los cambios de estado del modelo del mundo.

Sin embargo, la secuencia de escenas esta muy cableada en el código, de forma que corresponda al desarrollo de una fiesta de fin de semana, dejando a la simulación sólo la interacción entre los personajes. Esta secuencia de escenas puede ser considerada como una instancia de una gramática primitiva. Se proporciona como entrada una descripción del mundo en el que la historia tiene lugar. La víctima y el asesino dependen de las características de los personajes, también proporcionadas como entrada (con un ingrediente aleatorio adicional). Los motivos surgen en función de los eventos ocurridos durante el transcurso de la historia. El conjunto de reglas es muy limitado y sólo permite la generación de un tipo de historia muy concreto.

El Novel Writer sugiere una forma de modelar la focalización a través del uso de universos semánticos privados. Esto sólo está mínimamente esbozado en términos de cómo los sistemas operativos pueden permitir al sistema mantener copias en disco de un universo privado, cargarlas en memoria, tratarlas como un universo total mientras residen en memoria, operar con ellas como se requiera y, finalmente, volver a llevarlas al disco. En ningún momento se menciona cómo el material de un universo privado puede interactuar con el material de otro [7].

A continuación, mostramos un ejemplo de historia generada por este sistema (éste es sólo un pequeño episodio de una historia de 2.100 palabras):

```
The day was Monday. The pleasant weather was
sunny. Lady Buxley was in the park. James ran into
Lady Buxley. James talked with Lady Buxley. Lady
Buxley flirted with James. James invited Lady Bux-
ley. James liked Lady Buxley. Lady Buxley liked
James. Lady Buxley was with James in a hotel. James
caressed Lady Buxley with passion. James was Lady
Buxley's lover. Marion following saw the affair. Mar-
ion saw the affair. Marion was jealous.
```

Talespin

Talespin (Meehan 1977) es un sistema que cuenta historias sobre las vidas de las criaturas de un bosque. Combina *forward chaining* (de eventos a sus consecuencias) con *backward chaining* (de salidas deseadas expresadas como objetivos, resultados de un evento previo, a eventos concretos que puedan conducir a esa salida). Los objetivos pueden ser descompuestos en sub-objetivos en la fase de *backward chaining*.

Así, Talespin introduce los objetivos como *triggers* para cada acción, es decir, tras ejecutar una acción se añade el siguiente objetivo, o una lista de estos. También introduce la posibilidad de tener más de un personaje “solucionador de problemas” (e introduce listas de objetivos diferentes para cada uno). Además, los personajes poseen relaciones complejas (familiaridad, confianza, afecto…). Estas relaciones actúan como precondiciones en algunas acciones y como consecuencias en otras, constituyendo un modelo simple de motivación de los personajes. La personalidad de los personajes se modela en términos de grado de amabilidad, arrogancia, honestidad e inteligencia.

Meehan debate sobre qué hace válida a una historia (existencia de un problema, grado de dificultad

de solucionar el problema y naturaleza o nivel del problema resuelto), pero parece ser un procedimiento de evaluación externo al programa [8].

A continuación, podemos ver un ejemplo de un extracto de una historia generada:

```
John Bear is somewhat hungry. John Bear wants to
get some berries. John Bear wants to get near the
blueberries. John Bear walks from a cave entrance
to the bush by going through a pass through a val-
ley through a meadow. John Bear takes the blueber-
ries. John Bear eats the blueberries. The blueberries
are gone. John Bear is not very hungry
```

Author

Author (Dehn 1981) es un programa que intenta simular el comportamiento de la mente del autor cuando está creando una historia. Dehn sostiene que los mundos de las historias son desarrollados a posteriori, como justificación de los eventos que el autor ha decidido que sean parte de la historia. Un autor puede tener unos objetivos concretos en mente cuando se sienta a crear una historia, pero aunque no sea así, se acepta que hay algunos "meta objetivos" que limitan el proceso de creación de historias, como asegurarse de que la historia es consistente, plausible, que los personajes son creíbles, que mantiene la atención del lector en todo momento. A bajo nivel, esto puede traducirse como subobjetivos relacionados con situaciones, en las cuales el autor quiere que intervengan ciertos personajes, o el rol que ciertos personajes tienen en la historia.

Se entiende una historia como "la realización de una compleja red de objetivos del autor". Estos objetivos contribuyen a la estructura de la historia, guiando el proceso constructivo, pero no son visibles en la historia final.

Conforme a este modelo, gran parte del trabajo al inventar una historia recae en la continua reformulación de los objetivos del autor. Esto se engloba en el concepto de "reformulación conceptual": la idea inicial se reformula en el episodio principal (y eso a su vez en una sucesión de episodios), una caracterización se reformula como un episodio que la ilustra, un cambio en la relación entre dos personajes se reformula como un diálogo que la provoca. Algunos ejemplos de objetivos de autor a alto nivel serían hacer la historia plausible, dramática, ilustrar los hechos principales…

Author intenta modelar la mente del autor en el ámbito de la creación de historias (los sucesos sobre el mundo de la historia ya están estructurados, pero habría que añadir episodios memorables, personajes y demás). También intenta modelar la mente humana en el sentido de cómo se organiza el conocimiento y cómo se accede a él, siguiendo las teorías de cómo funciona la memoria (Kolodner 1980, Schank 1982).

Dehn considera la generación de historias como un proceso de razonamiento creativo, y como tal, debería contemplar dos características generales: el grado en el que dicho proceso es deliberado, y el grado en el que se produce por casualidad. Para modelar esto, Dehn postula dos metaobjetivos: cumplir el objetivo narrativo actual y encontrar mejores objetivos narrativos. Es este segundo

metaobjetivo el que garantiza la dualidad deliberado-azar, permitiendo cambios de dirección cuando aparecen oportunidades no contempladas en un principio [9].

Universe

Universe (Lebowitz 1983) modela la generación de *scripts* para una sucesión de capítulos de una telenovela (gran cantidad de personajes interpretando historias múltiples, simultáneas y superpuestas que no tienen final). Es el primer sistema, de generación de historias, en dedicar especial atención a la creación de los personajes. Presenta estructuras de datos complejas para representarlos, y propone un simple algoritmo para rellenarlas, en parte, de forma automática. Pero el grueso de la caracterización debe ser realizada manualmente por el usuario.

Universe está dirigido a explorar la generación de historias muy extensas, una serie continua más que una historia con principio y final. En primera instancia, está destinado a ser una ayuda para los escritores, con posibilidades de desarrollarlo en un generador de historias autónomo más adelante. Aborda una cuestión sobre el procedimiento de creación de una historia sobre un mundo ficticio: si el mundo debe construirse primero, y luego crear un argumento sobre éste, o si el argumento debe ser el que dirija la construcción del mundo, con personajes, localizaciones y objetos creándose según la necesidad. Lebowitz se declara en favor de la primera opción, razón por la cuál Universe incluye facilidades para crear personajes independientemente del argumento, al contrario que Dehn, que se decanta más por la segunda opción.

El proceso de generación de historias de Universe usa unidades de planificación para generar esbozos del argumento. El tratamiento del diálogo y la generación de texto a bajo nivel son pospuestas explícitamente para una etapa más tardía. Los fragmentos de argumento proporcionan métodos narrativos que cumplen objetivos, pero los objetivos considerados aquí no son de los personajes, sino del autor. Esto permite a los personajes emprender acciones que no han elegido hacer como agentes independientes (dando lugar a conflictos melodramáticos). La manera de proceder del sistema es parecida a la planificación descomposicional. El sistema posee un grafo de precedencia que posee los objetivos del autor pendientes y los fragmentos del argumento, y mantiene la relación que poseen entre sí y con eventos que ya hayan sido contados. Para planificar la siguiente fase del argumento, se elige un objetivo que cumpla las precondiciones y se expande. La búsqueda no es primero en profundidad, lo que hace que el sistema pase de expandir objetivos relacionados con una rama de la historia, a los de otra totalmente diferente. Al seleccionar fragmentos del argumento o personajes cuyos objetivos expandir, se da prioridad a aquellos que cumplan objetivos extra de los disponibles.

Un punto interesante de Universe es que, aunque se genere una historia sin un final como tal, el sistema alterna entre planificar la continuación del argumento y contar el argumento generado hasta el momento, desde la última vez que se contó.

El sistema incluye un mecanismo para expandir automáticamente su librería de fragmentos de argumento, creando nuevos fragmentos. Esto se consigue generalizando fragmentos existentes e instanciando la estructura resultante para generar más. Para evitar la pérdida de información, el proceso debe ser guiado por el análisis causal del fragmento inicial de argumento, que debe ser respetado por las operaciones de abstracción e instanciación. Para asignar un determinado carácter

a los fragmentos, sólo se generalizan ciertas características de éstos. Este proceso conduce a la identificación de los objetivos fundamentales del autor, como “mantener la tensión romántica” y “mantener la historia en movimiento”. Esto conduciría la generación de la historia, pero la única justificación dada para estos objetivos es que parecen válidos según la experiencia con historias melodramáticas [10].

A continuación, ponemos a modo de ejemplo una historia generada por este sistema:

```
Liz was married to Tony. Neither loved the other,
and, indeed, Liz was in love with Neil. However,
unknown to either Tony or Neil, Stephano, Tony’s
father, who wanted Liz to produce a grandson
for him, threatened Liz that if she left Tony,
he would kill Neil.
Liz told Neil that she did not love him,
that she was still in love with Tony, and that he
should forget about her. Eventually, Neil was
convinced and he married Marie. Later, when Liz
was finally free from Tony (because Stephano had died),
Neil was not free to marry her and their trouble
went on.
```

Minstrel

Minstrel (Turner 1993) es un programa que genera historias sobre el rey Arturo y sus caballeros de la tabla redonda. Es el primer generador de historias en abarcar específicamente cuestiones de creatividad (está descrito explícitamente en la tesis doctoral de Turner como “un modelo computacional de creatividad y generación de historias”). El programa se inicia con una moraleja que se usa como semilla para construir la historia. Minstrel crea historias de entre media página y una página completa. Según su autor, Minstrel puede generar unas diez historias de esta longitud, y puede crear una serie de escenas de historias más cortas.

Minstrel usa unidades de construcción consistentes en objetivos y planes para satisfacerlas. Estas unidades operan a dos niveles diferentes: en términos de objetivos de autor y en términos de objetivos de los personajes. La construcción de historias en Minstrel funciona como un proceso de dos fases, englobando una fase de planificación y una de solución de problemas. La fase de planificación opera sobre una agenda a nivel de autor que guarda los objetivos del autor. El proceso consume los objetivos de esta agenda, bien dividiendolos en subobjetivos de autor (que a su vez son guardados de nuevo en la agenda), o bien pasándose a la fase de solución de problemas (que intenta solucionarlos añadiendo los ingredientes necesarios a la historia). Hay dos operaciones en las que merece la pena pararse. Una engloba solucionar los objetivos a nivel de autor, instanciando un conjunto de esquemas parcialmente completos de los personajes. Esto se realiza mediante la consulta de la memoria episódica con los esquemas parcialmente completos mencionados anteriormente, usando los resultados para crear instancias del objetivo a nivel de autor correspondiente. La otra es la forma en la que los objetivos oportunistas se disparan. Cada vez que se crea una escena, Minstrel la revisa para comprobar si proporciona una oportunidad de aplicar uno

de los objetivos a nivel de autor que aseguren consistencia, o introduce uno de los motivos literarios deseados. Esto encaja con el segundo metaobjetivo de Dehn de buscar nuevos objetivos de autor [11].

El proceso de consulta de la memoria episódica está gestionado por TRAMs (*Transform Recall Adapt Methods*). Los TRAMs más básicos simplemente pasan la consulta tal cual está en la memoria episódica, y devuelven cualquier esquema que encaje. No obstante, en caso de fallo, entran en juego TRAMs más complejos que aplican una modificación básica a la consulta, preguntando en la memoria episódica con la resultante, y devolviendo una adaptación de cualquier resultado obtenido, revirtiendo la modificación aplicada a la consulta original. Los TRAMs pueden enlazarse con una cadena (que lleva a una cadena de sucesivas operaciones de adaptación). Esto captura un concepto muy similar al proceso de generalización propuesto por Lebowitz, y es la base de la afirmación de creatividad de Turner en Minstrel. A continuación, ponemos a modo de ejemplo ilustrativo, una historia generada por el sistema:

```
The Vengeful Princess
Once upon a time there was a Lady of the Court
named Jennifer. Jennifer loved a knight named
Grunfeld. Grunfeld loved Jennifer.
Jennifer wanted revenge on a lady of the court
named Darlene because she had the berries which
she picked in the woods and Jennifer wanted to
have the berries. Jennifer wanted to scare Darlene.
Jennifer wanted a dragon to move towards Darlene
so that Darlene believed it would eat her. Jennifer
wanted to appear to be a dragon so that a dragon
would move towards Darlene. Jennifer drank a mag-
ic potion. Jennifer transformed into a dragon. A
dragon moved towards Darlene. A dragon was near
Darlene.
Grunfeld wanted to impress the king. Grunfled
wanted to move towards the woods so that he could
fight a dragon. Grunfeld moved towards the woods.
Grunfeld was near the woods. Grunfeld fought a
dragon. The dragon died. The dragon was Jennifer.
Jennifer wanted to live. Jennifer tried to drink a mag-
ic potion but failed. Grunfeld was filled with grief.
Jennifer was buried in the woods. Grunfeld became
a hermit.
MORAL: Deception is a weapon difficult to aim.
```

Mexica

Mexica (Pérez y Pérez 1999**)** es un modelo computacional diseñado para estudiar el proceso creativo de la escritura, en términos de compromiso (*engagement*) y pensamiento (*reflection*, Sharples 1999). Está diseñado para generar historias cortas sobre los primeros habitantes de Méjico.

Mexica se sostiene en ciertas estructuras para representar su conocimiento: un conjunto de acciones (definida en términos de precondiciones y postcondiciones), y un conjunto de historias previas (representadas como conjuntos de acciones). Destaca sobre otros sistemas en que realmente construye sus propios esquemas partiendo del conjunto de historias previas. Estos esquemas son representados mediante una estructura, llamada contexto histórico (*Story-World Context*, SWC). Las SWC representan instancias de contextos (descritos en términos de conexiones emocionales y tensiones entre los personajes existentes) en los que una acción ha aparecido en una de las historias previas, y actúan como reglas durante la fase de *engagement*: una determinada acción se añade al argumento si se puede encontrar un SWC que contenga dicha acción, y que además concuerde con el argumento. Es importante mencionar que las SWC (y no las definiciones de acción en términos de sus precondiciones) se usan para encontrar la siguiente acción para extender el argumento. La fase de *reflection* revisa el argumento hasta ese instante, comprobando sobre todo la coherencia, el interés y la innovación. Las comprobaciones de innovación e interés incluyen comparar el argumento en ese momento con el conjunto de historias previas. Si la historia se parece demasiado a alguna de las anteriores, o su medida de interés es baja en comparación con el de las historias previas, el sistema entra en acción, asignando una guía que será seguida durante la fase de *engagement*. Estas guías son un equivalente de bajo nivel a los objetivos del autor, dirigiendo qué tipo de acciones pueden ser escogidas de entre las candidatas. La comprobación de coherencia sólo se lleva a cabo sobre la versión final de la historia, y engloba insertar en el texto, acciones concretas que necesitan ser explícitas, o bien objetivos o tensiones entre personajes que sean necesarias para comprender la historia. A menos que sean añadidos durante la comprobación, los objetivos y las tensiones no se incluyen en el texto final [12].

A continuación, podemos ver un ejemplo de una historia que genera:

```
Jaguar knight was an inhabitant of the Great
Tenochtitlan. Princess was an inhabitant of the
Great Tenochtitlan. Jaguar knight was walking
when Ehecatl (god of the wind) blew and an old tree
collapsed injuring badly Jaguar knight. Princess
went in search of some medical plants and cured
Jaguar knight. As a result Jaguar knight was very
grateful to Princess. Jaguar knight rewarded Princess
with some cacauatl (cacao beans) and quetzalli
(quetzal) feathers
```

Virtual Storyteller

La línea de trabajo iniciada por Talespin, basada en el modelado del comportamiento de los personajes, ha dado lugar a toda una rama de generadores de historias. Los personajes están implementados mediante agentes autónomos inteligentes que pueden elegir sus propias acciones, informados por su estado interno (incluyendo objetivos y emociones) y por la percepción de su entorno. La narrativa emerge de la interacción de estos personajes unos con otros. Esto garantiza argumentos coherentes, pero, como Dehn puntualizó, la falta de objetivos de autor hace que no sean interesantes necesariamente. No obstante, ha resultado ser muy útil en el contexto de entornos virtuales, donde la introducción de dichos agentes añade narrativa en un entorno interactivo.

El Virtual Storyteller (Theune 2003) introduce una aproximación multiagente a la creación de historias, donde se introduce un agente director que se encargue de cuidar el argumento. Cada agente posee su propia base de conocimiento (representando lo que sabe sobre el mundo) y reglas para regir su comportamiento. Concretamente, el agente director posee un conocimiento básico sobre cómo debe ser la estructura de un argumento (que posee un principio, un nudo, y un desenlace feliz) y ejecuta el control sobre las acciones de los personajes de tres formas: de entorno (introduce nuevos personajes y objetos), de motivación (asigna objetivos específicos a los personajes) y de prescripción (desautorizando una acción a realizar por un personaje). Sin embargo, el director no puede forzar a los personajes a realizar acciones concretas. Theune informa de que se contemplan reglas no estructurales para medir cuestiones como la sorpresa o la "impresionabilidad" [13].

El Virtual Storyteller incluye un agente narrador, encargado de traducir la representación de estados y eventos a frases en lenguaje natural. El esfuerzo se ha concentrado sobre todo en la correcta generación de pronombres para hacer que el texto resultante parezca más natural.

Utiliza, junto a la planificación convencional, la técnica del *late commitment* [14]. Consiste en que los propios agentes actores pueden cambiar el entorno de la historia "improvisando", de modo que les facilite cumplir sus objetivos. Si por ejemplo un personaje de la historia tiene que ir de A a B, puede "inventarse" que posee un coche para ir más rápido, cambiando así el entorno. Con esto, se consigue que las historias generadas dependan en menor medida del estado inicial del entorno, creando muchas más variantes.

El sistema se organiza en 3 capas:

1. La capa de *Fabula*, una red causal de todos los eventos que han ocurrido en el entorno de la historia (lo que ocurre en el escenario y por qué).

2. La capa *Plot* (trama, argumento) es una selección de eventos de la capa *Fabula*, que forman un todo coherente y consistente (hay varias tramas dentro de una misma fábula).

3. La capa *Presentation* representa la información necesaria para la presentación de la trama en un medio concreto. En el caso del Virtual Storyteller, ese medio será el lenguaje natural.

A continuación, podemos ver un pequeño fragmento de una historia generada por este sistema:

```
Because Princess Lovely heard that the lamb had
fled, she was sad.
She wanted the lamb back, so she wanted to find
it and ran out of the castle. Meanwhile,
the hungry children slaughtered the lamb. Because
Lovely saw the dead lamb, she was sad and hurt.
```

Fabulist

Fabulist (Riedl 2004) es una arquitectura para la generación y presentación automática de historias. Parte el proceso de generación de narrativa en tres niveles: generación de fábula, generación de los

hechos y generación de los medios.

La generación de la fábula usa un enfoque de planificación para la generación de la narrativa. El algoritmo de planificación IPOCL (*intent-driven partial order causal link*, algoritmo de orden parcial de enlaces causales, dirigido por intenciones) razona sobre la causalidad, y la intencionalidad y motivación de un personaje simultáneamente, a fin de producir secuencias narrativas causalmente coherentes (en el sentido de que avanzan hacia una conclusión), y posee elementos sobre la credibilidad de los personajes. Fabulist primero genera un plan narrativo que concuerda con el objetivo de salida, asegurando que todas las acciones y objetivos de los personajes son justificados con eventos dentro de la propia narrativa [15].

A continuación proporcionamos un ejemplo de historia generada por Fabulist. Los archivos de entrada incluyen: un modelo del dominio que describe el estado inicial del mundo de la historia y las posibles operaciones que los personajes pueden realizar, y un estado final. El plan para la siguiente historia se muestra en la figura 3:

```
There is a woman named Jasmine. There is a king
named Jafar. This is a story about how King Jafar
becomes married to Jasmine. There is a magic genie.
This is also a story about how the genie dies.
There is a magic lamp. There is a dragon. The drag-
on has the magic lamp. The genie is confined with-
in the magic lamp.
King Jafar is not married. Jasmine is very beautiful.
King Jafar sees Jasmine and instantly falls in love
with her. King Jafar wants to marry Jasmine. There
is a brave knight named Aladdin. Aladdin is loyal to
the death to King Jafar. King Jafar orders Aladdin to
get the magic lamp for him. Aladdin wants King
Jafar to have the magic lamp. Aladdin travels from
the castle to the mountains. Aladdin slays the drag-
on. The dragon is dead. Aladdin takes the magic
lamp from the dead body of the dragon. Aladdin
travels from the mountains to the castle. Aladdin
hands the magic lamp to King Jafar. The genie is in
the magic lamp. King Jafar rubs the magic lamp and
summons the genie out of it. The genie is not con-
fined within the magic lamp. King Jafar controls the
genie with the magic lamp. King Jafar uses the mag-
ic lamp to command the genie to make Jasmine
love him. The genie wants Jasmine to be in love
with King Jafar. The genie casts a spell on Jasmine
making her fall in love with King Jafar. Jasmine is
madly in love with King Jafar. Jasmine wants to
marry King Jafar. The genie has a frightening
appearance. The genie appears threatening to
Aladdin. Aladdin wants the genie to die. Aladdin
```

slays the genie. King Jafar and Jasmine wed in an
extravagant ceremony.
The genie is dead. King Jafar and Jasmine are mar-
ried. The end.

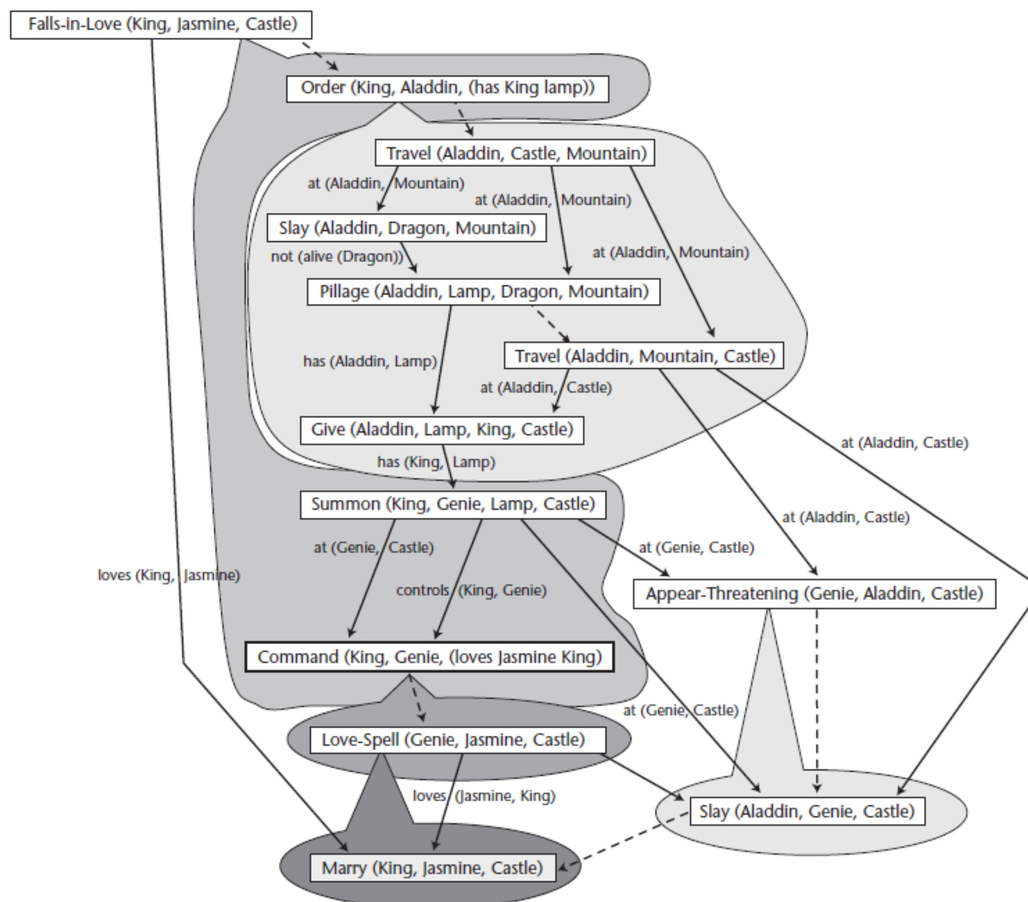

Figura 3: Plan para una historia de Fabulist [15]

Thespian

Thespian (Marsella, 2005) es un *framework* multi agente usado para la creación y simulación de historias interactivas. Está construido usando PsychSim, un sistema multiagente usado para la simulación social, especializado en la creación de personajes sociables para juegos. Está compuesto por agentes guiados por objetivos (proactivos) que hacen las veces de los personajes de la historia.

El sistema se organiza en dos capas: en la base tenemos el sistema multiagente, compuesto de agentes autónomos proactivos que hacen las veces de los personajes. Un aspecto clave de esta capa es la riqueza del diseño de estos agentes, proporcionándoles motivaciones, emociones, empatía y normas sociales. Los agentes de esta capa interactúan de manera autónoma con otros agentes y con el personaje controlado por el usuario, generando así la historia. Sobre esta capa se sitúa la del agente director, donde este redirige a los personajes de manera proactiva en el momento en el que prevé que sus comportamientos no seguirán las especificaciones sobre el argumento, que el autor proporciona como entrada. Esto se puede ver como un conjunto de objetivos para el sistema multiagente. Un aspecto clave de esta capa es que el agente director posee acceso a los modelos de

los agentes y del usuario. Utiliza estos modelos para evaluar si los objetivos del argumento se han cumplido, además de para redirigir a los personajes [16].

A continuación, ponemos un ejemplo de diálogo en una historia en la que los personajes sólo pueden intercambiarse información, aumentando su afinidad según las preguntas y respuestas:

```
1. Sergeant Smith to Kids: Hello!
2. Xaled to Sergeant Smith: Hello!
3. Hamed to Sergeant Smith: Hello!
4. Kamela to Sergeant Smith: Hello!
5. Sergeant Smith to Xaled: What is your name?
6. Xaled to Sergeant Smith: My name is Xaled.
7. Xaled to Sergeant Smith: What is your name?
8. Sergeant Smith to Xaled: My name is Mike.
9. Sergeant Smith to Xaled: How are y ou?
10. Xaled to Sergeant Smith: I am ne.
11. Xaled to Sergeant Smith: Are you an American?
12. Sergeant Smith to Xaled: Yes, I am an American.
13. Sergeant Smith to Xaled: I am learning Pashto.
```

Narrador en nn

En el sistema nn para ficción interactiva (Montfort 2007) el usuario controla al personaje principal de la historia introduciendo descripciones simples de lo que debe hacer, y el sistema responde con descripciones de los efectos de las acciones del usuario. Dentro de nn, el módulo narrador proporciona la funcionalidad propiamente dicha de contar la historia, de forma que el usuario puede pedir que se le cuente la historia hasta ese momento.

El módulo narrador de nn aborda cuestiones importantes en la generación de historias que no se han abarcado en sistemas previos: el orden de presentación de la narrativa y la focalización. En lugar de contar los eventos siempre en orden cronológico, el narrador permite varias posibilidades: *flashbacks*, *flashforwards*, intercalado de eventos de dos momentos diferentes, contar los eventos desde el final hasta el principio… También captura el correcto tratamiento de la tensión dependiendo del orden relativo del momento del discurso, del momento de referencia y del momento del evento. La focalización se maneja a través del uso de diferentes "mundos focalizadores" dentro del sistema. Aparte del propio mundo del sistema, nn mantiene mundos adicionales que representan las perspectivas y creencias individuales de cada personaje. Éstos pueden ser usados para el correcto tratamiento de la focalización (contar la historia desde el punto de vista de un personaje concreto) [17].

Party Quirks

Los actores de improvisación usan un juego para practicar. Consiste en que uno de ellos está en el escenario sin saber lo que va a pasar (el invitado, o *guest*). Posteriormente van entrando el resto de actores (que deben interpretar su papel improvisando). El objetivo del invitado es adivinar qué personaje están interpretando el resto de los actores, viendo como se comportan y cómo interactúan entre ellos.

Party Quirks (Brian Magerko 2010) es una versión simplificada de este juego para plataformas móviles. En él, el invitado es el jugador, y los demás actores son agentes *software* que interpretan un papel. Van saliendo primero uno, luego dos, etc… De forma que al principio se sabe pocas cosas del primero que sale. Sin embargo, a medida que van saliendo el resto de actores, se conoce más por cómo interactúan entre ellos, sus expresiones, etc…

Para hacerlo, se invitó a un grupo de actores de improvisación para que jugaran a este juego mientras se les grababa. Después, se les preguntó a cada uno de los actores que estaban pensando en cada momento del juego (con ayuda de la grabación), para poder aprender cómo expresaban sus pensamientos sin decirlos de manera literal. Una vez conseguido, se introdujo lo aprendido en forma de métodos en agentes software. Se escogieron un conjunto de dieciocho prototipos básicos de personajes (por ejemplo, pirata o *cowboy*) para hacer el juego asequible pero no trivial. Se entiende por prototipo unas construcciones idealizadas socialmente reconocibles que se asocian a un determinado tipo de personaje. Este enfoque es similar a la forma en que los humanos categorizan conceptos “confusos”. Se define cada prototipo como una colección de propiedades con un grado de pertenencia, en conjuntos que representan los atributos de cada personaje. Los atributos son adjetivos que definen al prototipo. Las acciones son actos físicos que normalmente transmiten atributos y son asociadas con al menos un par <atributo, grado de pertenencia>. Por ejemplo, el par <*usa_magia*, 0.8-1.0> implica una gran asociación con el uso de magia, que está conectado con la acción *controlWeather*. Cualquier personaje con *usa_magia* entre los mencionados valores puede ejecutar la acción *controlWeather* en el escenario.

El principal beneficio de usar pertenencia difusa es que captura la ambigüedad inherente a la interpretación. Realizar una acción con múltiples posibilidades puede llevar a otros actores a tener diferentes interpretaciones de la acción a las que se pretendía mostrar, lo que ocurre con frecuencia en las actuaciones. Los valores de la ambigüedad calculados también proporcionan los medios para determinar en qué medida las interacciones del huésped indican su convergencia con la "realidad" de la escena. En otras palabras, las acciones que el usuario ejecuta indican cómo de cerca están de adivinar lo que el “actor” interpreta [18].

Para el dominio de nuestra aplicación no nos es suficiente con hacer esto. Nuestro trabajo incluirá la presencia de diferentes localizaciones, de modo que no sólo es relevante cómo es la interacción en sí, sino también dónde se produce, cuándo, etc…

CAPÍTULO 3: Arquitectura

Cuando queremos contar una historia en la vida real, nos hacen falta ciertos ingredientes para que pueda considerarse como tal. Primero, y más importante, son los personajes. Las interacciones entre los mismos dan lugar a una serie de sucesos, que nos servirán de base para la historia a generar. También necesitamos un escenario en el que puedan moverse, lo que dará lugar, a su vez, a nuevas interacciones, esta vez con el entorno. Cuanto más complejo sea éste, mayor posibilidad de interacción con el mismo hay, pero menor posibilidad de interacción entre los personajes, por lo que debemos encontrar un equilibrio entre el tamaño del mapa y el número de personajes. Estos personajes deben conocer su entorno para poder satisfacer sus objetivos. Para ello, deben ser capaces de generar una estrategia, de forma que sepan cómo moverse por el mapa, y cómo interactuar con el resto de personajes y entorno. Una vez que tenemos esto, podremos comenzar a simular. En este punto, necesitamos una manera de recoger estos hechos, de forma que podamos manipularlos durante el proceso de narración. Para dicho proceso, debemos ser capaces de transformar una serie de hechos descritos en un formato concreto, a lenguaje natural reconocible. Además, para que los personajes puedan cumplir sus objetivos, deben ser capaces de generar una estrategia. Esta estrategia debe ser generada en el momento, para poder generar una alternativa en caso de que la que se generó en primera instancia no pueda llevarse a cabo. Por último, necesitamos un director, que les dé a los personajes los objetivos que deben solventar.
Este proyecto consta de cuatro partes principales, cada una con sus respectivos ficheros de configuración (ver figura 4):

- Sistema multi-agente: contiene todos los agentes necesarios para el funcionamiento del sistema.

- Mundo: contiene el mapa de la historia.

- Logger: responsable de recoger los eventos surgidos durante la ejecución.

- Planificador: responsable de generar los planes para los agentes. Éstos lo usarán para saber cómo deben cumplir sus objetivos, en términos de acciones a realizar.

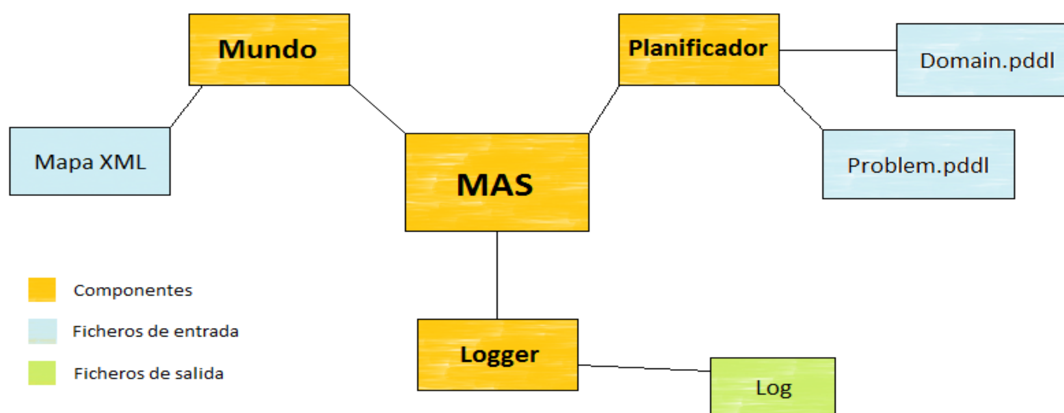

Figura 4: arquitectura del proyecto

A continuación, nos dispondremos a describir brevemente cada una de estas partes, de modo que se puedan tener referencias de la estructura de la aplicación para, más adelante, poder entrar en detalle con cada una.

3.1 Mundo

Hemos decidido que las historias que generaremos se llevarán a cabo en un mapa en el que tendremos varios escenarios, de forma que los personajes se muevan para poder cumplir sus objetivos. Esto añadirá riqueza a nuestras historias, ya que los posibles personajes podrían esconderse unos de otros o buscarse.

Este módulo poseerá como entrada un fichero de configuración en el que se especificará la estructura del mapa, de forma que podamos cambiarlo fácilmente en caso de querer generar historias diferentes. Se encargará de mantener y gestionar dicho mapa, atendiendo a las peticiones que le lleguen desde el sistema multiagente, como cambiar un personaje de localización. Además, deberá saber en qué localización se encuentran cada uno de los agentes para poder situarlos en el mapa, de forma que guarde el estado completo de la historia, lo cual es necesario para que los agentes puedan planificar correctamente.

3.2 Sistema multiagente

El módulo MAS (*Multi Agent System,* sistema multiagente) será el que contenga todos los agentes del sistema. Aquí se sitúan, además de los personajes, los agentes encargados de proporcionar objetivos y de gestionar el mundo. Por tanto, necesitaremos:

- Un agente por cada personaje de la historia, de modo que cada uno interprete un personaje. Estos agentes sólo serán capaces de interactuar con otros agentes del sistema multiagente y con el módulo de logger.

- Un agente encargado de comunicarse con el módulo mundo, de forma que sea el que atienda las peticiones de los agentes. Podrá comunicarse con el mundo y con el resto de agentes del MAS.

- Un agente encargado de crear al resto de los agentes al principio de la ejecución. Este agente no puede interactuar con ninguno de los componentes del sistema, salvo al principio de la ejecución.

3.3 Planificación

El módulo de planificación será el encargado de atender las peticiones de planes para los agentes del sistema multiagente. Decidimos que el planificador debería ser capaz de utilizar el lenguaje PDDL, de forma que en caso de necesitarlo, pudiéramos cambiar el planificador por otro sin necesidad de retocar demasiado el código.

PDDL (*Planning Domain Definition Language*) es un intento reciente de estandarizar un lenguaje de

descripción de dominios y problemas de planificación. Su desarrollo se produjo en gran medida para hacer posible la Competición Internacional de Planificación (*International Planning Competition*, IPC). Esta competición compara el rendimiento de distintos sistemas mediante el uso de benchmarks, por lo que era necesario un lenguaje común para especificarlos.

Debido a este requisito de usar PDDL, pudimos comprobar que eran necesarios dos ficheros para hacer funcionar cualquier planificador que tome como entrada problemas escritos en este lenguaje:

- Dominio: en este fichero se especifican las posibles acciones a realizar para resolver los problemas planteados. Este fichero será común para todos los agentes.

- Problema: en este fichero se especifican el estado inicial del problema y los objetivos a cumplirse. Incluirá una referencia al dominio del problema. Cada agente tendrá un fichero individual con el problema que desee planificar.

Además, otra de las ventajas de usar PDDL es que hace la aplicación mucho más configurable. Por ejemplo, simplemente añadiendo nuevas acciones al archivo PDDL correspondiente al dominio de la historia (es decir, dotando a los personajes de nuevas acciones), y creando sus clases Java correspondientes, las historias generadas variarán significativamente.

3.4 Logger

Como ya hemos mencionado anteriormente, en una historia los eventos en sí no son lo único importante. Por ello, necesitamos recoger no sólo lo que ocurre en la historia, sino también lo que se "dice" y en qué contexto. Por esta razón, necesitamos registrar todo lo que ocurre en la simulación, siendo esto las acciones que se están realizando y los mensajes intercambiados entre los agentes. Para ello, nos hemos decantado utilizar un logger, es decir, una herramienta que se encargue de recoger los eventos y mensajes intercambiados entre los agentes, para posteriormente escribirlos en un fichero.

Dicho archivo es lo que nos permite realmente "saber" qué es lo que está ocurriendo en la historia, siendo esto las acciones ejecutadas y los mensajes intercambiados.Sin embargo, debemos tener en cuenta que no todos los mensajes producidos durante la ejecución son relevantes para la historia, como por ejemplo, los mensajes de control que intercambian los agentes. Éstos mensajes no deberían aparecer en la historia final, ya que se usan sólo para mantener la coherencia interna.

Aunque actualmente nuestro programa escribe las acciones ocurridas por pantalla diréctamente desde el código, hemos adaptado todo para que se registre cada evento en el archivo de log del sistema, de forma que se pueda añadir un nuevo módulo de presentación que traduzca dichas acciones en frases en lenguaje natural en el futuro.

A continuación, pasamos a comentar más en detalle las partes descritas anteriormente, en lo que a implementación se refiere.

CAPÍTULO 4: Diseño e implementación

4.1 Sistema multiagente

4.1.1 Plataforma

Usaremos la plataforma Jade sobre Eclipse para la realización del proyecto. Aunque existen diferentes plataformas, hemos optado por esta ya que está ampliamente utilizada. Es una plataforma robusta y sólida, aunque la principal razón de su elección fué que sigue los estándares FIPA de manejo de agentes. Además, como ventaja adicional, utiliza el lenguaje Java, lo que nos permitió prescindir de aprender un nuevo lenguaje, ya que éste lo conocemos ampliamente de asignaturas pasadas.

Ahora, pasaremos a concretar algunos detalles de esta plataforma, que aunque básicamente siguen los estándares FIPA, son importantes para comprender la solución que hemos planteado y merecen la pena ser mencionados:

4.1.1.1 Comportamientos

Cada uno de los agentes del proyecto posee comportamientos, es decir, lo que realmente pueden hacer como agentes. Existen varios tipos en Jade, pero los que hemos usado por su funcionamiento, son los siguientes:

- OneShotBehaviour: comportamiento que sólo se realizará una vez.
- CyclicBehaviour: comportamiento que se ejecuta siempre (es decir, en cuanto termina, vuelve a empezar).
- FSMBehaviour: comportamiento que funciona como una máquina de estados, donde cada estado es a su vez otro comportamiento (estos pueden ser de tipos diferentes, incluso poder ser otra máquina de estados).
- Behaviour: clase abstracta. Se utiliza para, por ejemplo, implementar comportamientos que terminen en un momento concreto, o con un evento concreto.

4.1.1.2 Estados de los agentes

La plataforma Jade sigue las indicaciones del estándar FIPA en cuanto a los estados de los agentes. A continuación, nos disponemos a describirlos más en detalle (ver figura 5):

- *Initiated*. Se ha creado el objeto agente, pero aún no se ha registrado en el AMS (agent management system, el agente que supervisa el acceso y uso de la plataforma multiagente), no tiene nombre ni puede comunicarse con otros agentes.
- *Active*. El agente se ha registrado en el AMS, tiene nombre y puede acceder a las funcionalidades que Jade ofrece.

- *Suspended*. El agente está parado. Su thread interno está suspendido y no se está ejecutando ningún comportamiento.

- *Waiting*. El agente está bloqueado, en espera de algo. Su thread interno está en estado sleep, y se despertará cuando se cumpla alguna condición (normalmente, que llegue un mensaje concreto).

- *Deleted*. El agente ha terminado. Su thread interno ha terminado su ejecución, y el agente ya no está registrado en el AMS.

- *Transit*. Un agente móvil entra en éste estado cuando está migrando hacia una nueva localización.

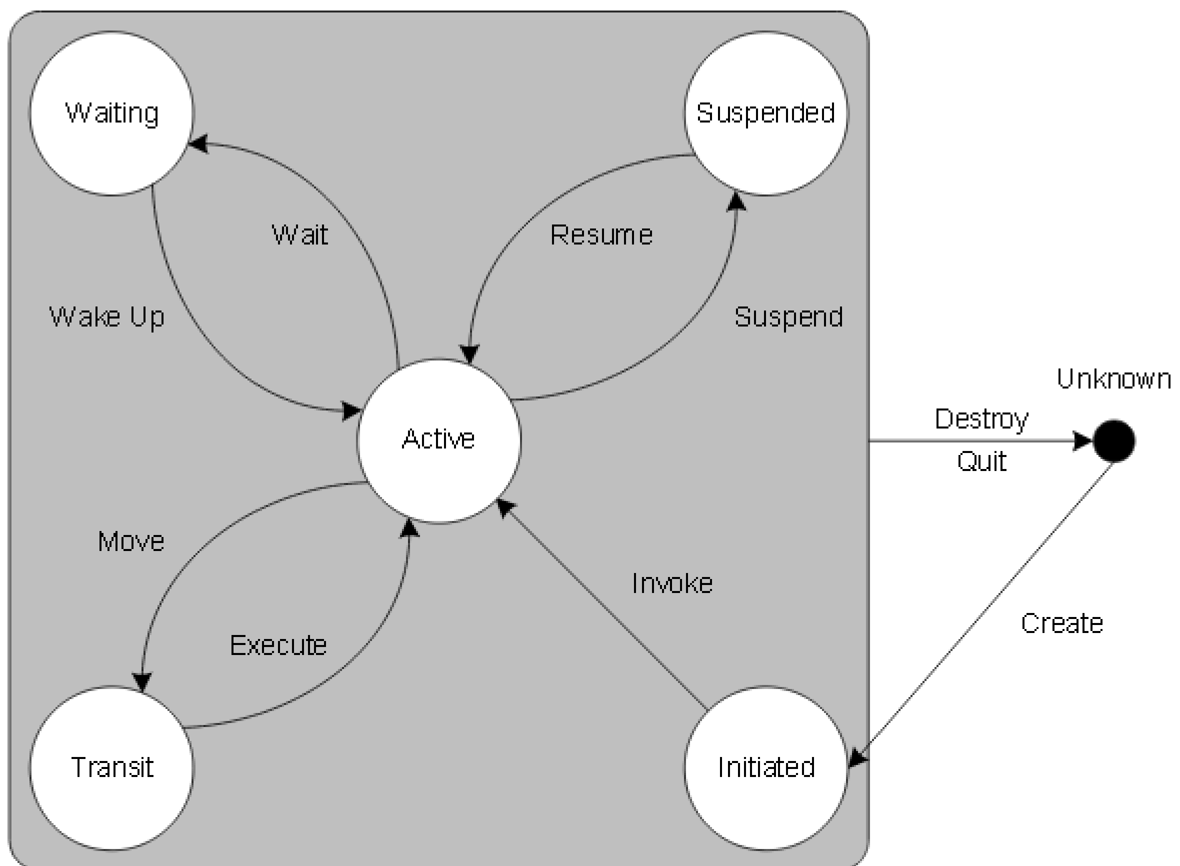

Figura 5: ciclo de vida de un agente según los estándares FIPA

4.1.1.3 Mensajes

A continuación, daremos una breve descripción de los tipos de mensajes que se usan en este proyecto, según los estándares FIPA:

- INFORM: Para decir algo a otro agente. El remitente debe creer en la corrección de lo que se envía.

- REQUEST: El remitente pide al receptor del mensaje que realice alguna acción. Es común que se pida al receptor que realice algún otro acto comunicativo.

- FAILURE: Sirve para decirle a un agente que no se ha podido llevar a cabo la acción pedida en el mensaje REQUEST anterior.

- CONFIRM: El remitente le confirma al receptor la corrección del contenido. El remitente cree inicialmente que el receptor no está seguro de este hecho.

- CFP: Agente emite una convocatoria de propuestas. Contiene las acciones a llevar a cabo y cualesquiera otros términos del acuerdo.

- PROPOSE: Se usa como respuesta a un mensaje CFP. Se propone un acuerdo.

- ACCEPT_PROPOSAL: Se usa para aceptar la proposición contenida en un mensaje de tipo PROPOSE anterior.

4.1.2 Agentes

Los agentes que hemos usado los hemos categorizado en dos tipos: aquellos que actúan como personajes, y los que usaremos como conductores de la historia.

4.1.2.1 Agentes como personajes

Tenemos una superclase Personaje (ver figura 6) que hereda de la clase de Jade Agent. Se añadió esta clase para añadir abstracción y legibilidad al código, así como para un futuro medio de poder hacer la aplicación configurable en el dominio. Esta última idea sería para tener personajes abstractos en lugar de personajes concretos, de modo que en lugar de “princesa” o “dragón”, tener clases como “héroe”, “villano”, “personaje secundario”... Así, después, podría indicarse en un fichero de configuración el nombre del personaje y el tipo de personaje que es (ver trabajo futuro).

A partir de aquí, cada uno de los personajes heredará de esta clase. Cada personaje tiene como atributos la cantidad de salud que tiene (entero), un String indicando la localización actual, una referencia al logger usado, el AID (identificador) del agente mundo y un ArrayList de pares objeto-cantidad. Aunque por el momento los personajes no son capaces de usar objetos, se añadió al comienzo del desarrollo del proyecto, y hemos decidido dejarlo para futuros trabajos.

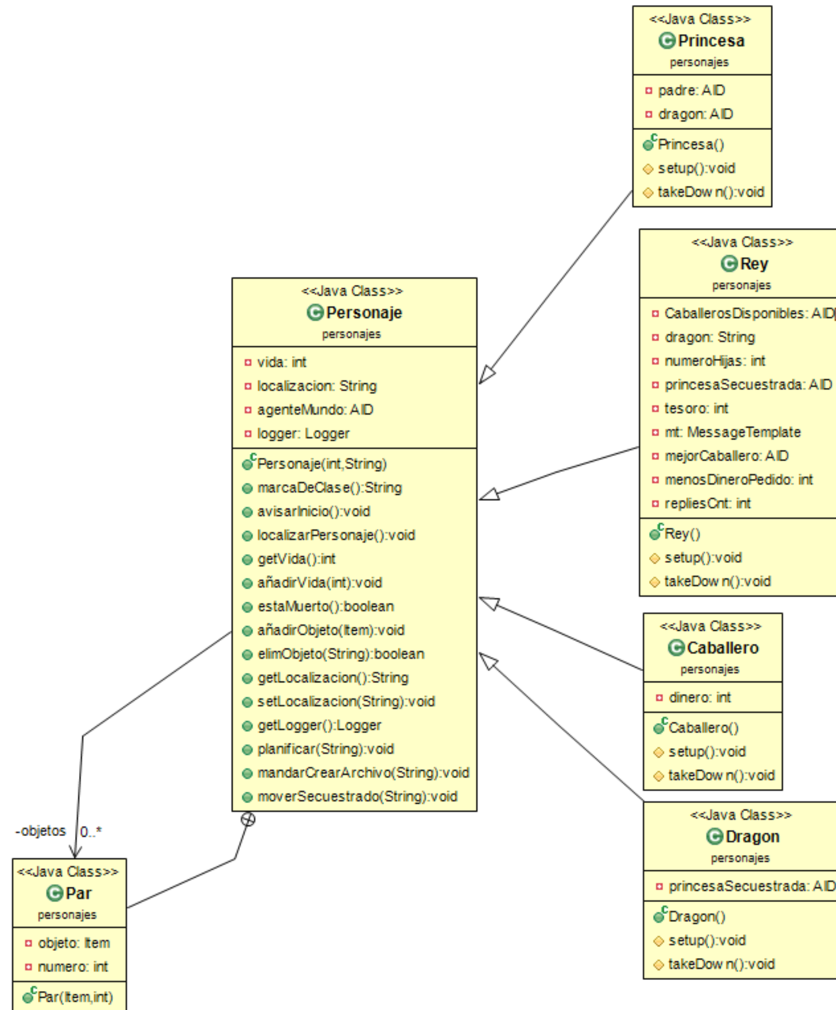

Figura 6: diagrama de clases de los personajes

Por otro lado, aunque en principio cada personaje poseía un archivo PDDL individual con sus objetivos, finalmente optamos por utilizar un archivo XML que describiera los objetivos de todos los personajes. A partir de este momento, cada vez que se quiera planificar, se enviará un mensaje al agente mundo pidiéndole que genere el archivo PDDL correspondiente al personaje en cuestión, que contendrá sus objetivos y el estado actual de la historia. No obstante, aún se conserva un archivo PDDL que contiene el dominio del problema (la historia).

Esta superclase contiene los métodos necesarios para que cada uno de los personajes que heredan de él pueda avisar al agente director de que se ha iniciado, pedir al agente mundo localizarse (mandándole un mensaje de tipo REQUEST con la localización inicial), y manipular sus atributos según sea necesario (añadir vida, objetos, saber si ha muerto, etc…). Sin embargo, hay dos funciones en las que merece la pena pararse:

- Función mandarCrearArchivo: esta función se encarga de enviar un mensaje de tipo REQUEST al agente mundo, pidiéndole que cree el archivo PDDL correspondiente al personaje en cuestión, incluyendo en él, el estado actual del mundo y sus objetivos (esto disparará el comportamiento ToPDDLfile del agente mundo que se describirá más adelante).

- Función planificar: esta función se encarga de parsear el plan devuelto por el planificador, y actuar en consecuencia. Tras llamar a la función mandarCrearArchivo, se creará un string que contenga el nombre del archivo PDDL correspondiente al dominio y el nombre del archivo PDDL correspondiente al personaje que desea planificar. Estos dos archivos se le pasarán como argumentos al planificador. Éste devolverá un string conteniendo las acciones a seguir. Una vez que las tenga, irá línea por línea creando la acción correspondiente a lo que el planificador dice, salvo en el caso en el que un personaje desee moverse junto con la princesa. En este caso, además de crear una acción de Mover para el personaje principal de la acción, le enviará a la princesa un mensaje de tipo REQUEST pidiendo que también se mueva. Por otro lado, en caso de que el personaje en cuestión muera, debido a la ejecución de alguna acción del plan, el parser dejará de ejecutar dicho plan. Por último, la función enviará un mensaje de tipo INFORM al personaje que pidió planificar, informando de que la planificación ha terminado.

Personajes de la historia

El sistema tiene cuatro personajes diferentes:

- Princesa: un personaje muy simple, pues no hace nada activamente. Sin embargo, las historias se generan alrededor de ella, ya que la historia comienza cuando el dragón la secuestra.

- Dragón: el “malo” de la historia. Su objetivo es secuestrar a la princesa y retenerla en su guarida indefinidamente.

- Rey: el padre de la princesa. Carece de objetivos hasta que la princesa es secuestrada. En este momento, su objetivo será rescatarla, para lo cual deberá contratar a un caballero.

- Caballero: el protagonista. Carece de objetivos hasta que el rey le contrata. Aquí, su objetivo pasa a ser rescatar a la princesa, para lo cual, deberá matar (o intentarlo más bien) al dragón.

Lógicamente de cada personaje se pueden crear el número que se desee, pero dado que el mapa es muy reducido, no tendría mucho sentido tener dos reyes. De igual forma, tener dos princesas carece de sentido, ya que cuando el dragón secuestra a una, se vuelve a su guarida a esperar a que vengan a salvarla.

Se pueden añadir varios dragones, y por tanto, si daría más sentido tener varias princesas, de tal modo que cada dragón vaya a por una distinta, o si tienen la intención de ir a por la misma, cuando llegue el segundo al castillo, verá que ya no está y tendrá que conformarse con otra. Igual pasa con los caballeros, con la diferencia de que el Rey designa a uno de ellos para que salve a su hija. Si este muere en el combate con el Dragón secuestrador, el Rey se encargará de elegir a uno de los caballeros restantes (añadiendo a los caballeros que hayan presentado su servicio después de la última elección).

Al principio, creamos a cada uno de estos agentes de forma que la historia estuviera muy cableada en el código. De ese modo, nada más inicializar a un dragón, este buscaba que princesas podía secuestrar. Seleccionaba a una de ellas de manera aleatoria, y la avisaba de que estaba

secuestrada, para que inmediatamente, esta princesa avisase a su padre, y el rey buscase caballeros, etc.

Una vez conseguimos que todo funcionase como queríamos, cambiamos los comportamientos de los agentes para que, en vez de funcionar de forma tan estricta, pudiera darse el caso de que algo fallase o saliera mal. El propio agente se daría cuenta de ello y volvería a un punto de la historia que permitiera volver a intentar la acción anterior o incluso cambiarla. Para tal fin, usamos el tipo de comportamiento de máquina de estados que proporciona Jade (FSMBehaviour), al que se pueden añadir estados que, funcionan como comportamientos individuales. La diferencia es, que al terminar cada comportamiento, devuelve un valor como salida, y al igual que ocurre con un autómata finito determinista, en función de esa salida, se pasará a un estado u otro, e incluso puede volver a repetirse el mismo.

Varios de los problemas a la hora de hacer este cambio, surgieron de que teníamos comportamientos cíclicos del tipo CyclicBehaviour, por lo que nunca terminaban y resultaba en historias inconsistentes. Para resolver el problema, optamos por cambiar estos comportamientos por otros que resultaban también cíclicos pero, que dadas unas condiciones, podíamos hacer que el agente el cual tenía ese comportamiento, lo eliminase.

Caballero

El caballero posee tres comportamientos diferentes (ver figura 7):

- OfrecerRescate. Comportamiento de tipo CyclicBehaviour que se inicia al crear al caballero. El comportamiento esperará a que el caballero reciba un mensaje del rey pidiendo ayuda. Una vez recibido, le responderá con un mensaje de tipo PROPOSE, con el dinero atributo como contenido.

- AceptarOfertaRescate. Espera un mensaje de tipo ACCEPT_PROPOSAL del rey, con los nombres del dragón y de la princesa como contenido, de forma que acepte las condiciones y le contrate. Una vez hecho esto, planificará el rescate de la princesa, y añadirá el comportamiento de FinPlanificación.

- FinPlanificación. Esperará un mensaje de tipo INFORM del planificador indicando que la planificación ha concluido. Una vez recibido, creará un mensaje para el rey en el que indicará si ha sido capaz de rescatarla o no (ha muerto en combate). En caso negativo (es decir, en caso de que el caballero haya muerto), responderá con un mensaje de tipo FAILURE.

Al iniciarse, el caballero se registra en las páginas amarillas, indicando que puede rescatar princesas. Después avisará al agente director de que se ha iniciado correctamente y se localizará en el mapa. Hecho esto, se inicializa el dinero que posee (como atributo), se informa por pantalla de que el caballero ha comenzado, se actualiza el log, y se añaden sus comportamientos OfrecerRescate y AceptarOfertaRescate. No ocurre nada especial cuando se crea.

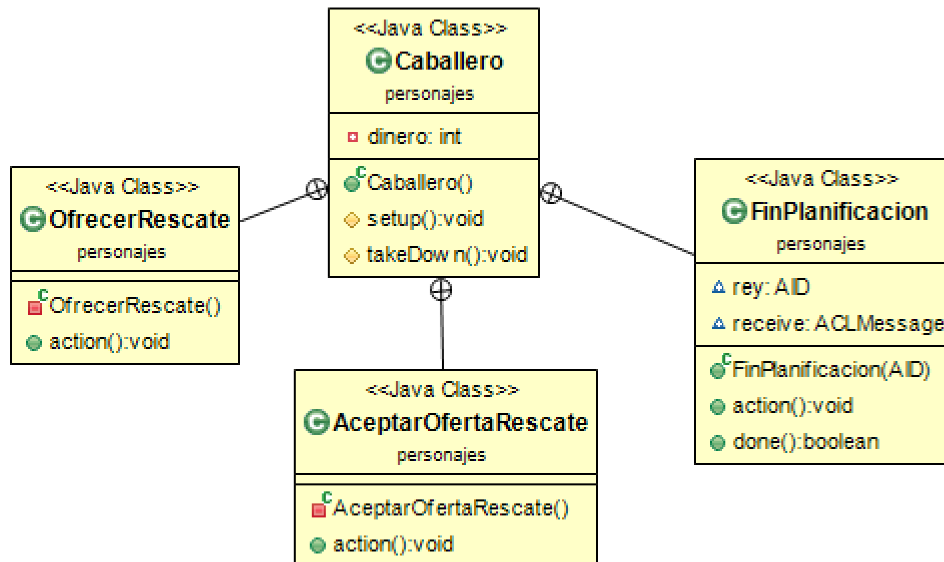

Figura 7: diagrama de clases del caballero

Dragón

El dragón posee cuatro comportamientos (ver figura 8):

- FinPlanificación. Al igual que el caballero, indica cuándo ha terminado la planificación. En ese momento, el dragón le enviará un mensaje informativo a la princesa de que ha sido secuestrada, y espera a que ésta le conteste. Una vez hecho esto, añadirá el comportamiento de Defender.

- Defender. Comportamiento que espera a que algún personaje le envíe un mensaje indicando que entra en batalla con él (a través de la acción Batalla). En este momento, comenzará una “batalla por turnos”, en la que los personajes intercambiarán los valores que restarán a su vida (en este caso, hemos decidido que dichos valores coincidan con sus valores actuales de salud, para simplificar). En caso de que el dragón muera, se actualizará convenientemente el estado, y se eliminará el agente del personaje.

- Secuestro. Este comportamiento se inicia al crear al personaje dragón. Buscará en las páginas amarillas a las princesas que son secuestrables. Si se han encontrado resultados, el dragón escogerá una de ellas aleatoriamente, y planificará el secuestro. Después añadirá los comportamientos de FalloSecuestro y de FinPlanificación para que estén operativos.

- FalloSecuestro. Este comportamiento se mantendrá a la espera de recibir un mensaje que indique el fallo en el secuestro (mensaje que generará la acción de Secuestrar en caso de que el agente mundo le responda con tal contenido, hecho que se produce cuando secuestrador y princesa no están en la misma localización). En tal caso, volverá a añadir el comportamiento de Secuestro, para que todo vuelva a comenzar.

Cuando el dragón se crea, además de avisar del inicio y localizarse, informará por pantalla (y por log del sistema) que se ha iniciado, y añadirá su comportamiento Secuestro.

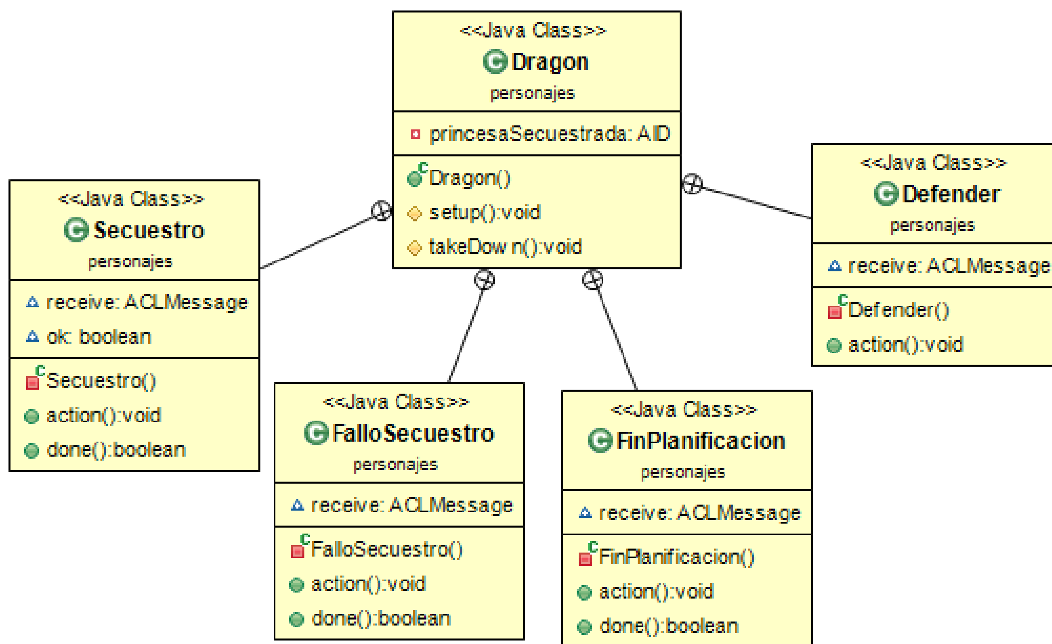

Figura 8: diagrama de clases del dragón

Princesa

La princesa tiene 3 comportamientos (ver figura 9). Todos ellos se añaden al momento de la creación de la princesa:

- MoverSecuestrada. Este comportamiento de tipo cíclico se pone en marcha cuando la princesa recibe un mensaje de tipo REQUEST diciendo que debe moverse (dicho mensaje será enviado por el método que parsea las acciones devueltas por el planificador, en caso de que otro personaje quiera moverse con ella). En ese caso, se moverá a la misma localización que su acompañante.

- AvisaAPadre. Este comportamiento espera el mensaje de tipo INFORM que el dragón le envía a la princesa para hacerle saber que ha sido secuestrada. En este momento, la princesa se borra de las páginas amarillas (ya que deja de ser secuestrable) y le envía al dragón su respuesta. Entonces, le envía al rey un mensaje de tipo REQUEST, con el nombre del dragón como contenido del mismo.

- Rescatada. El comportamiento está a la espera de que el rey le envíe a la princesa un mensaje tipo INFORM diciendo que ha sido rescatada. Una vez hecho esto, pone al agente en modo deleted.

La princesa posee como argumentos el AID (que es el identificador del agente) del padre y el del dragón que la secuestre. Como siempre, lo primero que hará al iniciarse será confirmar el inicio al agente director y localizarse. Luego creará el AID de su padre con el string que le pasa el agente director al crearla. Tras esto se informa por pantalla y por log de que se ha iniciado correctamente. Y

finalmente se registrará en las páginas amarillas indicando que es secuestrable, y añadirá todos sus comportamientos para que estén operativos.

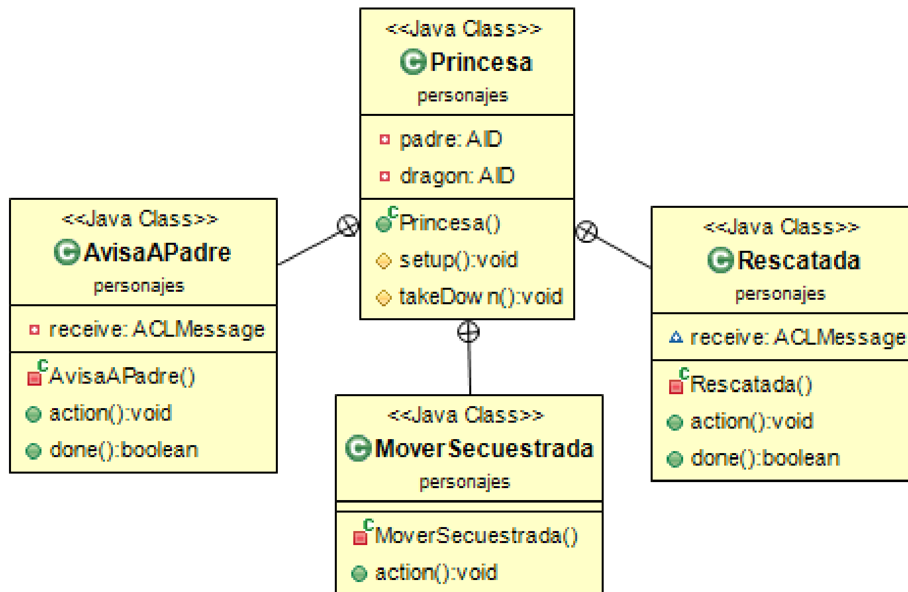

Figura 9: diagrama de clases de la princesa

Rey

El rey posee seis comportamientos. Hay que comentar que el rey funciona como una máquina de estados, que va ejecutando cada uno de los comportamientos. Por tanto, cada uno de estos comportamientos será ejecutado sólo una vez, pasando de unos a otros una vez finalizados:

- Atento. Espera a que la princesa le envíe un mensaje de tipo REQUEST pidiendo ayuda.
- Rescate. Buscará en las páginas amarillas todos los caballeros que ofrezcan un servicio de rescate. Terminará en el momento en que encuentre al menos uno.
- Ayuda. Aquí, se enviará un mensaje de tipo CFP a todos los caballeros encontrados en el anterior estado.
- RecibirOfertas. En este estado el rey se encargará de ver las respuestas de los caballeros. De entre ellas, se quedará con el caballero que pida menos dinero (pasado como contenido del mensaje). Para finalizar, si no se ha encontrado ningún caballero que cobre menos que el tesoro disponible, terminará con código buscará caballeros de nuevo.
- AceptarOferta. En este estado el rey enviará un mensaje tipo ACCEPT_PROPOSAL al caballero escogido en el estado anterior, indicando que acepta su oferta. Para finalizar, esperará el mensaje del caballero. Si el mensaje es de tipo INFORM, significará que el caballero seleccionado ha cumplido su misión. En caso contrario, querrá decir que el caballero ha muerto, y volverá a buscar caballeros para que completen la misión.

- Salvada. Aquí el rey se encargará de enviar un mensaje tipo INFORM a la princesa, diciéndole que ha sido rescatada por el caballero elegido anteriormente, y se restará al número de hijas del rey. En caso de que no le queden hijas por ser rescatadas, o de que su tesoro sea cero, el rey se pondrá en estado deleted. En otro caso, volverá al estado inicial.

Al iniciarse, además de localizarse y confirmar el inicio, mostrará por pantalla y log este hecho. Una vez hecho esto, inicializará su número de hijas con el número que el agente director le pasó a la hora de crearlo, e inicializará el valor de su tesoro a 100. Tras lo cual, creará la máquina de estados que representará su comportamiento. Será un objeto de tipo FSMBehaviour. Para hacerlo, primero registrara los estados y después las transiciones. Para registrar los estados, irá añadiendo el comportamiento correspondiente a dicho estado junto con un string que representa el nombre del estado. Y finalmente, se añadirán las transiciones, de forma que el autómata quede como en la figura 10. Después de haber especificado como será este comportamiento, sólo quedará añadirlo al agente (figura 11).

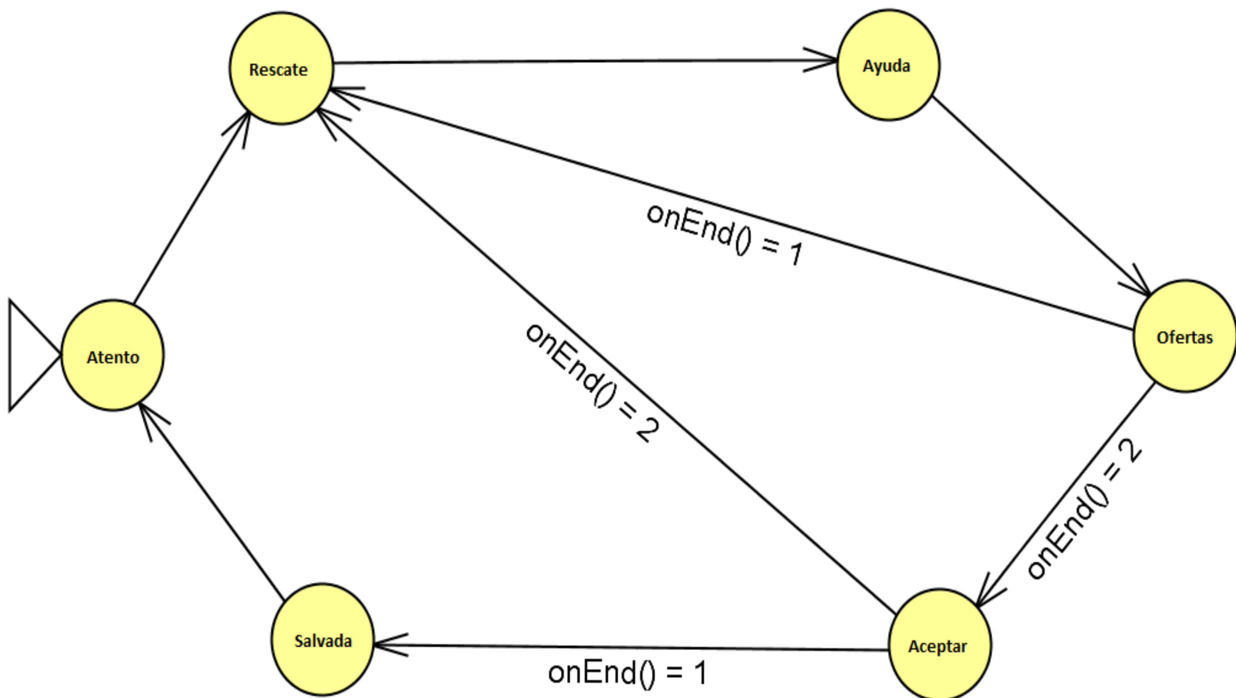

Figura 10: diagrama de transición del autómata del rey.

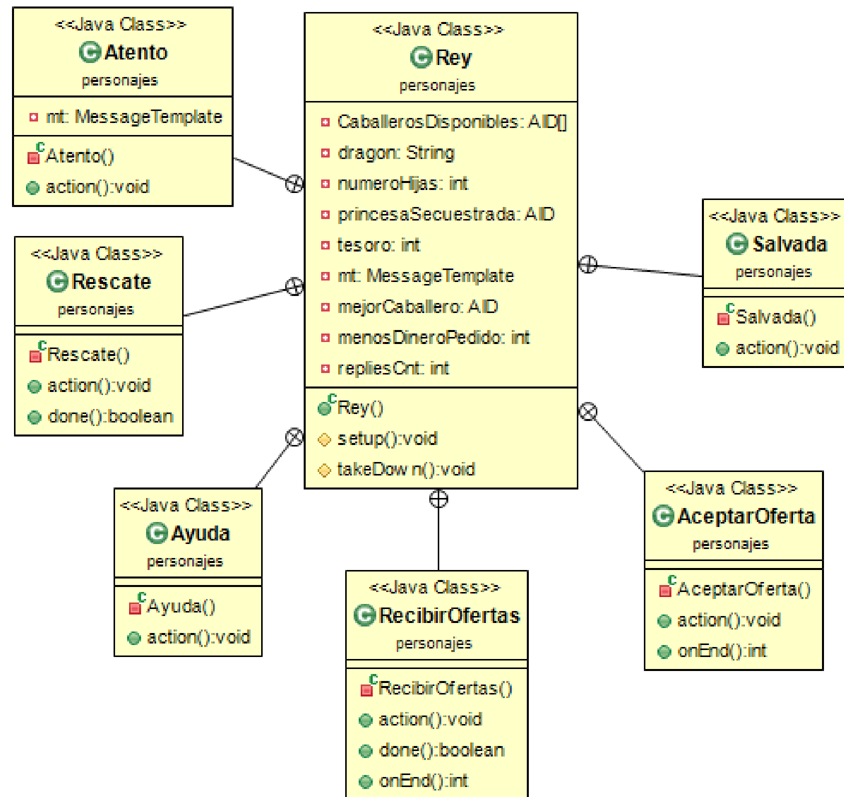

Figura 11: diagrama de clases del rey

4.1.2.2 Agentes conductores

Estos agentes son los encargados de “mantener” la historia. Serían el equivalente a los directores técnicos de una película. Su objetivo es crear y mantener el entorno de la historia, para que ésta pueda desarrollarse. Son:

- Agente mundo: encargado de actualizar en el mapa lo que va sucediendo. Cada vez que un personaje quiera, por ejemplo, cambiar de localización, deberá enviarle un mensaje para que se actualice su posición.

- Agente director: el primer agente que se crea. Es el encargado de crear al resto de los agentes . También se encargará de proporcionarles objetivos a lo largo de la historia, de forma que ésta pueda continuar. Actualmente, dichos objetivos son proporcionados en un fichero de configuración, aunque la intención para trabajo futuro sería que este agente fuera capaz de decidir qué objetivos dar en función de determinadas heurísticas, o bien preguntar al usuario, haciendo las historias mucho más interactivas.

Agente director

El agente director tendrá asociado un comportamiento de tipo OneShotBehaviour para cada agente (ver figura 12), para confirmar que ese agente se ha iniciado correctamente.

Cuando el agente director se inicia, va creando cada uno de los agentes necesarios para la historia.

Primero, crea el agente mundo, y después cada uno de los personajes, añadiendo para cada uno de ellos el comportamiento de confirmar inicio. También irá añadiendo a cada uno de los agentes a una lista (que posee como argumento), para saber cuáles son los agentes involucrados en la historia.

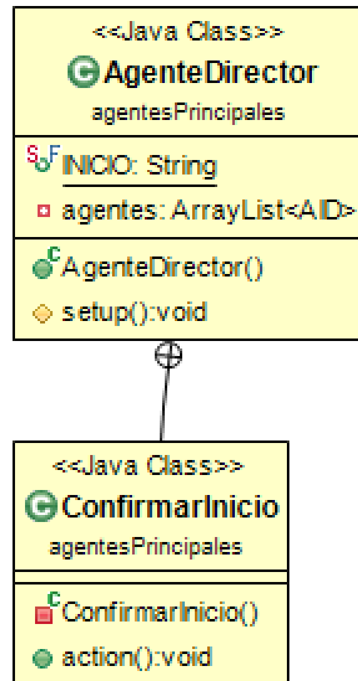

Figura 12: diagrama de clases del agente director

Agente mundo

El agente mundo posee siete comportamientos diferentes (ver figura 13), todos ellos de tipo CyclicBehaviour:

- ToPDDLfile. Este comportamiento se quedará bloqueado hasta que un personaje le envíe un mensaje de tipo REQUEST al agente mundo pidiéndole que cree (o modifique) el fichero correspondiente a dicho personaje. Este mensaje tendrá como contenido la clase del personaje que envía el mensaje, el nombre del personaje y el nombre de la princesa.

  Esta acción se realiza siempre antes de planificar, para que se cree (o se actualice) el fichero PDDL que contiene los objetivos del personaje (que se leerán del archivo XML que contiene los objetivos de todos los personajes) y el estado de la historia (que se leerá de la estructura de datos descrita más arriba), y así el personaje pueda planificar a partir del estado actual. El resto del archivo, es decir, el principio y el final, se escriben tal cual, ya que siempre son iguales.

- LocalizarPersonajes. Este comportamiento se lleva a cabo cuando un personaje envía un mensaje de tipo REQUEST al agente mundo pidiéndole que actualice su posición. Este mensaje contendrá: la clase del personaje, la localización destino, y dependiendo de la situación, la localización origen. Las situaciones que se pueden dar son las dos siguientes: o

bien que el personaje desee localizarse, como ocurre al principio de la ejecución de cada uno de los personajes, o bien que un personaje desee cambiar de posición. En ambos casos, el agente mundo se encargará, no sólo de colocar al personaje en la posición deseada (en caso de que sea posible), sino de actualizar el estado actual de la historia.

En caso de que el personaje esté cambiando de una localización a otra, primero deberemos eliminarle de la localización actual. Para ello, comprobaremos que la localización origen está conectada directamente con la destino, es decir, usando un único arco del grafo, y de ser así, eliminamos al personaje de la localización origen. Una vez hecho esto, pasaremos a incluirlo en la posición destino. Aquí añadimos al personaje a la localización nueva, actualizando a su vez el estado.

Si se ejecutó este comportamiento debido a que un personaje se había inicializado, y por tanto tendría que localizarse en alguna posición del mapa, además de incluirlo en la localización destino, habrá que añadir al personaje al estado. También añadiremos a este la casa del personaje, que será la localización destino que se pasó en el mensaje.

- Secuestro. Este comportamiento se produce al ejecutarse la acción de secuestro por parte del dragón. Dicha acción le manda un mensaje de tipo INFORM al agente mundo, para activar este comportamiento. Una vez recibido el mensaje, si el dragón y la princesa están en la misma localización, el agente mundo actualiza el estado, indicando que el dragón está con la princesa, que dicha princesa está secuestrada y que el personaje está lleno (le quita de la lista de personajes libres), y responde con un mensaje en el que indica el nombre de la princesa secuestrada. En caso contrario, el agente mundo devolverá un mensaje indicando el fallo.

- Liberar. Se ejecutará este comportamiento al realizar el caballero la acción de liberar princesa. La acción le mandará el mensaje al agente mundo para que actualice el estado, de forma que el dragón deje de tener a la princesa, y ésta pase a estar con el caballero.

- PersonajeEnCasa. Este comportamiento se produce al ejecutarse la acción de dejar en casa por parte del caballero. Dicha acción le mandará al agente mundo el mensaje correspondiente. Este actuará marcando a la princesa como liberada, añadiéndola a la lista de princesas salvadas e indicando que el caballero no está acompañado.

- ConvertirEnHeroe. El comportamiento comenzará cuando el caballero ejecute la acción de ser héroe, generando el mensaje correspondiente. El agente mundo se encargará de actualizar el estado añadiendo al caballero a la lista de héroes.

- MuertePersonaje. Se desencadena cuando la vida de un personaje llega a cero. En éste caso, el personaje manda un mensaje al agente mundo, y éste actualiza el estado eliminando al protagonista de la lista de personajes vivos.

El agente mundo poseerá como atributos el estado de la historia y el mapa. Al crearse, lo primero que hace el agente mundo es cargar el mapa. Para ello, leerá el archivo XML correspondiente. Primero creará un objeto localización y lo añadirá al mapa y al estado. Después irá añadiendo a la

localización sus adyacentes (también contenidos en el XML), y también los añadirá al estado, añadiendo también sus respectivos nombres.
Al iniciarse, lo primero que hará será confirmar el inicio al agente mundo, mediante un mensaje de tipo CONFIRM. Una vez hecho esto se registra en las páginas amarillas. Por último, añadirá todos sus comportamientos para que estén operativos.

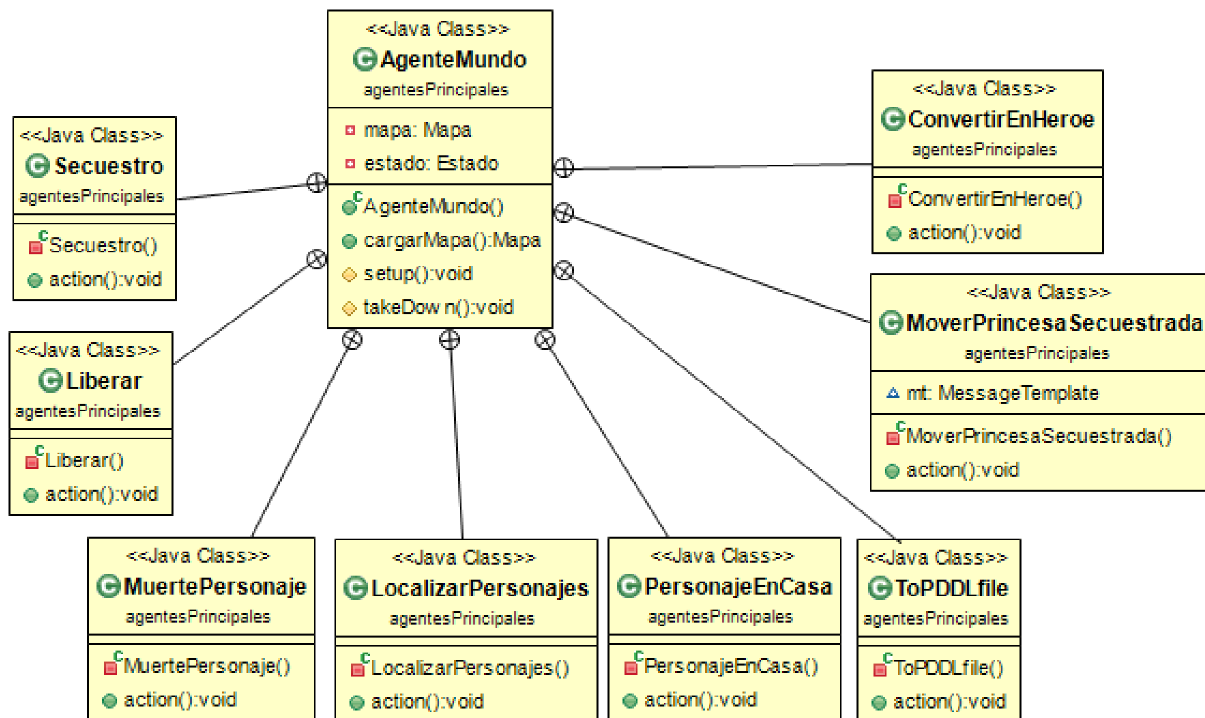

Figura 13: diagrama de clases del agente mundo

4.1.3 Ficheros de configuración

En esta sección nos disponemos a explicar cómo se organizan los diferentes ficheros de configuración presentes en el proyecto. Nos detendremos en aquellos ficheros que deben ser proporcionados explícitamente.

4.1.3.1 Objetivos de los personajes

Cada uno de los personajes posee unos objetivos que debe cumplir en la historia. Los distintos personajes realizarán sus acciones en función de los mismos, y las interacciones producidas por dichas acciones generarán la historia propiamente dicha.

Los objetivos de los personajes se proporcionarán como entrada para la aplicación. Estarán contenidos en un fichero XML como el que se muestra a continuación:

```
<Objetivos>

      <personaje tipo="Caballero" >
      <objetivo>(and (esHeroe Caballero) (salvada
Princesa))</objetivo>
      </personaje>

      <personaje tipo="Dragon" >
      <objetivo>(and (conPrinc Dragon Princesa) (enLoc Dragon
Montana))</objetivo>
      </personaje>

</Objetivos>
```

Sin embargo, para poder hacer uso del planificador que hemos escogido, el problema y el dominio deben pasarse como un archivo en PDDL. Así, podemos decir que cada personaje debe tener su propio fichero conteniendo sus objetivos y estado inicial en PDDL. Para lidiar con este problema, a la hora de hacer uso del planificador, generaremos el archivo PDDL correspondiente al personaje que se dispone a planificar a partir de este fichero de configuración. Por esta razón, y para simplificar el cómputo, los objetivos de cada personaje son incluidos en el archivo descrito anteriormente diréctamente en formato PDDL. De esta manera, el agente mundo se encargará de generar el archivo, con su cabecera (siempre igual), el estado inicial (que se generará a partir del estado actual de la historia, guardado por el agente mundo) y los objetivos (generados a partir del fichero anterior).

Para poder añadir nuevos tipos de personaje, lo primero será añadirlos al fichero de configuración anterior, describiendo sus objetivos. Posteriormente, deberemos añadir las acciones que pueden realizar al fichero PDDL del dominio, y por último añadiremos las clases Java correspondientes al personaje y a sus acciones.

4.1.3.2 Localización y Mapa

El objeto Localización representará un escenario de la historia. Contiene como atributos un string con su nombre, una lista de sus adyacentes, una lista de los nombres de los personajes que estén en dicha localización, y una lista de pares objeto-cantidad. Posee las funciones necesarias para poder manipular sus atributos adecuadamente.

El Mapa está compuesto por una lista de localizaciones. Posee funciones para poder añadir localizaciones y para que nos las devuelva.

Para poder crear el mapa, el agente mundo se encargará de leerlo desde un archivo XML. Dicho archivo contendrá las localizaciones con sus adyacentes, indicando si son seguras. El archivo de configuración del mapa tiene el siguiente formato:

```
<Mapa>

      <localizacion id="Castillo" >
      <esSegura> </esSegura>
      <conectadoCon>Montana Pueblo</conectadoCon>
      </localizacion>

      <localizacion id="Pueblo" >
      <conectadoCon>Montana Castillo</conectadoCon>
      </localizacion>

      <localizacion id="Montana" >
      <conectadoCon>Castillo Pueblo</conectadoCon>
      </localizacion>

</Mapa>
```

En este caso, mediante el uso de este archivo, se nos hace muy fácil añadir nuevas localizaciones al mapa, además de hacer sencilla la expansión de las mismas, ya que simplemente tendríamos que incluir más campos en la descripción, como por ejemplo, los objetos que estén presentes en cada una

4.2 Mundo

Nuestro mundo consiste en un mapa con tres localizaciones, de forma que los personajes pueden moverse entre ellas. En principio, el mapa estaba compuesto por posiciones, conectadas por caminos que podían estar abiertos o cerrados. Estos caminos también podían ser bidireccionales o no serlo, e incluso convertir un bidireccional en uno unidireccional y viceversa. De esta forma podíamos hacer que el camino se bloquease por algún evento o acción de uno de los personajes.

Las localizaciones que incluímos son:

- Castillo: donde empiezan la princesa y el rey.
- Montaña: donde comienza el dragón.
- Pueblo: donde está el caballero al inicio.

Como uno de nuestros objetivos principales es hacer el sistema fácil de configurar, utilizamos ficheros de texto para la configuración del mapa. En principio, creamos una estructura que debía seguir el archivo asociado al mapa, y creamos un cargador que lo reconociera y creara las distintas posiciones en el mapa junto con sus caminos. Sin embargo, nuestro tutor nos propuso utilizar archivos XML, de forma que la configuración fuese más legible y fácil de aprender y ampliar. Además, puesto que Java dispone de sus propios lectores XML, la creación del mapa se hizo infinitamente más sencilla y rápida. El archivo XML contiene las distintas localizaciones del mapa, y el nombre de las localizaciones a las que está conectada cada una.

Con el cambio en el formato de entrada del mapa, se nos ocurrió la idea de cambiar la estructura del

mapa. De tal forma que ahora simplemente tenemos localizaciones haciendo que el mapa funcione como un grafo. Esto hace que podamos tener localizaciones más complejas, teniendo otro mapa dentro de una localización. Por ejemplo, en la localización “Castillo” podemos hacer un mapa de este, teniendo las diferentes habitaciones y el cómo están conectadas entre sí, al igual que ocurre con el mapa externo.

Gracias al cambio en el mapa, se pueden utilizar los distintos algoritmos conocidos de cálculo de la ruta más corta, recubrimiento, etc… que, aunque ahora mismo no son necesarios dado el reducido tamaño, sí pueden ser útiles para futuras ampliaciones del proyecto. Se ha incluído en el código del proyecto la librería *JGraphT*, que es una librería de grafos gratuita de Java. Esta proporciona objetos y algoritmos presentes en la teoría de grafos. Actualmente, sólo la usa el planificador, pero podría utilizarse para crear un mapa con pesos entre las localizaciones. De esta forma, haríamos que los viajes entre las distintas posiciones del mapa, costase un cierto tiempo al agente.

4.2.1 Estado

El agente mundo guarda el estado de la historia en una estructura de datos que hemos llamado Estado. En él guardamos:

- Un HashMap<String, ArrayList<String>> para:
    - La localización y la lista de sus adyacentes.
    - La clase del personaje, y la lista de los nombres de los personajes de dicha clase.

- Un HashMap <String, String> para:
    - El nombre del personaje y su localización.
    - El nombre del personaje, y el nombre de la princesa con quien está.
    - La localización de las casas de los personajes.

- Un ArrayList<String> para:
    - Las localizaciones que son seguras.
    - Los personajes vivos.
    - Los personajes libres (sin acompañante).
    - Las princesas secuestradas.
    - Los nombres de todos los personajes y localizaciones.
    - Las princesas salvadas.
    - Los héroes.

Esta estructura de datos se irá actualizando en tiempo real, en función de los cambios que se produzcan en la historia.

A continuación, nos disponemos a describir los diferentes comportamientos de los que hacen uso los agentes de nuestro sistema.

4.2.2 Objetos

Todos los objetos parten de una clase abstracta principal llamada “Objeto”, que posee como atributos strings para su nombre y descripción. Incluye funciones para manipular dichos atributos y otras

(abstractas) para usarse, indicar si puede usarse e indicar si puede cogerse.

En el siguiente nivel, nos encontramos con que los objetos pueden ser items o decorado (extienden de Objeto), pudiéndose coger los primeros pero no los segundos. A su vez, los items pueden clasificarse como objetos con usos limitados (ExpirationItem) y objetos persistentes (PersistentItem, ambos extienden de Item). Los objetos de usos limitados poseen como atributo el número de veces que se pueden usar. Podrán usarse si dicho atributo es mayor o igual a uno.

4.3 Planificador

Las acciones que cada personaje realizará durante el transcurso de la historia están dirigidas según sus objetivos (carácter proactivo). Según estos objetivos, el personaje en cuestión deberá seguir un plan de acciones para poder completarlo. En principio, se pensó en utilizar un sólo planificador, en el agente director, de forma que cada personaje le pidiera el plan a éste. En seguida nos dimos cuenta de que el tiempo de ejecución siguiendo esta idea se hacía demasiado costoso, además de que la generación de planes para agentes con intereses en conflicto se haría difícil. Por esta razón, decidimos que cada agente tuviera su propia referencia al planificador. De esta forma, en caso de conflicto, sólo uno de los agentes implicados podrá seguir con su plan inicial, mientras que el resto tendrán que re-planificar a partir del estado actual de la historia. Esto hará que las historias sean mucho más complejas, ya que presentarán conflictos y cambios de planes.

Actualmente, los agentes realizan los planes de forma secuencial (un agente planifica y después ejecuta el plan al completo, después el siguiente planifica y ejecuta su plan, etc…) para evitar que se creen conflictos entre los objetivos por el momento. Esto reduce la riqueza de las historias generadas, pero nos es suficiente como primera aproximación.

4.3.1 JavaFF

JavaFF proporciona una implementación open source en Java del algoritmo de planificación *Fast Forward* [19]. Este planificador toma como entradas el problema y el dominio en PDDL, y escribe el resultado en un archivo. Además, al ser open source, nos ha permitido adaptarlo a nuestro proyecto, para evitar que escribiera los planes a un fichero, y en su lugar, los devolviera como un string, en un formato que nos resultase más sencillo manejar, con las acciones a seguir.

El algoritmo *Fast Forward*, que está basado en el planificador HSP (Heuristic Search Planner), los problemas se intentan solucionar mediante el uso de la planificación hacia adelante en el espacio de estados, con la ayuda de una función heurística extraída directamente del dominio. Se puede ver a FF como el sucesor del HSP, aunque difiere de su predecesor en algunos detalles:

- Un método más sofisticado de evaluación heurística, que tiene en cuenta la interacción entre hechos.

- Un nuevo tipo de estrategia de búsqueda local, que hace uso de la búsqueda sistematica para evitar caer en mínimos locales.

- Un método que ayuda a identificar aquellos nodos más prometedores.

4.3.2.1 Heurística

Una de las formas más comunes de derivar una heurística de un problema es mediante la simplificación del mismo, de forma que se pueda solucionar de manera eficiente. Usando esta solución simplificada, podemos estimar el coste de la solución real, de forma que sigamos primero los nodos más prometedores de entre el espacio de estados posibles. Por ejemplo, en el caso del problema del puzzle de 8, una posible simplificación del mismo es permitir que las fichas se muevan a cualquier posición vecina. La función de coste óptimo de dicho problema simplificado resulta ser la conocida heurística de la distancia Manhattan.

En el caso de la planificación Strips, obtenemos los valores heurísticos de un problema considerando la versión simplificada, en el que las listas de eliminación son ignoradas. Como resultado de lo anterior, las acciones pueden añadir nuevos predicados, pero no eliminarlos, haciendo que el problema simplificado se solucione cuando se generan todos los predicados objetivo. Por ejemplo, supongamos que tenemos una acción que mueve un robot de un punto A a un punto B. Dicha acción tendrá como precondición el hecho de que el robot esté en el punto A. Una vez terminada, la acción añadirá un hecho diciendo que el robot está en B, y eliminará la que dice que está en A. En la versión simplificada de ese problema, al final de la ejecución de la acción tendríamos que el robot está en A y también en B.

No es difícil ver que el coste de la solución simplificada siempre va a estar por debajo del coste del problema inicial, por lo que podríamos usar el coste simplificado como heurística. No obstante, computar dicho coste simplificado sigue siendo un problema NP-difícil [20]. En el caso del HSP, se utilizaba una aproximación a dicha función de coste, en función de unos pesos asignados.

La diferencia de FF con HSP reside en el siguente hecho: mientras que calcular el coste óptimo de la solución simplificada es NP-difícil, decidir entre si tiene solución o no está en P [21], y por tanto existe un algoritmo de decisión en tiempo polinomial. Un algoritmo que utiliza este es el bien conocido Graphplan [22]. Funciona de la siguiente manera: partiendo de un estado cualquiera (en un problema simplificado como el explicado anteriormente), se construye el grafo de planificación hasta que se cumplan todos los objetivos. En este grafo, se van intercalando capas de hechos y acciones. La primera capa de hechos contiene los hechos del estado inicial, y la primera capa de acciones todas las acciones aplicables a dicho estado. Los hechos que se añaden como efecto de la ejecución de las acciones, junto con los que estaban, forman la segunda capa de hechos. A esta capa se le vuelven a aplicar todas las acciones hasta que se llegue a una capa de hechos que contenga todos los objetivos. Una vez en este punto, se puede extraer un plan del problema simplificado de la siguiente manera: para cada capa, y empezando en la última, se procesan todos los objetivos. Si el objetivo está también presente en la capa anterior (como hecho), entonces se inserta en la lista de objetivos a cumplirse en la capa anterior. Si no, entonces se selecciona una acción de la capa anterior que dé como resultado el objetivo, y se añaden las precondiciones de dicha acción a los objetivos a cumplir en la capa anterior. Una vez procesados todos los objetivos de la capa actual, se pasa a procesar los de la capa anterior de la misma forma, parando al llegar a la capa inicial. Se estima el coste de la solución por el número de acciones realizadas.

4.3.2.2 Búsqueda

Mientras que su predecesor utiliza el algoritmo de escalada simple, *Fast Forward* utiliza una versión

más potente del mismo. En caso de no encontrar ningún nodo sucesor que mejore la heurística actual, se irá un paso más allá, para compararlo con la heurística de los sucesores de los sucesores. Si no, irá dos pasos más allá, y así sucesivamente. Cuando se encuentra un estado con mejor heurística, se sustituye el actual por el mismo, y se repite el proceso a partir de dicho estado.

4.3.2 Acciones

Como hemos mencionado antes, la función de planificar de la superclase personaje se encarga, además de planificar, de parsear el plan devuelto por el planificador y ejecutarlo. Para ello, la función lee las acciones contenidas en el String devuelto, creando las acciones pertinentes.

Las acciones son las clases en las que recae la ejecución. Se componen de una serie de atributos (que varían según la acción concreta, y que son necesarios para su correcta ejecución), y una función execute (de tipo void para todas las acciones salvo para de secuestrar, que será boolean), que llevará a cabo las acciones pertinentes.

Una vez obtenido el plan a seguir, el personaje se encargará de crear el objeto de tipo acción correspondiente a la acción parseada, y llamará a su función execute para que se lleve a cabo.

Nuestro sistema consta de seis acciones diferentes, por el momento:

- Batalla: Esta acción se lleva a cabo cuando un personaje secundario (caballeros o dragones), quiere enfrentarse con otro. Posee dos argumentos: el nombre del personaje que inicia la batalla y el de aquel contra el que se enfrenta. Cuando comienza la batalla, se crea un mensaje de tipo INFORM que se enviará al oponente. En el contenido del mensaje, figura la cantidad de salud que se restará al otro personaje (en este caso, hemos decidido que sea la cantidad de salud que posee el personaje para simplificar las cosas). Tras enviarlo, la acción se quedará bloqueada hasta que reciba la respuesta del oponente (que al igual que antes, por simplicidad, pasará su salud) y restará la cantidad su salud total. En caso de haber muerto en combate, lógicamente, dejará de ejecutar acciones, y se enviará el mensaje INFORM de fin de planificación.

- ConvertirseEnHéroe: Esta acción se lleva a cabo cuando el caballero ha rescatado a la princesa y la ha puesto a salvo en su casa. Posee dos atributos: el personaje que la ejecuta y una referencia al agente mundo.

  Esta acción simplemente se encarga de enviar al agente mundo un mensaje de tipo REQUEST, pidiéndole que le añada como héroe al estado. Después informará por pantalla de que el caballero se ha convertido en héroe, y llamará al logger para que lo escriba también en el archivo de log del sistema.

- DejarEnCasa: Esta acción se realiza cuando el caballero ha rescatado a la princesa de la custodia del dragón, esté con ella, y se disponga a ponerla a salvo para convertirse en héroe. Posee tres atributos: el personaje que la realiza, el nombre de la princesa a la que escoltará, y una referencia al agente mundo.

La acción enviará un mensaje tipo REQUEST al agente mundo pidiéndole que actualice el estado poniendo a la princesa como rescatada, y se bloqueará hasta que reciba la respuesta del agente mundo. Tras esto, informará por pantalla y llamará al logger para que escriba la información pertinente en el archivo de log del sistema.

- LiberarPrincesa: Esta acción se ejecutará cuando el caballero haya derrotado al dragón que tenía prisionera a la princesa, y aún no "esté con ella". Posee cuatro atributos: el personaje que ejecutará la acción (caballero), dos strings con los nombres de la princesa y el dragón correspondientes, y una referencia al agente mundo.

  La acción enviará un mensaje de tipo REQUEST al agente mundo, pidiéndole que libere a la princesa (lo que disparará el comportamiento LiberarPrincesa de dicho agente). Después se quedará bloqueado en espera de la respuesta del agente mundo. Por último, informará por pantalla, y llamará al logger para que escriba en el archivo de log del sistema la información pertinente.

- Mover: Como es obvio, esta acción se realiza cuando un personaje desea cambiar de localización. Posee cuatro parámetros: el personaje que la realiza, dos strings indicando los nombres de las localizaciones origen y destino y una referencia al agente mundo

  En caso de que la localización origen sea distinta a la localización actual del personaje, la acción llamará al logger para que escriba un error en el archivo de log del sistema, y escribirá por pantalla informando de dicho error. En caso contrario, enviará un mensaje de tipo REQUEST al agente mundo, pidiéndole que le cambie de posición, con la clase del personaje y las localizaciones destino y origen, como contenido del mismo. Esto disparará el comportamiento LocalizarPersonajes del agente mundo. Después, se bloqueará hasta que reciba la respuesta.

  Tras ésto, se pasará a comprobar el tipo de mensaje recibido en la respuesta. En caso de que no sea CONFIRM, la acción escribirá por pantalla infomando del error, y llamará al logger para que actualice el log del sistema con el error. En caso contrario, el personaje cambiará su localización actual, se informará por pantalla de que el personaje se ha movido a otra localización y se actualizará el log del sistema.

- Secuestrar: Esta acción se llevará a cabo cuando el dragón se dispone a secuestrar a la princesa. Posee tres atributos: el personaje que realizará la acción (dragón), un string con el nombre de la princesa que será secuestrada y una referencia al agente mundo.

  En este caso, la función execute será de tipo boolean en vez de void, para indicar al dragón si ha podido secuestrar a la princesa o no (de modo que tenga que volver a buscar princesas con su respectivo comportamiento). La acción se encargará de enviarle un mensaje al agente mundo informando de que se va a secuestrar a la princesa. Tras ésto, la acción se bloqueará en espera de la respuesta que le den. Si en el contenido de la respuesta recibida se informa de un fallo, entonces se devolverá false. En caso contrario, se informará por pantalla de que el dragón ha secuestrado a la princesa, y se actualizará el log del sistema pertinentemente.

4.4 Logger

Como logger para nuestra aplicación, nos hemos decantado por la herramienta *log4j,* la cual nos permite iniciar el registro durante la ejecución. Además nos permite decidir (vía un archivo de configuración) qué es lo que queremos registrar (en nuestro caso, básicamente todo), el layout, y qué recoger en cada entrada (fecha, acción, agente, etc…). Todo esto será guardado en un archivo de log.

4.5 Flujo de ejecución

Al comienzo de la ejecución, se crea el agente director, que será el encargado de crear primero al agente mundo, y después a los personajes de la historia. Cuando un personaje se crea, lo primero que hace es confirmar al agente director que se ha iniciado correctamente para, después, localizarse en el mapa. Esto se consigue mediante el paso de mensajes ACL. Para localizarse, cada personaje enviará un mensaje al agente mundo. Después, se registran en las páginas amarillas (estructura en la que los agentes se registran para ofrecer servicios al resto de agentes) aquellos personajes que presten un servicio, con un servicio distinto según el personaje. La princesa se registra como “secuestrable” y el caballero registra que puede rescatar princesas. El dragón y el rey no necesitan registrarse, si no que buscarán los servicios de la princesa y del caballero respectivamente llegado el momento. Además el agente mundo se encargará de cargar el mapa. Cuando el dragón comienza su ejecución, comienza a buscar princesas secuestrables en las páginas amarillas, hasta que encuentre al menos una. En ese momento, secuestrará una de las disponibles, y la retendrá en su guarida. En cuanto al caballero, comienza en espera de que el rey pida sus servicios. Una vez contratado, buscará a la princesa e intentará derrotar al dragón que la custodia. Cuando se produzca, llevará a la princesa de vuelta al castillo, después volverá a su casa, y se convertirá en héroe, terminando así la historia.

Como ya hemos comentado, este es el flujo básico de ejecución (ver figura 14), que se dará siempre y cuando el dragón consiga secuestrar a la princesa que seleccione, el caballero encuentre a dicha princesa, y logre derrotar al dragón. Si una de esas cosas falla, se volverá a algún comportamiento previo de uno de los agentes, para volver a intentar terminar la historia. Bien sea buscando una princesa distinta, o contratando otro caballero, etc.

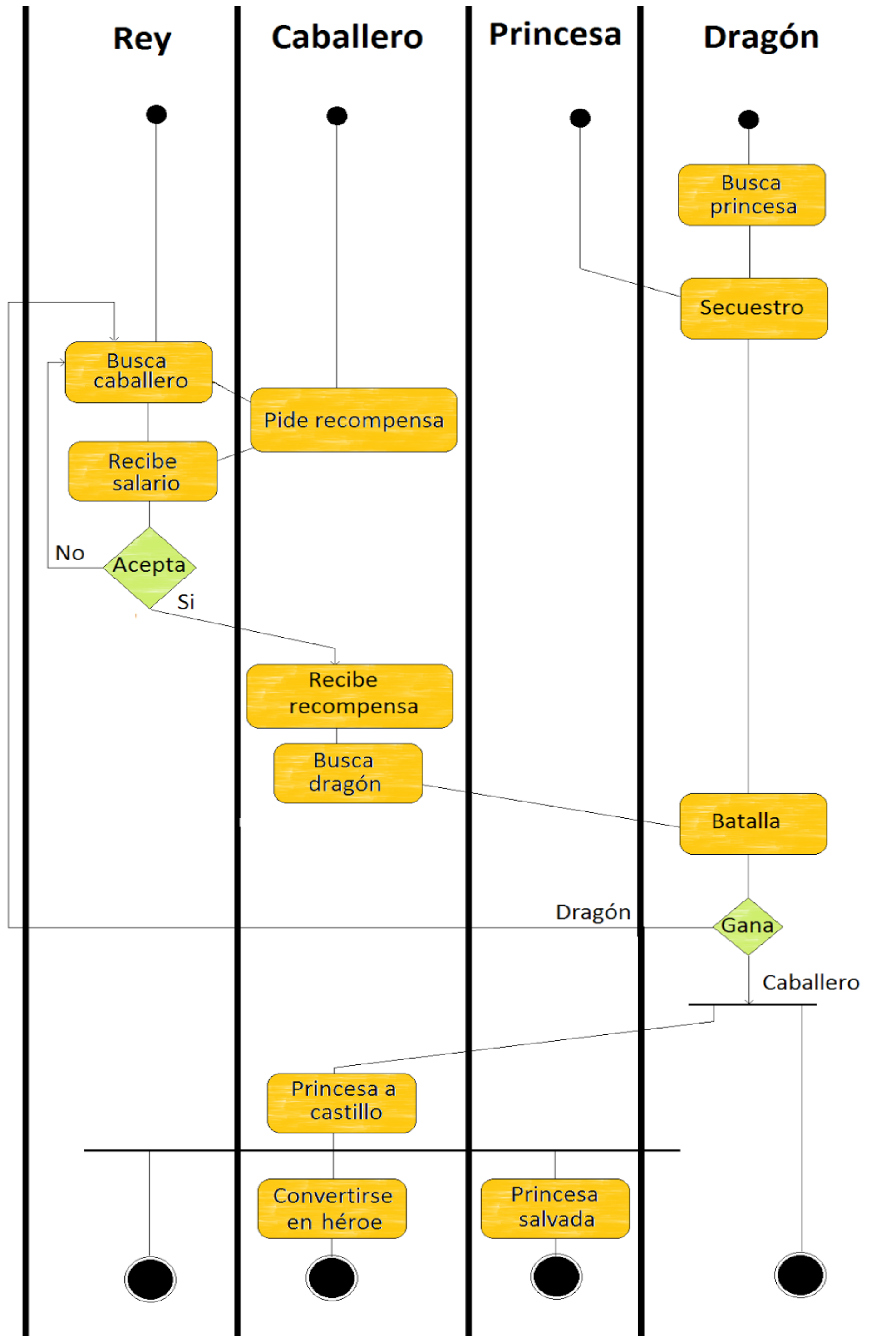

Figura 14: diagrama de actividad de una historia básica

CAPÍTULO 5: Resultados

Se ha implementado un primer prototipo de un generador de historias, capaz de generar diferentes historias dentro del ámbito de dragones y princesas, consiguiendo que todos los componentes anteriormente descritos trabajen juntos. Somos capaces de generar variaciones simples en torno a una historia en la que el dragón secuestra a la princesa, y el rey busca un caballero que la rescate. De momento, los personajes son fijos, es decir, funciona con los personajes que hemos descrito anteriormente. A continuación, pondremos algunos ejemplos de historias que se pueden generar:

Ejemplo 1

```
El Rey Rey está preparado.
La Princesa Princesita despierta.
La Princesa sale del castillo.
El dragón Draco emprende el vuelo en busca de alguna princesa desprotegida.
Las siguientes princesas son secuestrables:
Princesita
La Princesa Princesita ha sido secuestrada.
Intentando pedir rescate para la princesa Princesita
Encontrados los siguientes caballeros:
El caballero Paquito entra en escena.
Intentando pedir rescate para la princesa Princesita
Encontrados los siguientes caballeros:
Paquito
El dragón Draco ha muerto en batalla.
La Princesa Princesita fue liberada.
El Rey entrega 50 monedas al caballero Paquito
La Princesa llega al castillo con el caballero Paquito.
La Princesa Princesita pone fin a su aventura.
```

Ejemplo 2

```
La Princesa Princesita despierta.
El caballero Juanito entra en escena.
El Rey Rey está preparado.
El dragón Draco emprende el vuelo en busca de alguna princesa desprotegida.
El caballero Paquito entra en escena.
Las siguientes princesas son secuestrables:
    Princesita
Draco ha llegado a Castillo.
Draco ha secuestrado a Princesita.
Princesita ha llegado a Montana.
Draco ha llegado a Montana.
La Princesa Princesita ha sido secuestrada.
Intentando pedir rescate para la princesa Princesita.
```

```
Encontrados los siguientes caballeros:
    Paquito
    Juanito
Paquito ha llegado a Montana.
El dragón Draco ha muerto en batalla.
El caballero Paquito ha liberado a la princesa Princesita.
Princesita ha llegado a Castillo.
Paquito ha llegado a Castillo.
El caballero Paquito ha dejado a la princesa Princesita en su castillo.
Paquito ha llegado a Pueblo.
El caballero Paquito se ha convertido en héroe.
La Princesa Princesita fue liberada.
El Rey entrega 50 monedas al caballero Paquito.
La Princesa Princesita pone fin a su aventura.
```

Ejemplo 3

```
El caballero Juanito entra en escena.
El caballero Paquito entra en escena.
El Rey Rey está preparado.
La Princesa Princesita despierta.
El dragón Draco emprende el vuelo en busca de alguna princesa desprotegida.
Las siguientes princesas son secuestrables:
    Princesita
Draco ha llegado a Castillo.
Draco ha secuestrado a Princesita.
Princesita ha llegado a Montana.
Draco ha llegado a Montana.
La Princesa Princesita ha sido secuestrada.
Intentando pedir rescate para la princesa Princesita.
Encontrados los siguientes caballeros:
    Paquito
    Juanito
Paquito ha llegado a Montana.
El caballero Paquito ha muerto en combate.
El Rey recibe la noticia de la muerte del caballero, así que busca a otro.
Intentando pedir rescate para la princesa Princesita.
Encontrados los siguientes caballeros:
    Juanito
Juanito ha llegado a Montana.
El dragón Draco ha muerto en batalla.
El caballero Juanito ha liberado a la princesa Princesita.
Princesita ha llegado a Castillo.
Juanito ha llegado a Castillo.
```

```
El caballero Juanito ha dejado a la princesa Princesita en su castillo.
Juanito ha llegado a Pueblo.
El caballero Juanito se ha convertido en héroe.
La Princesa Princesita fue liberada.
El Rey entrega 50 monedas al caballero Juanito.
La Princesa Princesita pone fin a su aventura.
```

Hemos podido comprobar que es sencillo modificar el mapa, añadiendo nuevas localizaciones (por ser un archivo XML externo) y situar a los personajes en éstas, de forma que los personajes tengan que realizar travesías más largas para poder cumplir sus objetivos. También nos es fácil añadir personajes de un tipo existente, por ejemplo, añadiendo un segundo dragón que intente quitarle la princesa al primero, o varios caballeros con distintas características para que el rey elija.

Para poder añadir nuevos tipos de personaje, es necesario modificar el código fuente de manera directa, además del conocimiento presente en el dominio, pero el código se ha estructurado de manera que estos cambios sean relativamente fáciles de implementar.

Aún no hemos comprobado cómo de fácil sería generar historias dentro de otro dominio, como por ejemplo historias de vaqueros o de amor, pero tal y como lo vemos ahora, sería un trabajo más o menos sencillo, pero bastante largo, ya que hay que crear un archivo de dominio totalmente nuevo, personajes, acciones….

Como resultado final, queremos destacar que hemos publicado un artículo en el que hablamos de este sistema en la conferencia ICCC 2014 Late Breaking Papers. Tras pasar el filtro de rechazados, no pudimos ir a la exposición del mismo, que se celebró en Ljubljana, Eslovenia. En nuestro lugar, fué nuestro codirector Pablo Gervás. Allí se pudo ver una presentación explicativa del sistema. Está disponible una copia en los apéndices de esta memoria.

5.1 Conclusiones

Intentar generar creatividad computacionalmente es un trabajo en el que aún se tiene que avanzar bastante, ya que hasta que no seamos capaces de comprender cómo funciona nuestro cerebro, no seremos capaces de imitarlo, y eso se aplica también a la creatividad.

Sin embargo, con los conocimientos de los que disponemos, ya somos capaces de “inventar” cosas de manera automática, basándonos en éste caso en simulaciones y aleatoriedad. No obstante, cabe mencionar que es un trabajo bastante arduo y complejo, ya que para generar pequeñas historias con un dominio fijado, hemos necesitado un año de trabajo en el tema. Además es una tarea que tiene distintos enfoques de avance. Por un lado, podríamos dedicarnos simplemente a ampliar el mapa y añadir personajes, de tipos ya existentes y de nuevos tipos, y seríamos capaces de generar historias mucho más largas y complejas con modificaciones relativamente sencillas. Por otro lado, podríamos hacer más completos a los personajes, añadiendo sentimientos o estados de ánimo, lo que haría que nuestras historias fueran mucho más “orgánicas” y realistas, pero sería bastante complejo de implementar.

Hemos podido ver que el campo de la generación de historias es uno con multitud de enfoques diferentes, siendo el que nos ocupa de la simulación basada en agentes, uno de los más interesantes a nuestro parecer. Sin embargo, las implementaciones ya existentes basadas en éste enfoque, se concentraban en diferentes características, unos añadían sentimientos a los personajes, otros les daban la capacidad de crear objetos que antes no estaban, etc… por lo que para nuestra aplicación, tuvimos que "innovar" en función de los objetivos que queríamos cumplir.

Por otro lado, hemos podido comprobar que la planificación es todo un campo de estudio en sí mismo. Por el momento, hemos decidido pasar "de puntillas" por el tema, usando un planificador open source basado en STRIPS con hipótesis de mundo cerrado. Aún así, resultaría crucial para algunos de los trabajos futuros que a continuación propondremos, investigar más este tema, ya que actualmente no podemos realizar cuestiones como planificación conjunta.

Nuestros objetivos eran:

- Diseñar un generador de historias que sea capaz de generar varias historias diferentes. Hemos cumplido este objetivo, ya que nuestra aplicación es capaz de generar historias diferentes e independientes (ver sección resultados).

- Hacer el generador de historias configurable. Hemos cumplido este objetivo, ya que, como se puede ver, hemos incluido la posibilidad de ficheros de configuración para las acciones y el mapa.

- Hacer un generador capaz de cambiar el contexto de la historia mediante un fichero sencillo de configuración. Este objetivo no se ha podido cumplir debido a que se ha prestado más atención a la propia generación de la historia, de modo que las historias resultantes fueran más complejas, ya que en cierta medida, la aplicación es bastante configurable.

- Hacer que los personajes utilicen la planificación, o replanificación en caso necesario. Este objetivo se ha cumplido parcialmente. Nuestros personajes son capaces de usar el planificador proporcionado con el programa para cumplir sus objetivos, aunque debido a la simplicidad de las historias generadas, no se producen conflictos que exijan replanificación.

5.2 Discusión

Para la realización de este proyecto nos hemos basado en algunas de las ideas expuestas en la parte de trabajos previos. Por estar nuestro trabajo basado en la simulación con agentes inteligentes, es evidente que nuestra mayor fuente de inspiración ha sido el Virtual Storyteller, ya que Thespian, aunque posee bastante en común también, se basa en la interactividad. En este caso, nuestra capa de presentación es una aplicación externa (cortesía de Pablo Gervás) que se encarga de tomar como entrada los eventos que se han producido durante la historia, y genera frases en lenguaje natural. Puesto que nuestro programa genera, por el momento, historias muy cortas, no habría capa de argumento, ya que tomamos todas las acciones que se producen. Aunque nuestros personajes son mucho más simples, ambos trabajos coinciden en la simulación mediante agentes como vía principal para generar historias. Thespian funciona de la misma forma, con la salvedad de la inclusión del

usuario como un personaje más. Ambos trabajos utilizan un planificador de orden parcial.

Sin embargo, en nuestra aplicación no disponemos por el momento de un agente director que cuide que el argumento sea interesante, al contrario que en Virtual Storyteller. Debido a la longitud de nuestras historias, por el momento esto no es un problema, pero deberemos tratar este problema a la hora de generar historias más largas y complejas.

Aunque el agente director del Virtual Storyteller es mucho más completo que el nuestro, podemos destacar una diferencia significativa en el modo en el que luego se narrarán los hechos. En el Virtual Storyteller, el agente director debe ser capaz, no sólo de redirigir la historia en caso de que sea necesario, si no de construir una red causal con todos los elementos de la historia (hechos, personajes, …). Esto se debe a que el módulo de presentación que se propone necesita de dicha estructura para poder contar la historia. Sin embargo, en nuestro caso, disponemos de un logger que recoge las acciones, con sus detalles (fecha, hora, agente, acción o mensaje, con su contenido…). Estos son los datos que usa la aplicación externa para construir las sentencias en lenguaje natural.

A pesar de que nuestras historias son incluso más simples que las que ya era capaz de generar el Novel Writer en 1977, se ha intentado en todo momento que el código esté bien estructurado y lo menos cableado posible, para facilitar las expansiones y trabajos futuros que propondremos. Por el momento, sólo somos capaces de generar un tipo de historia, aunque hemos tenido en cuenta a la hora de programar, que una de nuestras principales aspiraciones era la de hacer un generador de historias que pudiese cambiar de dominio.

Al igual que Talespin, nuestra aplicación utiliza un mapa físico para computar la historia, de forma que los personajes tengan que moverse entre localizaciones. También coincide en el hecho de que nuestra aplicación permite más de un personaje solucionador de problemas (permite tener varios caballeros). Sin embargo, nuestros personajes no poseen relaciones afectivas, estados de ánimo, ni ninguna variable que pueda condicionar su comportamiento de una u otra manera (la cantidad de salud de caballeros y dragones es aleatoria, pero no entra dentro de lo que queremos expresar aquí).

5.3 Trabajo futuro

Debido a la magnitud del proyecto, algunos de los objetivos inicialmente propuestos no han podido ser cumplidos. Por ello, dejamos para trabajo futuro éstos y otras ampliaciones que creemos serían muy interesantes de implementar, ya que aumentarían la complejidad de las historias generadas significativamente. Algunas eran parte de nuestros objetivos iniciales, pero no pudieron completarse debido a que concentramos nuestra atención en otros.

Añadir paralelismo

Actualmente, los agentes de nuestra historia funcionan de manera secuencial, es decir, planifican y después ejecutan su plan sin que ningún otro agente pueda hacer nada. Esto se debe a que simplifica la lógica de la aplicación, y lo tomamos como una primera aproximación.

Sin embargo, sería interesante que todos los personajes funcionasen en paralelo, dando lugar a conflictos. En caso de que la realización de un plan implique la no realización de otro, uno de los

personajes no podrá seguir con su plan y tendrá que replanificar a partir del estado vigente en el momento del conflicto.

En un principio, el tratamiento de conflictos podría ser aleatorio, es decir, de los implicados se elige uno al azar que será el que prosiga con su plan inicial, haciendo planificar al resto. No obstante, una opción muy interesante sería la de hacer que los personajes “compitan” por los recursos. Por ejemplo, si hay dos caballeros que entran en conflicto, podríamos decir que se peleen para ver quién es el que sigue adelante.

Estimamos que este (en mayor o menor medida) debería ser una de las primeras modificaciones a incluir en nuestro programa, ya que añadiría mucha riqueza sin necesidad de modificar demasiado el código fuente.

Cambio de dominio

Uno de los objetivos que nos planteamos inicialmente para este proyecto fue intentar que fuera muy configurable. Sin embargo, una de las cosas que no hemos tenido tiempo de implementar es el cambio de dominio. Nuestra idea en principio era que el programa tuviera un fichero de configuración en el que se especificara el tipo de historia que se quería (una de vaqueros, de piratas, …).

Inicialmente, como se ha descrito anteriormente en esta memoria, comenzamos a implementar el programa de forma que nos fuera sencillo hacerlo en el futuro. Nuestra idea inicial era tener tipos de personajes en lugar de personajes concretos (como tipo héroe, villano, etc…) y obtener el tipo concreto de personajes desde el archivo de configuración mencionado anteriormente.

Expandir el mapa

Por supuesto, añadiendo complejidad al mapa obtendremos historias diferentes y más complejas. El simple hecho de añadir localizaciones haría a los personajes moverse más, pudiendo ocurrirles cosas por el camino.

Además, podría hacerse un mapa por niveles. Es decir, que cada localización tuviera a su vez un mapa, de modo que podríamos dividir el castillo en habitaciones, el pueblo en barrios, etc… De esta forma, los planes serían más complejos, ya que para, por ejemplo, cruzar el pueblo, habría que ver el camino más corto entre los barrios. Así, además, la guarida del dragón podría ser más compleja, haciendo que el caballero tenga que buscar dentro al dragón. Si a esto le añadiésemos la posibilidad de que el dragón se fuera moviendo por la cueva (aleatoriamente al principio, pero siguiendo algún tipo de patrón de huída del caballero más tarde), la historia se haría mucho más completa.

Así mismo, se pueden añadir objetos a las localizaciones, como nos habíamos propuesto, de forma que los personajes puedan interactuar con dichos objetos, haciendo a las historias mucho más complejas. Así, por ejemplo, podríamos decir que el caballero sólo puede matar al dragón con la “espada mágica” que está guardada bajo llave, y podríamos colocar la llave en alguna localización del mapa. De esta forma, cuando el caballero sea contratado por el rey, deberá incluir la búsqueda de la llave, el ir a por la espada y el matar al dragón en su plan.

Expandir personajes

Como ya hemos mencionado con anterioridad, la inclusión de varios personajes de una misma clase haría las historias más complejas. Por ejemplo, añadiendo más caballeros, princesas y dragones, habría varias historias coexistiendo.

También podemos crear nuevos personajes, modificando el archivo XML con los objetivos de cada personaje, y añadiendo las acciones correspondientes al archivo PDDL de dominio. A su vez, deberíamos asociar estas acciones a clases en Java que las implementen, para poder ejecutarlas.

Sin embargo, podemos ir un paso más allá complicando la historia haciendo que los personajes sean más complejos. Como hemos mencionado arriba, se podría añadir al dragón la capacidad de huir del caballero una vez que éste le sigue para darle caza. En principio podría ser moviéndose aleatoriamente, de forma que no habría que implementar más cosas en el caballero aparte de replanificar si se ha llegado a la localización donde estaba el dragón y éste ha huido. Para complicarlo más, podríamos hacer que el dragón siguiera un patrón de huida, de forma que cuando "vea" al caballero, huya a otra localización. De esta forma, un sólo caballero nunca sería capaz de atraparle. Por ello, podríamos hacer que varios caballeros le "acorralen".

Para complicar aún más las cosas, se podrían añadir estados de ánimo, siendo en un principio simplemente un atributo que guarde este estado, iniciado aleatoriamente, haciendo así que, por ejemplo, ejecute unas acciones u otras según como esté. También podríamos hacer que estos personajes posean un equipamiento, con diferentes propiedades según el equipamiento concreto. Es decir, como en el caso de los caballeros, que tengan un arma concreta y una coraza concreta. Así, podría no ser suficiente para cumplir sus objetivos con el equipamiento que en ese momento posean, haciendo que deban incluir en su plan la búsqueda de equipamiento mejor (inicialmente podría ser simplemente cambiarlo, pero más adelante podrían crearse tiendas en las que el personaje tenga para comprar el equipamiento deseado).

Interactividad

Una funcionalidad interesante podría ser la de dar al usuario el papel de alguno de los personajes. De esta forma, se podría alterar la historia en el momento. Nos parece, además, que sería una modificación relativamente sencilla, ya que las acciones ya están implementadas en clases Java. Por tanto, sería suficiente con dar al jugador a elegir entre las posibles acciones que puede ejecutar el personaje en cuestión, sin cambiar nada más tal y como está el proyecto, o parando todos los hilos mientras se toma la entrada en caso de que los agentes trabajen en paralelo.

Expansión del agente director

Ahora mismo, nuestro agente director es el encargado de crear al resto de los agentes. Como posible expansión, proponemos que el agente vaya "alterando" la historia.

Tomando como ejemplo el agente director del Virtual Storyteller, nuestro agente director debería conocer la estructura básica de una historia, y ser capaz de actuar sobre las acciones de los personajes para que la historia sea más interesante, por ejemplo, negando al caballero el ir por un determinado camino, haciendo que deba escoger otro por el que está su enemigo, etc.

5.4 Aportaciones de los miembros

5.4.1 Parte Jose Luís Ledesma López

Inicialmente leí varios de los artículos que se mencionan en la bibliografía, para tener una idea de que se había probado o conseguido en este ámbito. Tras aclarar las ideas de por donde íbamos a orientar el proyecto, me informe sobre el uso de agentes y los pasos de mensajes entre ellos. Trabaje con la documentación de JADE y los ejemplos que la acompañan. Uno de ellos en concreto, fue el que ayudó a realizar la primera versión del programa, ya que trataba sobre unos vendedores y un comprador, donde este último pedía ofertas por un determinado libro. Así que basándome en los comportamientos de esos dos agentes, creé una relación parecida entre el Rey y el Caballero, donde el primero pedía ofertas a las instancias del segundo, para que rescataran a la Princesa. Lógicamente eran mucho más simples que ahora, ya que sólo mandaban mensajes entre ellos, y en un orden establecido.

Mi compañero y yo nos dividimos la estructura que queríamos dar al comportamiento del Rey, para hacer comportamientos más pequeños y poder construir la máquina de estados que ahora posee. Una vez con todos los comportamientos terminados, me encargué de construir el FSMBehaviour con todas las transiciones entre los estados. Después cree la clase Princesa, que usé como disparador para el comportamiento del Rey que habíamos construido. Además que el Rey terminase su comportamiento mandando un mensaje a la Princesa para que supiera que estaba liberada.

Junto con el personaje Dragón, que creó Iván, teníamos todos los personajes creados, así que nos pusimos con el entorno. Aprovechando unos archivos que hicimos para una práctica de una asignatura, reutilice la parte del mapa, en las que añadí ligeras variaciones para que se adecuara a nuestro caso, aunque todavía los personajes no se podían mover por el. Tras una reunión con el director de proyecto, optamos por cambiar la forma de representar el mapa, por lo que cambiamos sus respectivas clases, algunas incluso suprimiéndolas, como ocurrió con la clase Camino. También, como queríamos que fuera lo más configurable posible, cambiamos el formato de entrada del mapa, que anteriormente era uno establecido por nosotros. Así que creamos el mapa en formato XML, que resultaba más claro, y además más sencillo de modificar para crear distintos mapas. Ayudé a Iván a cambiar el cargador del mapa que había construido, aunque finalmente usamos las librerías de Java para tratar con archivos XML, así que hicimos el método desde cero.

Como queríamos usar un programa externo para que usará los hechos de nuestra historia como entrada, y así pudiera narrarla, necesitábamos que nuestro prototipo hiciera un log de lo que sucediese entre los agentes. Tras decantarnos por log4j, estudié su funcionamiento y como indicar que guardase lo que nos interesaba, usando el archivo *log4j.properties* para tal fin. Incluí en el proyecto su librería e hice los cambios pertinentes en el código para que fuera escribiendo la información que nos interesaba. Lo siguiente que hicimos fue buscar planificadores que usaran PDDL. Encontramos el planificador JavaFF, que además podíamos modificar al disponer de su código. Generé los archivos del dominio del problema, y los dos correspondientes a los problemas del Dragón y Caballero. Una vez comprobado que el planificador devolvía el plan correctamente en ambos casos, procedimos a modificar el código del planificador JavaFF de tal forma que nos lo devolviera en un formato que pudiéramos manejar. Añadí el planificador al proyecto, de tal forma que pasándole el nombre del archivo del dominio y el de un problema, obtuviéramos el plan, y así poder trabajar con él.

Una vez incluído el planificador y que Iván creará el Estado, procedimos a modificar el AgenteMundo para que, según los mensajes que recibía de los personajes, corrigiese el estado en consecuencia. En algunas de las modificaciones tuve que realizar unos nuevos cambios, ya que podíamos simplificar al AgenteMundo usando comportamientos que ya tenía de antes, y que siguiera haciendo los mismos cambios en el estado. También extendí el Estado, para que guardara algunos datos más, que eran necesarios para generar el estado inicial del problema que se pasaría al planificador. Lo único que faltaba era que los personajes supieran cuáles serían sus objetivos, y ya que usamos XML para crear los mapas, pensamos que era buena idea usarlo de nuevo. De esta forma se conseguía que fuese más configurable, pudiendo incluir distintos objetivos para un mismo personaje y seleccionar uno u otro dependiendo del momento de la historia. Ya que íbamos a generar todo el archivo del problema en PDDL, creé el XML con los objetivos directamente en dicho lenguaje, para no tener que hacer un proceso entre medias, ya que además el lenguaje es bastante claro y está muy estandarizado.

Ya teníamos el planificador funcionando, y lo que restaba por hacer era ir seleccionando las acciones que devolvía y hacer que los personajes las fueran realizando. En un principio pensé en que fueran comportamientos, pero surgieron problemas a la hora de intercambiar mensajes con el resto de agentes, ya que estaban bloqueados realizando otra acción o incluso pendientes de que finalizara esta misma. Así que cambié las acciones por clases independientes. Cambié también el comportamiento del Dragón donde secuestra a la princesa, ahora buscaría una princesa, y cuando encontrase una, llamaría al método planificar. En este método, lo primero que hice es que llamara al comportamiento del AgenteMundo, que autogeneraba el archivo del problema en función del estado en ese momento y los objetivos del personaje que obtuviera del XML. Después usar el planificador JavaFF con ese archivo junto al del dominio, y del plan que devolviera fuera cogiendo una a una cada acción, y creando y ejecutando la acción correspondiente en el personaje que mandó planificar. Además hice que comprobara que siempre el personaje que mandaba planificar, era el que tenía que ejecutar la acción, y en caso contrario, que replanificara desde el estado en que se quedó. Por último, mandara un mensaje al personaje que llamó a este método para que supiera cuando había terminado de ejecutar el plan.

Faltaba ya por último, volver a modificar los comportamientos de los agentes que iban a planificar. Yo realicé los cambios oportunos en la clase Dragón, haciendo que si recibía un mensaje de fallo en el secuestro, volviera a añadir su comportamiento inicial de buscar princesas, y por tanto volviese a planificar cuando se decidiese por una. Y por el contrario, si recibía el de fin de planificación, es que había conseguido su objetivo y pasaba al comportamiento de defenderse de los caballeros que intentasen liberar a la princesa que había secuestrado.

5.4.2 Parte de Iván Manuel Laclaustra Yebes

Al igual que mi compañero, comencé leyendo los artículos que nuestro director de proyecto nos comenzó a enviar para documentarnos, además de buscar otros por nuestra cuenta. Al principio, en la fase de modificación de un ejemplo de Jade, cree la clase Dragón. En principio este lo único que hacía era esperar a que hubiera una princesa en el sistema para, automáticamente, secuestrarla, de forma que comenzara la historia. Paralelamente, nos dimos cuenta de que el Rey y el Caballero funcionaban igual que una máquina de estados, por lo que decidimos implementarlos de manera que usaran la clase FSMBehavior. Con esta decisión, decidimos dividirnos la transformación de dichos personajes (en mi caso, el Caballero), dividiendo los comportamientos entre ambos miembros del grupo para, después, poder añadirlos a las máquinas de estados correspondientes.

Dado que nuestra idea inicial era que los personajes pudieran utilizar objetos, me dispuse a crearlos. Para ello, me basé en parte del código de una práctica que realizamos Jose Luís y yo en un curso anterior. En principio la idea era que hubiera objetos con los que se pudiera interactuar, pero no pudieran cogerse (como el decorado) y objetos usables por los personajes. Por ello, decidí que los personajes deberían tener una lista de objetos en su poder.

Pensando en que cómo podríamos implementar la función de cambiar de contexto de la historia (lo cual, en principio, era uno de nuestros objetivos), me di cuenta de que una posible forma podría ser que los personajes fuesen representados por clases de personajes, que deberían especificarse en un fichero de configuración con el programa. Así, tendríamos protagonista, antagonista actor secundario, etc… Esta forma de ver la situación, junto con el hecho de que todos los personajes tienen características similares (nombre, objetos, etc...), me llevaron a tomar la decisión de incluir una clase “Personaje”, de la que heredaran todos, aunque en ese momento eran personajes concretos. No obstante, con el paso del tiempo, nos fuimos concentrando más en otros objetivos, por lo que descartamos esta posibilidad, aunque dejamos esta clase para añadir legibilidad y modularidad a nuestro código.

Una vez que teníamos creada la primera versión de la estructura de datos que representaría nuestro mapa, me dispuse a crear un cargador que lo leyera de un fichero de entrada y lo creara siguiendo las indicaciones especificadas. Para ello, una vez más, utilicé parte del código de una práctica pasada. Tras esto, diseñe una estructura que debía seguir el archivo de especificación del mapa, de forma que el cargador pudiera leerlo correctamente. Una vez creado el agente mundo, decidí que tuviese un comportamiento que cargara del mapa. Dicho comportamiento utilizaba el cargador mencionado para crear el mapa que poseía. Sin embargo, durante el proceso de creación del agente mundo nuestro director de proyecto nos sugirió que usáramos archivos XML para especificar el mapa, de forma que fuera mucho más legible para el usuario, por lo que, entre los dos, cambiamos el cargador del mapa. Así, este primer cargador quedó obsoleto y no se usó más, ya que pasamos a utilizar las librerías presentes en Java para el manejo de archivos XML. Con ello, diseñamos entre ambos una nueva estructura del mapa, de forma que el nuevo cargador lo pudiera reconocer.

También teníamos pensado incluir un agente director, de forma que proporcionara los objetivos. Por ello, me dispuse a crear una primera versión (que finalmente ha resultado ser la última) de dicho director, de forma que creara los personajes de la historia, y tuviera un comportamiento asociado a cada uno de ellos para confirmar su correcta inicialización.

Hecho esto, decidimos incluir un planificador para los personajes, por lo que ambos comenzamos a investigar las distintas opciones de planificadores ya implementados, con las restricciones de que debían ser open source, para poder modificarlos, y debían usar el lenguaje PDDL, de forma que en un futuro pudiéramos (nosotros, o futuras personas) cambiarlo por otro de manera sencilla. Una vez decidido que el planificador a utilizar sería *JavaFF* y, una vez se incluyó en el proyecto, nos dispusimos a modificarlo, ya que el planificador escribía los planes generados en un fichero, además de producir múltiples salidas por consola. Decidí en este punto que el planificador devolviera la estructura de datos que manejaba para conocer las acciones del plan, aunque más tarde cambiamos de idea.

Una vez que teníamos el planificador plenamente operativo, me dispuse a cambiar los comportamientos del Caballero, de forma que funcionase utilizando el planificador, y no como una máquina de estados.

En el momento en que Jose Luís decidió usar XML para los objetivos de los personajes, nos dimos cuenta de que debíamos tener el archivo PDDL correspondiente al problema para poder utilizar el planificador. Por ello, me dispuse a añadir un nuevo comportamiento al agente mundo, de forma que se activara en el momento en que un personaje tuviera que planificar, al recibir la petición. Este comportamiento se encargaba de leer el archivo XML correspondiente a los objetivos, y auto generar el archivo PDDL correspondiente al personaje que realizó la petición. Sin embargo, para ello, debía conocer también el estado actual de la historia, de forma que pudiera añadirlo a dicho fichero y planificar según la situación actual. Por esta razón, implementé una primera versión de una estructura de datos llamada Estado, en la que se almacenaban los nombres de todos los personajes de la historia en el momento, las posiciones del mapa y sus adyacentes, las localizaciones de cada uno de los personajes y los personajes vivos. Una vez hecho esto, me dispuse a actualizar los comportamientos del agente mundo, de manera que reflejaran los cambios producidos en el estado en todo momento. En el caso de añadir personajes, decidí que sería un comportamiento en sí mismo, de forma que los personajes al crearse enviaran una petición al agente mundo para que les añadiera, aunque posteriormente decidimos cambiarlo.

5.5 Conclusions

Generating computational creativity is a subject that has yet to advance, as we won't be able to imitate our brain until we're not able to comprehend how it works, and that also applies to creativity.

However, with the knowledge we already have, we're capable of creating things automatically, basing, in this case, in simulations and randomness. Nevertheless, it's good to mention that this is an arduous and complex task, as to generate small stories with a fixed domain, we've needed a year of work in the subject. Furthermore, this subject has different advance approaches. On the one hand, we could just extend the map and the character, being able to generate much longer and more complex stories by just doing relatively simple modifications. On the other hand, we could make the characters themselves more complex, adding feelings and moods, making our stories much more "organic" and realistic, but it would be complex to implement.

We've seen that the subject of storytelling is one with multiple approaches, being at hand the story generation based on agents, one of the most interestings from our point of view. However, existing implementations based in this approach, used to concentrate in different characteristics (some added feelings to the characters, others gave them the ability to create objects that weren't there, etc…), so, for our application, we had to "innovate", according to the goals we wanted to achieve.

On the other hand, we've seen that planification is a whole investigation subject itself. By now, we've been "tiptoeing" around this issue, using an open source planner based in PDDL with closed world assumption. Even so, it'd be crucial for some of the future work we'll describe above, investigating this subject, as right now we're not able to do such things as joint planning.

Our goals were:

- Designing a storyteller that was able to generate various different stories. We've achieved this goal, as our application is capable of generating different and independent stories (see results section).

- Making the storyteller configurable. We've also achieved this goal since, as can be seen, we've included configuration files for the actions and the map.

- Making the storyteller capable of changing the context of the story by a simple configuration file. We've not been able to achieve this goal, since we've payed more attention to the story generation process itself, so the resulting stories were more complex as, to some extent, the application is already very configurable.

- Making the characters able to use planification, or replanification when needed. We've partially achieved this goal. Our characters are capable of using the planner given with the program to achieve their goals, although due to the simplicity of the stories generated, there are no conflicts that require replanification.

Apéndice A: Artículo publicado

En esta sección, adjuntamos una copia del artículo publicado en conferencia ICCC 2014 Late Breaking Papers. Tras pasar el filtro de rechazados, no pudimos ir a la exposición del mismo, que se celebró en Ljubljana, Eslovenia, por lo que nuestro codirector Pablo Gervás fue a representarnos (a nosotros y a los directores).

En el artículo exponemos un pequeño resumen de lo que es nuestro trabajo, con una pequeña descripción de cada uno de los módulos explicados en esta memoria, así como el trabajo relacionado y los resultados obtenidos hasta ese momento.

A continuación, podemos ver el artículo publicado.

Kill the Dragon and Rescue the Princess: Designing a Plan-based Multi-agent Story Generator

Iván M. Laclaustra, José L. Ledesma, Gonzalo Méndez, Pablo Gervás
Facultad de Informática
Universidad Complutense de Madrid
Madrid, Spain
{ilaclaus, josledes, gmendez, pgervas}@ucm.es

Abstract

We describe a prototype of a story generator that uses a multi-agent system and a planner to simulate and generate stories. The objective is to develop a system that is able to produce a wide range of stories by changing its configuration options and the domain knowledge. The resulting prototype is a proof of concept that integrates the simplest pieces that are necessary to generate the stories.

Introduction

When trying to generate stories automatically, it is mandatory to research how actual stories work. That, inevitably, makes you think: "What makes a story interesting?".

While researching for this project, we realized that in a story, most of the times, the most important thing is not WHAT, but HOW things happened. This represents a huge challenge, since it is difficult to simulate things such as time (we must be able to simulate time, so that things are not done immediately), conversations (they have to be fluid, spontaneous), and many more. Similarly, there are some actions that lack interest in themselves, but may have some if combined with others. For example, eating or sleeping, are actions that may not appear in the final story, but may be worthy of attention if the character meets someone while eating. Of course, some of the stories generated will just be sets of facts without any relation or interest, but that is part of the process.

One of the ways we have for generating stories is by simulating them. Then, you just have to run the simulation and see what happens. We achieve this by simulating the stories using autonomous intelligent agents. Each of the agents of the story is going to act as a character, which will act independently from the others, but depending on the story world's state. Then stories are generated by "filming" what these actors do and say. Our main goal for now, is to make a small Dungeons & Dragons story, which has more than one ending.

Related Work

The first story telling system for which there is a record is the Novel Writer system developed by Sheldon Klein (Klein et al. 1973), which created murder stories within the context of a weekend party. It relied on a microsimulation model where the behaviour of individual characters and events were governed by probabilistic rules that progressively changed the state of the simulated world (represented as a semantic network). The flow of the narrative arises from reports on the changing state of the world model. A description of the world in which the story was to take place was provided as input. The particular murderer and victim depended on the character traits specified as input (with an additional random ingredient). The motives arise as a function of the events during the course of the story. The set of rules is highly constraining, and allows for the construction of only one very specific type of story. The world representation allows for reasonably wide modeling of relations between characters. Causality is used by the system to drive the creation of the story but it is not represented explicitly.

TALESPIN (Meehan 1977) is a system which tells stories about the lives of simple woodland creatures. TALESPIN was based on planning: to create a story, a character is given a goal, and then the plan is developed to solve the goal. TALESPIN introduces character goals as triggers for action. Actions are no longer set off directly by satisfaction of their conditions; an initial goal is set, which is decomposed into subgoals and events. TALESPIN introduced the possibility of having more than one problem-solving character in the story (and it introduced separate goal lists for each of them). The validity of a story is established in terms of: existence of a problem, degree of difficulty in solving the problem, and nature or level of problem solved.

Lebowitz's UNIVERSE (Lebowitz 1985) modelled the generation of scripts for a succession of TV soap opera episodes. It aimed at exploring extended story generation, a continuing serial rather than a story with a beginning and an end. It is in a first instance intended as a writer's aid, with additional hopes to later develop it into an autonomous storyteller. The actual story generation process of UNIVERSE uses plan-like units (plot fragments) to generate plot outlines. Plot fragments provide narrative methods that achieve goals, but the goals considered here are not character goals, but author goals. This is intended to allow the system to lead characters into undertaking actions that they would not have chosen to do as independent agents. The system keeps a precedence graph that records how the various pending author goals and plot fragments relate to each other and to events that have been told already. To plan the next stage

of the plot, a goal with no missing preconditions is selected and expanded.

The line of work initiated by TALESPIN, based on modeling the behaviour of characters, has led to a specific branch of storytellers. Characters are implemented as autonomous intelligent agents that can choose their own actions informed by their internal states (including goals and emotions) and their perception of the environment. Narrative is understood to emerge from the interaction of these characters with one another. This guarantees coherent plots, but, as Dehn pointed out, lack of author goals implies they are not necessarily very interesting ones. However, it has been found very useful in the context of virtual environments, where the introduction of such agents injects a measure of narrative to an interactive setting.

The Virtual Storyteller (Theune et al. 2003) introduces a multi-agent approach to story creation where a specific director agent is introduced to look after a plot. Each agent has its own knowledge base (representing what it knows about the world) and rules to govern its behaviour. In particular, the director agent has basic knowledge about plot structure (that it must have a beginning, a middle, and a happy end) and exercises control over agent's actions in one of three ways: environmental (introduce new characters and object), motivational (giving characters specific goals), and proscriptive (disallowing a character's intended action). The director has no prescriptive control (it cannot force characters to perform specific actions). Theune et al. report non-structural rules are contemplated, to measure issues such as surprise and "impressiveness". The Virtual Storyteller includes a specific narrator agent, in charge of translating the system representation of states and events into natural language sentences. The development effort on the narrator seems to have focused on correct generation of pronouns to make the resulting text appear natural.

The story generator

The objective of this work is to develop a story generator that can generate different stories using the same initial information and that, in addition, can be easily modified to generate a wider range of stories.

With these objectives in mind, we have developed a first prototype that works as a proof of concept to test our approach. This prototype has been developed using very simple, unsophisticated components with the aim of substituting them with more complex ones once the feasibility of the solution has been tested.

The generator is structured in four modules, each of them with their corresponding configuration files: a multi-agent system, which contains an agent for each character and a set of managing agents (currently the world agent, the simulation agent and the director agent), a logger (in charge of collecting the events of the story), a planner (what the characters use to know what to do), and the world (contains the map where the characters interact).

The world

The world is basically a map with different locations, connected by paths between them, in order to make the characters move around it. The prototype we have built has a map formed by three locations:

- Castle: Where the king and the princess are.
- Village: Where the knight starts at.
- Cave: Dragon's home.

Since one of our main goals is to make this storyteller easy to configure, we decided to use text files to load the map and the objects present in each location. The map is structured as an XML file that contains a list of locations with pointers to the locations they are connected to, and the objects and characters situated there, so it works as a graph.

The multi-agent system

The multi-agent system is implemented using the JADE (Bellifemine, Caire, and Greenwood 2007) agent platform. This first prototype generates stories with four types of characters:

- Princess: the character around which the story is built up.
- Dragon: its goal is to kidnap the princess and hold her prisoner in his cave.
- King: the father of the princess. When his daughter is kidnapped, his goal is to find a suitable knight and hire him to kill the dragon. If the knight fails, the king looks for another one, until the princess is safe and sound back in her father's castle.
- Knight: He has no goals until the princess is kidnapped. From then on, his goal is to kill the dragon and take her back to her father.

New stories can be created by simply adding more characters of a type, which are specified at the beginning in a configuration file. In addition, the director agent may create them if it fits the objectives of the story. For example, creating more than one knight, when the princess is kidnapped, the king will look for all the knights available, and will hire the one with lowest fees.

Each character works as a finite state machine consisting of one state per behavior type and a "waiting" state where they are when they don't have active goals.

The world agent is in charge of managing the map, so that all the other agents have a consistent view of the world. Every time a character moves to a new location, he has to send a message to the world agent, so the map gets updated.

The simulation agent is in charge of managing the result of the actions that cannot be directly obtained by the planner, such as the result of the battle between the dragon and the knight.

Finally, the director agent is the one in charge of creating all the necessary agents of the story, these being: the world agent, the simulation agent and the characters. It also makes the necessary decisions to keep the story going, such as setting new goals for the characters. Currently, these decisions are hand written in a configuration file, but the purpose is for this agent to be able to generate them dynamically according to certain heuristics or ask the user to suggest what the new goals should be, in order to make the generator more interactive.

Planning

Each character's actions are driven by their own goals, which are used to plan the sequence of actions they have to carry out to achieve these goals. At the beginning we thought of using just one planner to generate the whole story, but soon it was clear that the planning process would be costly, that the number of possible stories would be small and that it would be difficult to obtain valid plans for agents with conflicting interests. Therefore, we decided it would be more suitable to use separate planners for each agent, so that each of them could make their own plans according to their interests and, in case of conflict, they would have to create new plans to achieve their goals.

We decided to use a STRIPS-based planner (Fikes and Nilsson 1971), since it is quite simple and it is a straightforward option to generate simple stories. In addition, we wanted it to work with PDDL (McDermott 1998) so it would be easy to substitute it with a more sophisticated one in the future.

With this choice, adding a new character to the story involves the creation of another class with the character and two PDDL files, one for its actions, and one for its initial state and goals.

We decided to use the JavaFF planner (Coles et al. 2008) because it works with PDDL and it is open source. The planner takes the domain and the problem in PDDL as inputs, and writes the plan (as a list of actions) into an output file. Since it is open source, we were able to modify it, in order to make the planner return a list of actions (the data structure managed by the planner) instead of writing it to a file. By just adding new actions to the character's PDDL file, new stories are generated, as plans may change including these new actions.

At the time of writing this paper, agents make their plans sequentially (one makes its plan and executes it, then the next one), so that they don't interfere with each other's goals while executing their plans. This reduces the richness of the generated stories, but it is still a good solution to test the validity of the proposed solution.

Capturing the events of the story

As we already said, the only important things are not only the events themselves, so we need a way to gather what happens in the story, but also what is "said" and in what context. Namely, we need a log of everything that happens in the simulation, including the actions that are carried out and the messages exchanged between the agents. We have used the log4j library (Gulcu 2003), which allows the user to enable logging at runtime without modifying the application binary. It also allows us to decide what to enter the log (in our case, it would be everything), the layout, what to save in the log (date, action, agent) and more. Everything is configurable via a parameters file, and will be saved as a log file.

This log is what enables us to actually know what has happened in a certain story, what actions were executed and what was said (scilicet, what messages were interchanged between the agents). However, we must keep in mind that not all the exchanged messages are likely to appear in the final story. For example, all characters have to send a message to the world agent when moving, in order to keep the map updated. These messages should not appear in the story, as their goal is to guarantee internal consistency.

Results

We have implemented a simple prototype where all the described components work together to generate simple, short variations of a story (in Spanish) where a dragon kidnaps a princess and her father the king manages to hire a knight who rescues her and takes her back to her father:

El rey Felipe está preparado.
La princesa Laura despierta.
La princesa sale del castillo.
El dragón Draco emprende el vuelo en busca de alguna princesa desprotegida.
La princesa Laura ha sido secuestrada.
El rey intenta pedir rescate para la princesa Laura.
El caballero Rafael entra en escena.
El rey intenta pedir rescate para la princesa Laura.
El caballero Rafael busca al dragón Draco.
El dragón Draco ha muerto en batalla.
La princesa Laura fue liberada.
El rey entrega 50 monedas al caballero Rafael.
La princesa llega al castillo con el caballero Rafael.
La princesa Laura pone fin a su aventura.

As far as we have been able to test, it is easy to modify the world map to add new locations and situate the characters in them, so they have to make longer journeys to achieve their goals. It is also easy to add new characters of existing kinds so, for example, we can add a second dragon that tries to kidnap the princess from the first one's den.

To make further changes, such as adding new types of characters or actions, it is already necessary to modify the source code of the generator, as well as the domain knowledge, but the code is sufficiently well crafted so that these changes can be easily made. We still have not tested how easy it is to generate a story in a different domain, such as a superheroes story, a western or a love story, but as far as we can see now it may be more painstaking than difficult.

As of now, the stories we generate consist of all the events that take place in the simulation, so our current work is focused on the content extraction, so that we can tell just the relevant events in a relevant order.

To transform the generated logs into text we are using the TAP text generator (Gervás 2011) that receives a crafted set of information and transforms it into an ordered set of sentences that replicates the events that took place in the simulation in the form of a story.

Therefore, in a still simple way, we have developed a story generator that, by means of simple modifications, is able to generate a fair amount of different, although related, stories.

Future Work

Some of the goals we had in mind at the beginning of this project could not be achieved, mostly because of time constraints. We describe some of them here, so they can be used as a starting point for future contributions.

One of the first thing that comes to mind is expanding the world. As the characters and world we are using now are very limited, stories generated are just little paragraphs and there are not many variations between different executions of the application. Just by adding new locations and new characters, we will be adding more possibilities to the story to move along, so that we get more possible stories, which become more intricate at the same time.

As we said before, at the moment, the characters in our application work sequentially, for practical reasons. This reduces the possibilities of the stories generated, since it is more difficult for conflicting interests to appear, or for characters to collaborate to achieve a common goal. A good improvement would be to make all the characters work in parallel, so they would make their plans based on the initial state. While executing their plans, the actions of some characters may interfere in the plans and goals of others. There is when re-planning comes in. Re-planning would make characters interact a lot more, making them compete for the resources to achieve their goals.

In addition, we may want to increase the richness of the stories by making the characters more complex. Adding a slight mood to the characters can make possible stories increase significantly, as the same character may have different behaviors with different moods. Another possibility would be to add feelings and even personality traits.

A lot of richness can also be added via expanding the map. Having a sub-map inside every location would make much more complex plans. Each location can contain different objects, usable and decorative, so the characters can interact with them. For example, you could have a dragon which cannot be killed without a magical sword, so the knight has to find the hidden key to get it.

A much more difficult (and interesting) goal is to make the theme of the story configurable. The idea is to create a configuration file where you can state the theme of the story. That would make everything more difficult, since you can't work with the characters directly. The agents can adopt the role of "actors", instead of characters. With that, there would be a "main character", an "antagonist", a "damsel in distress", and various "secondary actors" in each story. By doing this, you could include in the theme configuration file the names of the characters, their mood (if any), their role, how their actions work (the action "attack" for a knight would make him use his sword, while for a policeman, it would make him use his gun), and have a PDDL file of actions for each role.

Another improvement would be to make the user take the role of a character, so his decisions affect the final result of the story. At first, it could work as in conversational adventures (Montfort 2004), so the user tells the system what actions to carry out. After that, the system would work just as it usually does.

Finally, another option is to give the characters the possibility of making up the details of the story. For example, in our story, the knight could pretend he has a magical weapon to kill the dragon. This endows the stories generated with a whole new level of richness, because new facts are created on the fly.

Acknowledgments*

This paper has been partially supported by the projects WHIM 611560 and PROSECCO 600653 funded by the European Commission, Framework Program 7, the ICT theme, and the Future Emerging Technologies FET program.

References

Bellifemine, F. L.; Caire, G.; and Greenwood, D. 2007. *Developing multi-agent systems with JADE.* Wiley series in agent technology. Wiley.

Coles, A.; Fox, M.; Long, D.; and Smith, A. 2008. Teaching forward-chaining planning with javaff. In *Colloquium on AI Education, Twenty-Third AAAI Conference on Artificial Intelligence.*

Fikes, R. E., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, 608–620. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Gervás, P. 2011. UCM submission to the surface realization challenge. In *Surface Realization Challenge. Challenges 2011 Session at 13th European Workshop on Natural Lan- guage Generation (ENLG 2011).*

Gulcu, C. 2003. *The Complete Log4j Manual*. QOS.ch.

Klein, S.; Aeschliman, J. F.; Balsiger, D.; Converse, S. L.; Court, C.; Foster, M.; Lao, R.; Oakley, J. D.; and Smith, J. 1973. Automatic novel writing: A status report. Technical Report 186, Computer Science Department, The University of Wisconsin, Madison, Wisconsin.

Lebowitz, M. 1985. Story-telling as planning and learning. *Poetics* 14:483–502.

McDermott, D. 1998. PDDL - the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Meehan, J. R. 1977. TALE-SPIN, an interactive program that writes stories. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 91–98.

Montfort, N. 2004. *Twisty Little Passages: An Approach to Interactive Fiction.* Cambridge, MA, USA: MIT Press.

Theune, M.; Faas, E.; Nijholt, A.; and Heylen, D. 2003. The virtual storyteller: Story creation by intelligent agents. In *Proceedings of the Technologies for Interactive Digital Storytelling and Entertainment (TIDSE) Conference*, 204–215
.

Bibliografía

[1] Pablo Gervás (2009), *"Computational approaches to storytelling and creativity"*, AI Magazine 30 (3), 49.

[2] Genesereth & Ketchpel (1994), *"Software Agents",* Stanford University, Magazine Communications of the ACM, vol. 37, 48-ff.

[3] Stuart J. Russell & Peter Norvig (2003), *"Inteligencia Artificial: Un Enfoque Moderno",* segunda edición, editorial Pearson.

[4] Agent Platform Special Interest Group (2000), *"Agent Tecnology Green Paper",* OMG Document agent/00-09-01, Versión 1.0, 1 September 2000, 250 First Ave, Suite 201 Needham, MA 02494.

[5] Foundation for intelligent physical agents (2002), *"FIPA agent management specification"*, SC00023J, FIPA TC Agent Management.

[6] Jörg Hoffmann (2011), *"Everything you always wanted to know about planning (but were afraid to ask)",* Saarland University, Saarbrüken, Germany.

[7] Klein, Sheldon, et al. (1973). *"Automatic novel writing: A status report".* Technical Report 186, Computer Science Department, The University of Wisconsin, Madison.

[8] Meehan, James R. (1977). *"Tale-Spin, an interactive program that writes stories".* Proceedings of the Fifth International Joint Conference on Artificial Intelligence, MIT, Cambridge, MA, August 22–25. Los Altos, CA: Kaufmann, 91–98.

[9] Dehn (1981), *"Story generation after Talespin"*, Proceeding IJCAI'81 Proceedings of the 7th international joint conference on Artificial Intelligence, vol. 1, 16-18.

[10] Lebowitz, Michael (1983). *"Creating a Story-Telling Universe."* A. Nundy (ed). Proceedings of the Eighth International Joint Conference on Artificial Intelligence, August 8–12, Karlsruhe, Germany. Los Altos, CA: Kaufmann, vol. 1, 63–65.

[11] Turner, Scott R. (1993), *"Minstrel: a computer model of creativity and storytelling".* PhD Dissertation, University of California at Los Angeles, Los Angeles.

[12] Pérez y Pérez, Rafael (1999). *"MEXICA: A Computer Model of Creativity in Writing".* PhD Dissertation, The University of Sussex.

[13] Mariët Theune, Sander Faas, Anton Nijholt, Dirk Heilen (2002), *"The Virtual Storyteller",* University of Twente. In ACM SIGGROUP Bulletin, vol. 23, issue 2, ACM Press, 20-21.

[14] Ivo Swartjes y Mariët Theune (2009), *"Late Commitment Virtual Story Characters that can Frame their World",* Technical Report TR-CTIT-09-18, Centre for Telematics and Information Technology, University of Twente, Enschede, the Netherlands.

[15] Riedl, Mark O., & R. Michael Young (2010). *"Narrative Planning: Balancing Plot and Character".* Journal of Artificial Intelligence Research, vol. 39: 217–68.

[16] Si, Marsella & Pynadath, *"Thespian: An architecture for interactive pedagogical drama",* in AIED (2005), and *"Thespian: Modeling socially normative behavior in a decision-theoretic framework"*, Intelligent Virtual Agents (2006), 6th International Conference on Intelligent virtual agents, 369-382.

[17] Montfort, Nick (2007). *"Generating narrative variation in interactive fiction".* PhD Dissertation, University of Pennsylvania, Philadelphia.

[18] Brian Magerko, Chris DeLeon, Peter Dohogne (2011), *"Digital Improvisational Theatre: Party Quirks"*, Intelligent Virtual Agents, 10th International Conference on Intelligent virtual agents, 42-47.

[19] Jörg Hoffmann (2001), *"The Fast-Forward planning system",* Institute for Computer Science, Albert Ludwigs University, in AI Magazine, vol. 22, number 3, 57-62.

[20] Bonet, Loerincs & Geffner (2000), *"HSP: Heurístic Search Planner"*, Entry at AIPS-98 Planning Competition, AI Magazine, vol. 21 (2).

[21] Bylander, T. 1994, *"The computational complexity of propositional STRIPS planning".* Artificial Intelligence 69 (1 - 2):165-204

[22] Blum, A. L., and Furst, M. L. 1997. *"Fast planning through planning graph analysis"*. Artificial Intelligence 90(1 - 2):270-298