TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Generación Automática de Dominios

Generación de Dominios de Planificación Jerárquica a partir de Descripciones de Videojuegos en VGDL

Autor
Ignacio Vellido Expósito

Director
Juan Fernández Olivares

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Granada, Junio de 2020

Generación de Dominios de Planificación Jerárquica a partir de Descripciones de Videojuegos en VGDL

Ignacio Vellido Expósito

Palabras clave: *Inteligencia Artificial*, *Ingeniería del Conocimiento*, *Planificación Jerárquica*, *HTN*

Resumen

La generación de dominios en planificación jerárquica, en concreto planificación HTN (Hierarchical Task Network), es una tarea meticulosa requerida siempre que queramos utilizar un planificador jerárquico para resolver una gran cantidad de problemas en un dominio. Cuanto más complejo el dominio a representar, más esfuerzo debe ponerse en la escritura de este, lo cuál dificulta la adopción de técnicas de planificación para casos complejos como, por ejemplo, los videojuegos. En este trabajo nos centramos en simplificar el proceso de generación de dominios HTN para resolver problemas de videojuegos. A partir de la descripción de los objetos que intervienen en un videojuego, y de su dinámica, descrito en el estándar VGDL (Video Game Description Language), proponemos un proceso automático de Ingeniería del Conocimiento para generar Dominios de Planificación, de forma que simplifiquemos el proceso de representación del conocimiento de planificación en videojuegos.

Automatic Generation of Hierarchical Planning Domains from VGDL Videogame Descriptions

Ignacio Vellido Expósito

Keywords: *Artificial Intelligence*, *Knowledge Engineering*, *Hierarchical Planning*, *HTN*

Abstract

The automatic generation of hierarchical planning domains, in particular HTN (Hierarchical Task Network) planning domains, is a meticulous task required every time we try to use a hierarchical planner for resolving a huge variety of problems. The more complex the domains we try to represent, more effort must be put in the creation of these domains. This makes the representation of complex domains, like videogames, a very difficult task to accomplish. Therefore, in this work we focus on the simplification of the HTN-domains generation for videogames problems. Knowing the dynamics of the game and the objects involved in it, described in VGDL (Video Game Description Language), we propose a Knowledge Based System with an automatic compiling process from VGDL videogames descriptions into HTN-domains.

Yo, **Ignacio Vellido Expósito**, alumno de la titulación de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 79056166Z, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Ignacio Vellido Expósito

Granada, a 24 de junio de 2020

D. **Juan Fernández Olivares**, profesor del Departamento de Ciencias de la Computación e Inteligencia Artificial.

Informa:

Que el presente trabajo, titulado ***Generación automática de dominios***, ha sido realizado bajo su supervisión por **Ignacio Vellido Expósito**, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 24 de junio de 2020.

El tutor:

Juan Fernández Olivares

Índice general

Índice de figuras

Capítulo 1

Introducción

La planificación automática es una rama de la Inteligencia Artificial centrada en la búsqueda de planes que resuelven un determinado problema [Ghallab et al., 2016]. Para ello hace uso de un programa, llamado *planificador*, el cual recibe una descripción del mundo *(dominio)* y una especificación del caso concreto a resolver *(problema)*. A partir de esta entrada, el planificador es capaz de encontrar (si es posible) la secuencia de acciones capaces de resolver el problema, siguiendo las reglas establecidas en el dominio.

Estos elementos acarrean un gran problema: la creación de un dominio que represente con exactitud los requisitos es una tarea larga, difícil y que debe ser tratada cuidadosamente. Además, optimizar un dominio (de manera que se represente de la forma más simple posible) raramente resulta sencillo, siendo en la mayoría de casos, por las características del problema, directamente imposible de realizar.

Este proyecto se centra en la aplicación de técnicas de planificación a la resolución de problemas de actuación en videojuegos, con el objetivo de reducir tanto el esfuerzo que un ingeniero de conocimiento debe emplear, como las dificultades que debe superar para elaborar un dominio de planificación de calidad para un agente en un videojuego. La idea fundamental es, por tanto, utilizar de partida un lenguaje próximo a la descripción de videojuegos y definir un proceso de traducción de este lenguaje a un lenguaje de planificación.

De esta manera, siguiendo un lenguaje simple, sencillo de entender, y al mismo tiempo expresivo, como es el caso de VGDL (**V**ideo **G**ame **D**escription **L**anguage) [Schaul, 2013], dejamos de preocuparnos sobre predicados, acciones, etc. y nos concentramos únicamente en los objetos presentes en nuestro problema, junto a las relaciones que existen entre ellos.

El lenguaje de planificación en el que nos centraremos seguirá el paradigma de planificación jerárquica, o HTN (del inglés, **H**ierarchichal **T**ask **N**etworks), en el que la representación de dominios de planificación sigue un esquema jerárquico donde existen acciones compuestas, descritas a un alto nivel de abstracción, y acciones primitivas, del menor nivel de abstracción. Este es un paradigma de planificación usado con grán éxito en la resolución de problemas reales, y que sigue un esquema conceptual que se asemeja en gran medida al de un videojuego.

Para conseguir el objetivo del proyecto se ha definido un proceso de Ingeniería del Conocimiento donde, recibiendo la descripción de un juego y de un nivel en VGDL, se genera automáticamente un dominio y un problema HTN representando las dinámicas del juego. Estos archivos se proporcionan como entrada a un planificador HTN, el cuál genera planes para un agente en el juego. Con este dominio base, un diseñador humano podría especificar una estrategia específica para el agente de forma que este actúe persiguiendo los objetivos deseados.

Para mejorar la verificación de la calidad del proceso, se integra un proceso de replanificación con el framework GVGAI, lo que nos permite obtener una representación visual de los resultados. Esto nos proporciona adicionalmente un ciclo constante de readaptación del agente, facilitando la persecución de objetivos a alcanzar en el juego.

Para validar el proceso propuesto se ha llevado una experimentación con 4 casos de estudio, produciendo dominios para dos juegos de puzles y dos juegos reactivos. Unidos al módulo de replanificación, en algunos casos se ha mejorado la estrategia predeterminada del agente para visualizar mediante GVGAI el potencial que tiene en la resolución de juegos.

Con este proyecto conseguimos tres grandes avances en el campo de la planificación automática: (1) demostramos que mediante el uso de una base de conocimiento y un lenguaje sencillo, podemos crear un proceso de traducción que nos permita definir automáticamente dominios de gran complejidad; (2) la estructura base de los dominios producidos permite incluir una estrategia para el agente de forma que este sea capaz de resolver juegos complejos; (3) el módulo de replanificación no solo ayuda en la actuación del agente, sino que permite a un diseñador humano (como un estudiante) depurar y comprender con facilidad el dominio implementado.

En las siguientes secciones introducimos conceptos necesarios para lo comprensión de esta memoria junto a un análisis del estado del arte. Seguidamente, se muestra el proceso de desarrollo que ha seguido el proyecto. En el capítulo 4 se incluye una descripción global del funcionamiento del software y las partes que lo forman. En el capítulo 5 se detalla la implementación realizada para la construcción de cada una de las partes. Por último, en los dos últimos capítulos

se muestra la experimentación realizada sobre cada caso de estudio, y se realiza un análisis de los logros del proyecto, las contribuciones que aporta y los posibles trabajos futuros que derivan de este.

Capítulo 2

Preliminares

2.1. Trabajos relacionados

El problema de la generación de dominios de planificación a partir de descripciones en otros lenguajes se ha abordado previamente en el campo del eLearning [Castillo et al., 2010], de la gestión de procesos de negocio (BPM) [González-Ferrer et al., 2013], y de la representación de guías de práctica clínica [Fdez-Olivares et al., 2011]. En todos los casos se parte de un lenguaje próximo al dominio de aplicación (BPMN en un caso, Asbru en otro) y se generan dominios de planificación que se emplean para ayudar a expertos en la toma de decisiones.

Consideramos que estos trabajos han resuelto con éxito los problemas a los que se enfocan, pero se alejan de la propuesta presentada en este proyecto. En nuestro caso, pretendemos no concentrarnos en una única tarea y generalizar a un mayor rango de casos, como múltiples videojuegos de distinto tipo.

Nosotros usamos VGDL para describir los objetos de un videojuego, sus relaciones e interacciones, y generamos un dominio de planificación que describe las acciones que puede realizar un agente junto a la dinámica de dicho juego. Además, a partir de la descripción del nivel de juego, se genera un problema de planificación. El dominio generado se utiliza para que un usuario experto defina la estrategia que puede seguir el agente para superar un determinado nivel de juego.

2.2. Antecedentes

2.2.1. Planificación automática

La planificación automática es un área de la IA dedicada al estudio de técnicas encargadas de resolver problemas de actuación de agentes, tanto físicos (ej: robots), como virtuales (ej: jugadores en videojuegos) [Ghallab et al., 2016].

Existen dos elementos necesarios para la resolución de problemas de planificación, estos son:

- **Problema de planificación**: En él se representa (1) un estado preliminar que describe la situación inicial en la que se encuentra el agente, así como los objetos del entorno; y (2), un objetivo que indica la situación que debe alcanzar el agente.
- **Dominio de planificación**: En él se representa un modelo de acciones que el agente puede realizar, al igual que la dinámica del entorno, es decir, cómo las acciones del agente producen cambios que afectan al entorno.

Ambos elementos se proporcionan como entrada a un programa, llamado **planificador**, que lleva a cabo un proceso de búsqueda para generar un plan de actuación. Este plan indica la secuencia de acciones que, partiendo de la situación inicial, alcanza la situación objetivo.

El esquema de representación del conocimiento usado para describir tanto dominios como problemas de planificación es la **lógica de predicados**. A lo largo de los años se ha definido un lenguaje estándar, denominado PDDL [Fox and Long, 2003], que permite la representación de dominios y problemas de planificación.

En PDDL los dominios se estructuran en las siguientes partes:

- **Tipos** (objects): Clases de objetos presentes en el mundo, representadas jerárquicamente.
- **Predicados** (predicates): Relaciones o características de los objetos.
- **Funciones** (functions): Predicados que almacenan un valor numérico.
- **Primitivas** (actions): Operadores que cambian el mundo. Cuentan con tres partes: **parámetros** implicados; una sección de **precondiciones** (predicados que deben ser ciertos cuando se produzca la acción, o funciones que deben tener un valor concreto especificado en la precondición); y otra de **efectos** (cambios que se producen en el estado del mundo).

En la figura 2.1 podemos ver un ejemplo de las diferentes partes definidas para un dominio concreto. Se puede apreciar cómo la definición de tipos sigue el formato **[lista de tipos] - padre**; cómo tanto a los predicados como a las funciones se le asocian unos tipos concretos, usando el esquema
?nombre - tipo; y cómo se define una acción junto a la utilización de sus parámetros.

```
(:types
    ...
    jugador - jugador
    oro diamantes - monedas
)

(:functions
    ...
    (cantidad_oro ?j - jugador)
)

(:predicates
    ...
    (misma_posicion ?j - jugador ?o - oro)
)

(:action coger-oro
    :parameters (?j - jugador ?o - oro)
    :precondition (and
                      (misma_posicion ?j ?o)
                  )
    :effect (and
                (increase (cantidad_oro ?j) 1)
            )
)
```

Figura 2.1: Ejemplo de las diferentes partes de un dominio PDDL.

```
(:objects
    ...
    jugador1 - jugador
    oro1 oro2 oro3 - oro
)

(:init
    ...
    (misma_posicion jugador1 oro2)
)

(:goal (and
            (= (cantidad_oro jugador1) 3)
            ...
        )
)
```

Figura 2.2: Ejemplo de las diferentes partes de un problema PDDL.

Por otro lado, un problema PDDL cuenta con tres partes:

- **Objetos** (objects): Lista de instancias presentes en el problema.
- **Estado inicial** (init): Hechos que son ciertos al inicio del problema (compuestos por predicados y asignaciones de valores a las funciones).
- **Objetivo** (goal): Condiciones que deben ser ciertas al final del plan.

En la figura 2.2 tenemos un ejemplo de problema para el dominio de la figura 2.1, instanciando cuatro objetos (uno de tipo *jugador* y tres de tipo *oro*). Se puede apreciar cómo se afirman los predicados en la sección **:init** y cómo se establece un objetivo en la sección **:goal**.

A día de hoy la planificación es un campo muy amplio que se divide en múltiples ramas, cada una expandiendo en expresividad y centrándose en tipos de problemas más concretos. Algunos ejemplos de estas ramas son: planificación temporal; probabilística; jerárquica. En este TFG nos hemos centrado en el paradigma de planificación jerárquica, descrito a continuación.

2.2.1.1. Planificación jerárquica (HTN)

La planificación jerárquica, también conocida como HTN (**H**ierarchichal **T**ask **N**etworks), es una rama de la planificación que permite representar acciones descomponibles, es decir, acciones de alto nivel que tienen una o varias alternativas de ser realizadas.

Este tipo de estructuras pueden ser representadas en la planificación clásica, como en el lenguaje PDDL, pero la cantidad de acciones y predicados adicionales que se deben introducir complica en gran manera la representación.

```
(:task <T>
    :parameters (...)

    (:method <T_1>
        :precondition (...)
        :tasks (
            (subtask_1 ...)
            ...
            (subtask_1 ...)
        )
    )
    ...
    (:method <T_N>
        ...
    )
)
```

Figura 2.3: Representación de una tarea en HPDL. Cada tarea recibe una lista de parámetros y contiene uno o más métodos. De forma similar a las clásicas primitivas de PDDL, cada método tiene una precondición (como expresión lógica de predicados) y un número arbitrario de subtareas abstractas.

HPDL y Siadex

HPDL (**H**ierarchical **P**lanning **D**escription **L**anguage) es una variante HTN de PDDL desarrollada por [Fdez-Olivares et al., 2006] para el planificador Siadex. El lenguaje HPDL se centra en simplificar la escritura de la jerarquía de acciones introduciendo el concepto de **tarea**, una representación de actividad descomponible.

En la figura 2.3 se muestra el formato de una tarea. Cada tarea puede ser descompuesta en varios métodos permitiendo una división clara en subtareas, dónde cada método representa un modo alternativo de completar la tarea principal. Además, cada subtarea puede ser tanto una primitiva como otra tarea compuesta.

El resto de partes del dominio en HPDL (primitivas, tipos, predicados, etc.) siguen la misma estructuración que en PDDL.

En HPDL un problema se representa de la misma manera que en PDDL, a excepción del **goal**. Un goal en HPDL no es una serie de hechos que deben ser ciertos al terminar el plan, sino una serie de tareas a completar. Por tener un comportamiento diferente, es llamado **task-goal** en este lenguaje.

2.2.1.2. Ingeniería del conocimiento en planificación

La definición de un dominio en problemas complejos es una tarea complicada para el diseñador. Este debe contar con el conocimiento suficientemente para poder definir todos los requisitos de manera precisa y sin errores. Las técnicas de Ingeniería del Conocimiento dentro de este campo pretenden facilitar y ayudar al diseñador en el proceso para poder construir dominios de mayor calidad.

Formalmente la Ingeniería del Conocimiento en planificación (**K**nowledge **E**ngineering in **P**lanning and **S**cheduling, KEPS) se encarga de la adquisición, formulación y validación de conocimiento para la definición de modelos de dominio en planificación.

Un **modelo de dominio en planificación** es una abstracción del contenido invariante de un dominio en los problemas sobre los que se puede emplear, generalmente correspondiendo a las diferentes partes que lo forman, como predicados, objetos, acciones, etc.

Tal y como explican [McCluskey et al., 2016], este modelo de dominio va más allá del uso clásico en otras ramas del desarrollo del software, donde este tipo de modelos se suelen representar mediante diagramas con el objetivo de descubrir requisitos del software a producir. Los modelos de dominio en KEPS se suelen definir en un lenguaje formal independiente del usado para los dominios.

Gracias a estos modelos de dominio, podemos utilizar las técnicas de Ingeniería del Conocimiento para la producción de grandes cantidades de dominios con características comunes entre sí.

Existen aproximaciones anteriores que han centrado su esfuerzo en aplicar este paradigma para la resolución de videojuegos, como es el caso de [Couto Carrasco, 2015]. En él los autores representan manualmente tres juegos de puzzles usando el lenguaje PDDL.

Esto nos mostró la posibilidad de simplificar la creación de este tipo de dominios usando el lenguaje subyacente que los define. El poder partir de un lenguaje simple que nos ayude a definir dominios ampliaría el rango de problemas a los que la planificación se podría enfrentar, a la vez que se reduce la dificultad y el tiempo de definición de estos problemas.

2.2.2. GVG-AI

GVGAI (**G**eneral **V**ideo **G**ame **AI**) es un entorno para la creación y el testeo de agentes en múltiples entornos [Perez-Liebana et al., 2016]. Cuenta en su interior con más de 100 juegos de distinto tipo, formados por descripciones de juego y de los niveles en el lenguaje VGDL. Aunque este framework está orientado a abordar problemas de Inteligencia Artificial General, creemos que puede ser un buen entorno de pruebas para técnicas de planificación, concretamente integrando

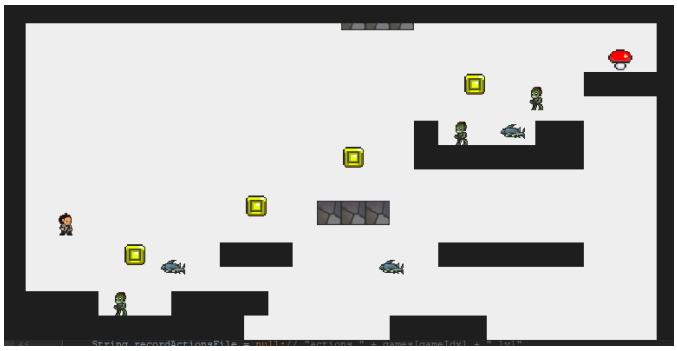


Figura 2.4: Imagen de Mario Bros en GVGAI.

planificación con actuación.

El lenguaje subyacente que utiliza GVGAI es una variante en Java de VGDL, la cual contiene la mayoría de tipos definidos en la versión VGDL original, y expande algunas características adicionales para adaptar otros tipos de juegos.

2.2.3. VGDL

VGDL (**V**ideo **G**ame **D**escription **L**anguage) [Schaul, 2013] es un lenguaje de alto nivel para la descripción de juegos, que se especializa en velocidad y simpleza.

Existen tres variantes de este lenguaje: la original, implementada en Python [Schaul,]; su reimplementación [Vereecken, , VGDL 2.0], que actualmente sigue con soporte; y [GVGAI, , java-vgdl], la versión utilizada por GVGAI en su framework.

Este proyecto se centra, aunque no es exclusivamente, en java-vgdl. De esta manera existe la posibilidad de probar los dominios generados (y sus respectivos planes) mediante el entorno GVGAI y la definición de un agente.

Se ha escogido VGDL como lenguaje base por su facilidad de compresión sin comprometer una alta capacidad expresiva. Existe una alta gama de tipos predefinidos en VGDL, yendo desde tipos estáticos hasta varias clases de NPCs y agentes. Esto le permite representar una gran variedad de juegos, incluyendo no solo aquellos con mundos cuadriculados sino también con físicas 2D. Este lenguaje también cuenta con una gran potencialidad, pues en apenas 52 líneas puede definir juegos suficientemente complejos como el Mario Bros.

Un juego en VGDL se especifica con un archivo de descripción de juego, detallando las dinámicas; y un archivo por cada nivel indicando las posiciones iniciales de los objetos en cada uno de ellos.

```
SpriteSet
   user    > VerticalAvatar
   boulder > Missile        orientation=DOWN
```

(a) Definición de objetos y sus parámetros

```
LevelMapping
   u > user
   b > boulder
```

(b) Caracteres que representan cada objeto en el archivo de definición de nivel

```
InteractionSet
   user boulder > killIfFromAbove
```

(c) Ejemplo de interacción entre objetos

```
TerminationSet
   SpriteCounter stype=user limit=0 win=False
```

(d) Criterio de parada para el juego

Figura 2.5: Las cuatro partes en las que se compone una descripción de juego en VGDL.

Las dinámicas en VGDL se estructuran en cuatro partes, pudiendo definirse en cualquier orden:

- **SpriteSet**: Especificación de los objetos que participan en el juego (avatares, enemigos, misiles, etc.). Sigue el formato:

  ```
  nombre > tipo [lista_de_parámetros]
  ```

  En la figura 2.5a se muestran ejemplos de objetos y sus atributos, donde se puede ver que *boulder* es un misil que solo se mueve hacia abajo; y que *user* es un *VerticalAvatar*, es decir, un agente con dos movimientos posibles (*up* y *down*). En esta parte se permite la declaración de objetos de forma jerárquica, de forma que los hijos puedan heredar atributos y comportamientos de los padres. Esta herencia no es múltiple, un hijo solo puede derivar de un único padre.

  Los niveles de la jerarquía se definen en base a la indentación, yendo de padres a hijos.

- **LevelMapping**: Aquí se indican los caracteres que representan a los objetos en un archivo de descripción de nivel, donde cada línea de esta parte sigue el formato:

  ```
  carácter > lista_de_tipos
  ```

  La figura 2.5b muestra como **boulder** y **user** se simbolizan mediante las letras **b** y **u** respectivamente. Estos serán usados en la descripción de un nivel de juego (detallado más adelante).

- **InteractionSet**: En esta sección se indican las interacciones entre objetos, producidas mediante la colisión de ellos, siguiendo el formato:

  ```
  sprite1 sprite2 > tipo [lista_de_parámetros]
  ```

  Como ejemplo, en la figura 2.5c vemos una interacción que involucra a un objeto *boulder* y a un *user*, llamada *killIfFromAbove*. Es importante tener

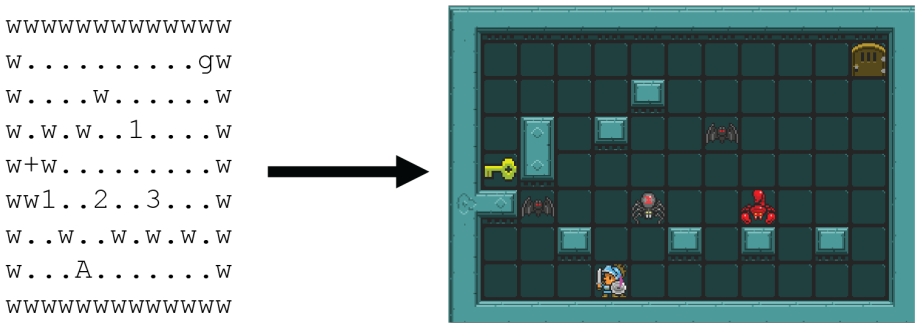


Figura 2.6: Representación de un nivel en GVGAI. Se puede ver cómo cada carácter representa una instancia en el juego. Por ejemplo: *w* significa *wall* (muro), *A* significa *avatar* y cada número un enemigo diferente.

en cuenta que el orden en el que se definen los objetos es relevante, siendo el primero el que produce la interacción y el segundo el principal receptor de sus efectos, aunque en algunos casos más de un objeto se puede ver afectado por la interacción.

La definición de múltiples interacciones se puede resumir siguiendo el formato de GVGAI-VGDL, en el que se indican todos los objetos que producen una misma interacción para uno dado.

- **TerminationSet**: Se constituye por una serie de criterios que indican las condiciones de parada del juego. Cada uno de ellos sigue el formato:

```
condición [lista_de_parámetros] win=<True/False>
```

Podemos ver en la figura 2.5d un ejemplo, donde se especifica que cuando el número de objetos *user* actualmente vivos llega a cero, el juego termina y el agente pierde la partida.

Un archivo de **descripción de nivel** se compone de una matriz 2D de caracteres en cuyas celdas se define la posición inicial de una instancia de un objeto. Cada carácter se asocia con su objeto siguiendo las reglas definidas en el *LevelMapping*. La figura 2.6 muestra una transformación desde una descripción de un nivel a la visualización que produce GVGAI.

Podemos apreciar que las descripciones de juegos y sus niveles siguen una conceptualización similar a la de un dominio HTN y sus problemas asociados. Es decir, una descripción de juego en VGDL indica objetos, relaciones e interacciones que pueden representarse mediante un dominio de planificación HPDL. Por otro lado, un archivo de descripción de nivel contiene información que puede representarse como el estado inicial de en problema de planificación en HPDL.

Adicionalmente, la ejecución basada en turnos característica de los juegos en mundos cuadriculados se asemeja a una tarea HTN recursiva con criterios de parada.

2.2.4. ANTLR

ANTLR (**AN**other **T**ool for **L**anguage **R**ecognition) [Parr and Quong, 1995] es un generador de intérpretes que, siguiendo un lenguaje especial de definición de gramáticas, permite procesar y traducir textos o archivos binarios.

En este TFG hemos elegido ANTLR para parsear la información de los ficheros VGDL (de juego y de nivel) debido a que resulta más sencillo de aprender que otros lenguajes similares. Su uso permite obtener un programa (llamado **listener**) capaz de leer un archivo de descripción de juego en VGDL y extraer a partir de él las datos relevantes en el proceso de parseo.

ANTLR tiene muchas características en común con el lenguaje usado en LEX. Principalmente se centra en la asociación de expresiones regulares a patrones que almacenan la información encontrada. Además, permiten la inclusión de código a ejecutar cuando se detecta cada una de las reglas.

Existen tres términos de relevancia a mencionar:

- **Lexer rules**: Reglas para detectar los tokens (unidad de información básica) de la gramática.
- **Parser rules**: Reglas para el reconocimiento de expresiones. Adicionalmente, se pueden asignar etiquetas a las diferentes opciones de las parser rules para poder referenciarlas en el listener producido. De forma similar, también se permite asignarle nombres a conjuntos de tokens dentro de una regla.
- **Fragments**: Elementos reutilizables que ayudan al reconocimiento de tokens en las lexer rules, de cara a simplificar la definición de la gramática.

ANTLR puede devolver el intérprete en múltiples lenguajes, y en nuestro caso hemos especificado la salida a Python. Para este proyecto se ha utilizado la última versión de ANTLR en el momento de su realización, la 4.7.2.

Capítulo 3

Plan de Trabajo

3.1. Metodología

El proyecto sigue un esquema clásico del ciclo de vida de un proceso de ingeniería del conocimiento, durante el cuál se ha seguido una metodología basada en Scrum. Se ha mantenido una serie de reuniones semanales en las que se han ido especificando objetivos realizables a corto plazo, revisando y debatiendo continuamente lo realizado en las semanas anteriores.

Esta metodología de trabajo nos permitió tener un control cercano de la evolución del proyecto sin necesidad de haber especificado con precisión todos los requisitos a comienzos de este.

Recordando que el objetivo principal era la generación automática de dominios, y más concretamente, la traducción de descripciones VGDL en estos dominios, se destacaron los siguientes subobjetivos principales:

OBJ-1 Estudio y análisis del lenguaje VGDL.

OBJ-2 Estudio y análisis de los lenguajes PDDL y HPDL.

OBJ-3 Estudio y creación de una gramática en ANTLR para el lenguaje VGDL.

OBJ-4 Utilizando la gramática anterior, creación de un traductor VGDL en dominios HPDL.

OBJ-5 De igual manera, creación de un traductor VGDL en problemas HPDL.

OBJ-6 Evaluación inicial.

OBJ-7 Integración con replanificación.

OBJ-8 Experimentación. De forma que se valide el conocimiento integrado y el proceso de replanificación.

Para cumplimentar los objetivos anteriores se siguió el siguiente plan de trabajo:

1. Lectura de artículos oficiales, documentación y libros.
2. Estudio y análisis de los lenguajes PDDL, HPDL y VGDL.
3. Estudio y creación de una gramática en ANTLR para el lenguaje VGDL.
4. Creación de una base de conocimiento de VGDL y HPDL.
5. Modificación del parser automáticamente generado por ANTLR para adaptarlo a la creación de un dominio HPDL.
6. Ampliación del traductor para poder generar problemas HPDL.
7. Experimentación inicial de la generación de distintos dominios y sus respectivos problemas.
8. Integración con replanificación.
9. Experimentación y evaluación final, en diferentes juegos y niveles.

3.2. Temporización

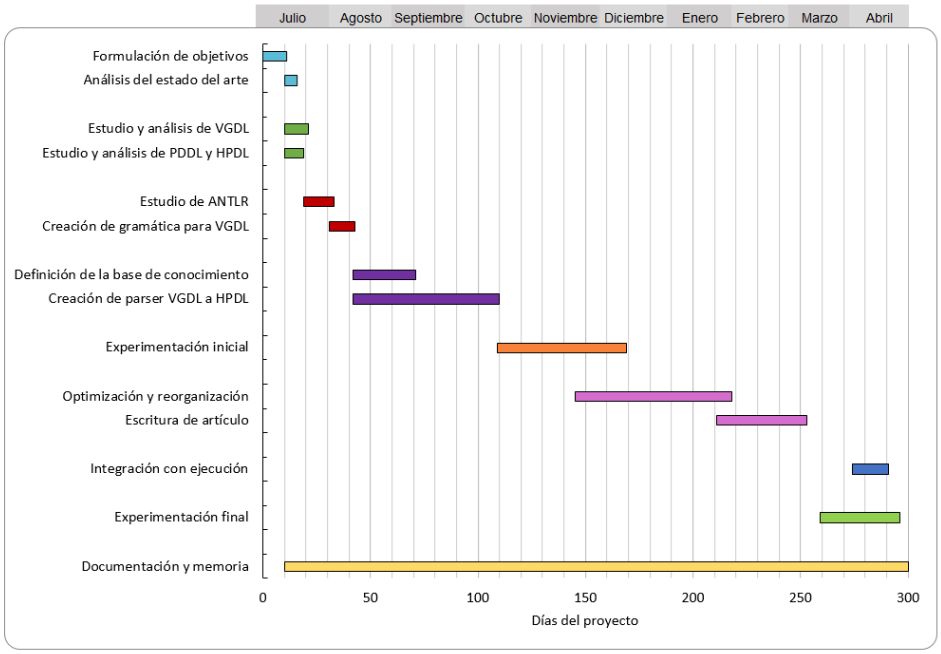


Figura 3.1: Diagrama de Gantt del proyecto.

TAREA	FECHA DE INICIO	FECHA DE TERMINACIÓN	DÍA DE COMIENZO	DÍAS DE DURACIÓN
Primera fase				
Formulación de objetivos	5/7	15/7	0	11
Análisis del estado del arte	15/7	20/7	10	6
Segunda fase				
Estudio y análisis de VGDL	15/7	25/7	10	11
Estudio y análisis de PDDL y HPDL	15/7	23/7	10	9
Tercera fase				
Estudio de ANTLR	24/7	6/8	19	14
Creación de gramática para VGDL	5/8	16/8	31	12
Cuarta fase				
Definición de la base de conocimiento	16/8	13/9	42	29
Creación de parser VGDL a HPDL	16/8	22/10	42	68
Quinta fase				
Experimentación inicial	22/10	20/12	109	60
Sexta fase				
Optimización y reorganización	27/11	7/2	145	73
Escritura de artículo	1/2	13/3	211	42
Séptima fase				
Integración con ejecución	4/4	20/4	274	17
Octava fase				
Experimentación final	20/3	25/4	259	37
Documentación				
Documentación y memoria	15/7	29/4	10	290

Figura 3.2: Tareas y fechas del proyecto.

El plan de trabajo siguió la siguiente temporización, resumida en los diagramas 3.1 y 3.2. La duración del proyecto alcanzó 10 meses de desarrollo, aunque no a jornada completa. Parte de él se realizó como contratado técnico en el proyecto de investigación [TIN2015-71618-R,].

Esta temporización se dividió en las siguientes fases:

3.2.1. Primera fase

Durante las primeras semanas se hizo un estudio del problema y del estado del arte. Posteriormente, se plantearon los objetivos a grandes rasgos del problema, y se discutieron las posibles maneras de solucionarlo. Temas tales como la viabilidad de VGDL como lenguaje a traducir, el entorno GVGAI como software para la experimentación y el lenguaje HPDL para definir los dominios... también fueron planteados aquí.

Hubo un tema de discusión sobre este último, la viabilidad de PDDL frente a HPDL, al ser PDDL el lenguaje más estandarizado dentro de la planificación. La idea resultante fue que aunque PDDL tuviera un mayor soporte y una ma-

yor gama de planificadores sobre los que experimentar, la ausencia de jerarquía complicaría en gran medida los dominios devueltos. Es posible implementar una "falsa" jerarquía de tareas con PDDL, pero la cantidad de predicados y acciones adicionales que habría que incluir, además de la complejidad que añade a la comprensión del dominio, lo volvería poco práctico.

3.2.2. Segunda fase

La segunda fase consistió en estudiar el lenguaje VGDL en sus tres versiones, ya que en caso de ser necesario sería conveniente introducir conceptos de cualquiera de ellas. Por otro lado, puesto que HPDL es una extensión de PDDL, el estudio de los lenguajes de planificación comenzó por este último.

Aunque los lenguajes de planificación se habían impartido en la materia *Técnicas de los Sistemas Inteligentes* del grado de Ingeniería Informática, los conocimientos que se requerían para el proyecto debían ser más profundos que los adquiridos en la asignatura.

El estudio y comprensión de VGDL resultó complicado en su momento. La ausencia de bibliografía descriptiva y la poca documentación disponible implicó tener que indagar en el propio código del parser original [Schaul,]. Se aprovechó esta etapa para recopilar toda la información que se iba encontrando sobre el lenguaje de cara a facilitar la implementación de la base de conocimiento.

3.2.3. Tercera fase

Una vez adquiridos los conocimientos que a primera vista parecían necesarios para el proyecto, comenzó la fase de realización.

El primer paso fue la creación de una gramática de cara a facilitar el proceso de traducción. Para esto se plantearon herramientas clásicas como LEX y YACC, pero se decidió apostar por una que actualmente está cobrando importancia, ANTLR.

ANTLR permite usarse en distintos lenguajes, y aquí se planteó sobre cuál de ellos generar el parser. Se optó por Python por ser un lenguaje interpretado, con gran soporte y puesto que ya se tenían conocimientos sobre él.

Al final de esta etapa ya se contaba con una gramática definida para VGDL, que con las herramientas de ANTLR permitían obtener la base del parser que más adelante se iba a desarrollar.

3.2.4. Cuarta fase

Una vez obtenida la gramática, comenzaron dos tareas simultáneamente. Por un lado se implementaba la parte genérica del parser (lectura de archivos, escritura de resultados y la integración con ANTLR), mientras que por otro se terminaba de completar el listener devuelto por ANTLR.

Terminada la parte genérica del parser, se definió la base de conocimiento y se implementó la integración con el resto del software, obteniendo así un proceso para generar dominios de principio a fin.

Este proceso fue de los más llevaderos durante el proyecto, y corresponde a la mayor parte del tiempo puesto en la implementación del software.

3.2.5. Quinta fase

Una vez acabado el parser, comenzó una fase inicial de experimentación, más tarde expandida con la creación a mano de una estrategia para el agente. El objetivo era verificar y explorar las capacidades del software para resolver planes de manera eficiente.

Se vió que definiendo una nueva gramática en ANTLR el software podría traducir nuevos lenguajes a dominios, y que alterando la base de conocimiento (pero manteniendo su estructura) se podría traducir en distintos lenguajes de planificación.

Al mismo tiempo seguía la experimentación con los dominios generados para HPDL con nuevos juegos.

3.2.6. Sexta fase

Una vez terminada esta prueba, se optó por realizar una reorganización del código y la base de conocimiento de cara a facilitar la extensión del parser para nuevos lenguajes, tanto de entrada como de salida.

Durante esta fase se propuso la idea de presentar el proyecto a diferentes congresos, focalizándose primeramente en el workshop KEPS de la conferencia ICAPS 2020. Por tanto, se decidió redactar un artículo que mostrara las características del trabajo, y durante un tiempo se centró el esfuerzo en la escritura de este.

Debido a la crisis del COVID-19 y al retraso en las fechas de las conferencias, a día de terminación de la redacción de esta memoria aún no se ha enviado el artículo escrito.

3.2.7. Séptima fase

Una vez visto el correcto funcionamiento de los dominios generados, se realizó el proceso de replanificación y su integración con el framework GVGAI. De esta manera se podría comprobar visualmente los planes devueltos por el planificador y aumentar la funcionalidad con la replanificación en tiempo real.

3.2.8. Octava fase

Para terminar, se volvió a experimentar para revisar las capacidades del software, con la idea de tener unas ideas definitivas de su funcionamiento. Al estar incluída la replanificación, ya era posible tener una visualización del comportamiento del agente en estos dominios.

Capítulo 4

Metodología

En este proyecto hemos generado dominios y problemas HTN a partir de descripciones de videojuegos y de sus niveles de juego asociados en el lenguaje VGDL. Los dominios describen en términos de tipos, predicados, tareas, etc. los objetos y las interacciones en el juego de manera precisa, permitiendo a un diseñador humano definir la estrategia del agente de acorde a los objetivos deseados. Por otra parte, los problemas se han generado directamente a partir de los ficheros de nivel.

El proceso basado en el conocimiento implementado recibe como entrada dos archivos: una descripción del juego en VGDL y un nivel concreto de este. Adicionalmente, hace uso de una Base de Conocimiento definida previamente con la metodología necesaria para transformar la información extraída en contenido HPDL. Como resultado, el proceso produce un dominio y un problema HPDL.

El procedimiento, representado en la figura 4.1, sigue los siguientes pasos:

1. Mediante la gramática definida en ANTLR se parsea el archivo de descripción de juego y se extraen una serie de entidades del juego (*game entities*), representando los conceptos de interés para un dominio HTN definidos en la descripción de juego en VGDL.

2. Estas entidades entran en un módulo responsable de producir el dominio de salida, dónde se almacena parte de la base de conocimiento. Este módulo instancia las entidades y según el tipo de los objetos y de las interacciones produce diferente contenido para el dominio HPDL (tipos, predicados, tareas...)

3. De manera similar, las entidades también pasan por un módulo encargado de producir el problema HPDL (objects, init y goal), conteniendo otra parte de la base de conocimiento.

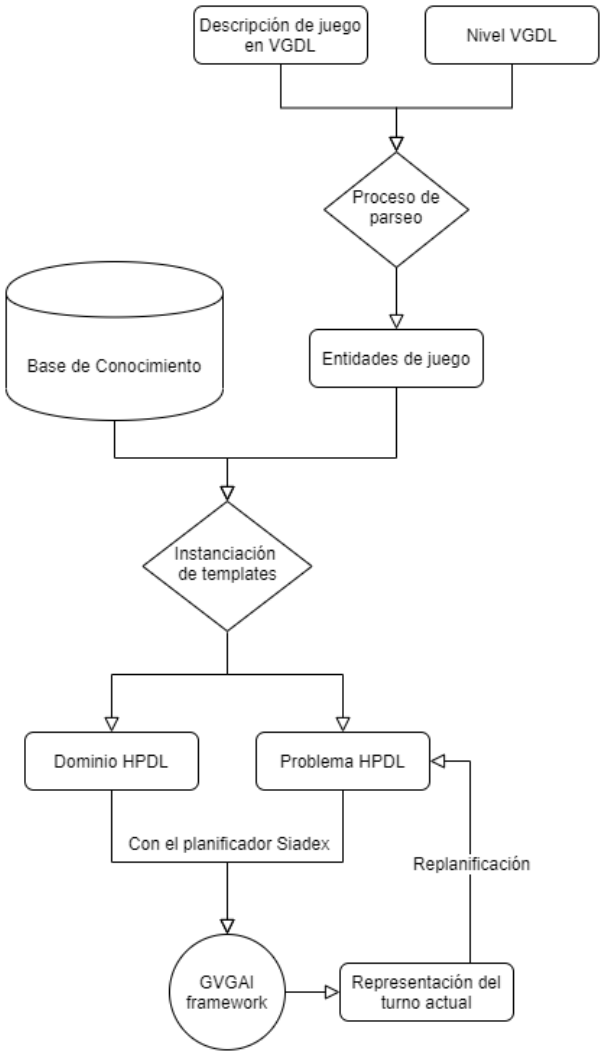


Figura 4.1: Proceso de parseo general para un videojuego.

4. Se lanzan los archivos de salida con el planificador HTN y se produce un plan inicial para GVGAI. Si se desea, se puede activar un proceso de replanificación.

   Este módulo de replanificación, definido como un agente dentro del entorno GVGAI, llama al planificador HTN cada vez que aplica todos los movimientos recibidos en la llamada anterior. Previamente transforma el estado del juego actual en la descripción de un nivel VGDL y, junto a la descripción del juego VGDL, lo pasa por el parser para producir un nuevo dominio y un nuevo problema.

4.1. Entidades de juego

Las entidades de juego representan los conceptos de interés para un dominio HTN definidos en la descripción de juego en VGDL. Según la parte del archivo de descripción se saca una información u otra:

- **SpriteSet**: Se extrae el nombre y el tipo de cada objeto definido. Adicionalmente, puesto que los objetos están declarados de forma jerárquica, es necesario mantener esta estructura entre las entidades. Por último, se almacenan los atributos con carácter funcional (como velocidad o dirección de movimiento), mientras que los centrados en aspecto de visualización son ignorados. Estos atributos se asocian junto a sus entidades para especificar las propiedades de los objetos en HPDL.
- **InteractionSet**: Se almacena el tipo de interacción, los objetos involucrados en cada una y qué rol se asocia a cada uno (emisor o receptor).
- **LevelMapping**: Se guardan las correspondencias entre los caracteres y los objetos de cara a producir correctamente el problema HPDL.

Estas entidades son independientes del lenguaje objetivo hacia el que se traduce, por lo que se podría ampliar este proceso para otros lenguajes siguiendo la misma metodología que con HPDL.

4.2. Base de conocimiento

La base de conocimiento es una colección de plantillas que facilita la traducción de VGDL a HPDL. Cada una de las plantillas se define como una abstracción de código HPDL que permite instanciar las entidades de juego descubiertas en los pasos anteriores. La selección de cada plantilla se realiza en base a los atributos que acompañan a cada una de las entidades.

La base de conocimiento se reparte en tres secciones, responsables de producir las tareas, predicados, métodos y primitivas contenidas en un dominio HPDL,

```
(:action AVATAR_MOVE_UP
    :parameters (?a - <T>)
    :precondition (and
        (can-move-up ?a)
        (orientation-up ?a)
    )
    :effect (and
        (decrease (coordinate_x ?a) 1)
    )
)
```

Figura 4.2: Ejemplo del template de una acción para el movimiento del avatar, donde T indica el tipo de avatar.

al igual que los predicados iniciales en un problema HPDL. Estas secciones están formadas por templates genéricos, a la espera de ser instanciados por las entidades de juego. Las secciones son:

- **Sprites**: Enfocada en representar los objetos del juego (a excepción del avatar).

  Cada sprite acaba transformándose en un objeto en HPDL y, dependiendo del tipo con el que se haya definido, se incluirán primitivas y predicados adicionales. Además, a cada sprite se le asocian funciones HTN para controlar su posición actual y previa. Esto se debe a que existen interacciones en VGDL que devuelven los objetos a posiciones anteriores, como es el caso de *stepBack*. Cuando se trata con objetos movibles, también se incluyen primitivas para controlar la orientación del objeto.

  Como ejemplo, un objeto de tipo **Missile** producirá primitivas y predicados para actualizar su movimiento en cada turno, desplazándose únicamente en la dirección indicada en sus parámetros.

- **Avatar**: Los avatares son un subtipo de *sprites* que representan los agentes en el juego. Las acciones disponibles para cada avatar varían con el tipo de este, y en la mayor parte están limitadas a juegos cuadriculados, incluyendo comúnmente movimientos en las cuatro direcciones cardinales y la posibilidad de usar un *sprite* definido previamente (por ejemplo, una espada con la que atacar enemigos).

  La base de conocimiento almacena predicados y templates para cada posible acción VGDL de un avatar y conoce cuáles están disponibles para cada tipo. Con estos templates, y dependiendo del tipo de avatar, se incluye una estrategia simple, aunque es preferible que un diseñador humano la modifique en base a los objetivos que se requiere que el avatar alcance.

```
(:action MOVING_WALL_STEPBACK
    :parameters (?x - moving ?y - Inmovable)
    :precondition (and
        (= (coordinate_x ?x) (coordinate_x ?y))
        (= (coordinate_y ?x) (coordinate_y ?y))
    )
    :effect (and
        (assign (coordinate_x ?x) (lastCoordinate_x ?x))
        (assign (coordinate_y ?x) (lastCoordinate_y ?x))
    )
)
```

Figura 4.3: Ejemplo de una interacción en un dominio de salida.

En la figura 4.2 se muestra un ejemplo de un template para una primitiva. Después del proceso de parseo dónde se averigua el tipo del avatar, la variable T se instancia y se incluye la primitiva en el dominio de salida.

- **Interacciones**: Se asocia a cada interacción VGDL una primitiva particular y dos templates para los métodos (comprobación y regeneración de comprobación de interacciones). También almacena las tareas globales que controlan la comprobación de colisiones.

Una vez se tienen las entidades de juego y los templates producidos por la base de conocimiento, es necesario realizar una instanciación antes de producir el dominio y el problema HPDL. Con esta idea, la salida se divide según el archivo.

4.3. Domain generator

El dominio se construye de la siguiente forma:

- **HPDL types**.
  Para cada sprite hijo, se refleja la jerarquía definiendo el padre como el tipo del objeto. Cuando se alcanza un sprite sin padre, el tipo asociado es aquel incluído en la descripción VGDL. Todos los tipos de sprites acabarán heredando del objeto más genérico, siendo en el caso de HPDL el supertipo *Object*.

- **HPDL primitives**.
  Por cada posible movimiento del avatar se incluye una nueva acción, añadiendo una adicional para el movimiento nulo (no hacer nada). Para el resto de sprites, se añade una primitiva a aquellos que la necesitan, siendo generalmente los que se actualizan cada turno (objetos movibles como balas o misiles).

De cara a evitar una tarea recursiva que llame a esta primitivas con cada instancia del objeto como parámetro, se ve más conveniente incluir una única primitiva que, haciendo uso de la sentencia *for*, actualice todas las instancias simultáneamente.

Finalmente, por cada interacción se incluye una primitiva que reproduzca sus efectos. Las precondiciones comprobarán que ambas instancias se encuentren en la misma casilla y, según el tipo de interacción, en algunos casos será necesario incluir comprobaciones adicionales.

En la figura 4.3 vemos un ejemplo de una primitiva asociada a una interacción, llamada StepBack. Esta interacción involucra a un sprite de tipo *moving* y a uno de tipo *wall*, haciendo que el objeto *moving* vuelva a la casilla donde se encontrara en el turno anterior.

- **HPDL tasks**.
  Se incluye una tarea global llamada *Turn* conteniendo dos métodos que se comprueban secuencialmente. El primero verifica si se cumplen las condiciones de parada, y el segundo invoca al resto de tareas que forman el turno (detalladas en la sección 4.5).

  Resumidamente, estas tareas son:

  - Una tarea compuesta que engloba todas las posibles primitivas de un avatar, cada una en un método diferente.
  - Un wrapper (una tarea que engloba a una o varias primitivas, sin añadir ninguna comprobación previa) con un único método, común a todas las primitivas del resto de sprites e incluídas como subtareas.
  - Una tarea recursiva para comprobar todas las posibles interacciones en un turno del juego, las cuales no son evaluadas en ningún orden concreto.
  - Una tarea para regenerar los predicados necesarios para evaluar las interacciones en el turno siguiente.

- **HPDL predicates**.
  Además de los predicados necesarios para evaluar las interacciones, es necesario mantener otros dos tipos de predicados: para conocer la orientación de cada objeto movible y para controlar cuando un movimiento está disponible para el avatar.

- **HPDL functions**.
  Para las funciones se debe mantener un control de las posiciones previas y actuales de cada objeto. Esto se debe a que existe interacciones que aparecen con frecuencia, como *stepBack*, que devuelven un objeto a la posición que tenía en el turno anterior. Adicionalmente, para tener en cuenta los posibles criterios de parada es importante mantener contadores de cada tipo de objeto definido y de los recursos que ha obtenido el avatar.

4.4. Problem generator

Al igual que el proceso de generación del dominio, el problema se construye con ayuda de las entidades de juego y los templates de la base de conocimiento. Para cada carácter en la matriz 2D del archivo de descripción de nivel se crea una nueva instancia del objeto en la posición señalada, y se incrementan los contadores de forma apropiada. El objetivo corresponde a una llamada a la tarea recursiva **Turn**, en donde se llama al ciclo del juego y se especifican los criterios de parada.

4.5. HPDL Turn

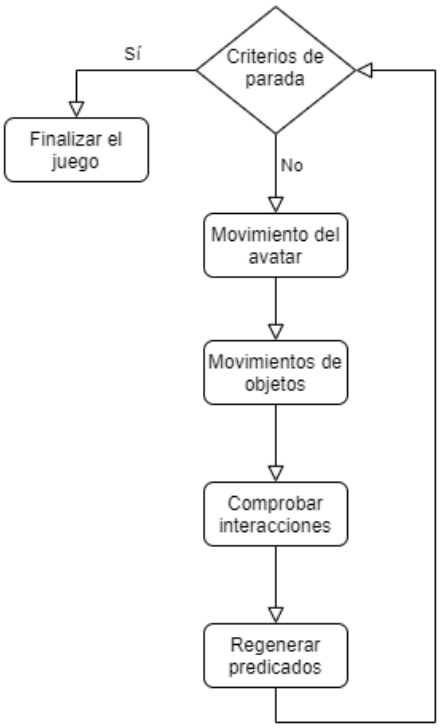


Figura 4.4: Ciclo *Turn* en el que se basan los dominios.

Como solo estamos centrados en representar las dinámicas del juego, y no en resolverlo, la **task-goal** (el objetivo) en los problemas HPDL se corresponde con una llamada a una tarea recursiva **Turn**. Esta tarea, que representa un turno en el juego, se divide en cinco subtareas, como se muestra en la figura 4.4:

- Primeramente, se comprueban los criterios de parada. Si se han completado todos, el planificador termina y devuelve el plan actual. En otro caso, continua.

En nuestra experimentación, al estar centrados en la replanificación, hemos utilizado dos condiciones: que se alcance un turno concreto (por ejemplo, el turno número cinco); o que la cantidad de avatares llegue a cero (es decir, que se pierda la partida).

- Si los criterios no se han dado, se comienza un nuevo ciclo con el siguiente procedimiento:

  1. Se elige un nuevo movimiento para el avatar, dónde la estrategia por defecto escoge el primer movimiento disponible para él.
  2. El resto de objetos no estáticos actualizan su posición, llamando a las primitivas correspondientes de forma secuencial.
  3. Por cada par de objetos actualmente presentes en el juego se comprueba si colisionan, indicando que se debe producir una interacción. En este caso, se invoca la primitiva asociada a la interacción concreta.
  4. Se regeneran todos los predicados necesarios para el turno siguiente.

4.6. Replanificación

El hecho de no usar replanificación implica que desde el primer momento el planificador debe buscar un plan inicial que cumpla los objetivos deseados. Esto hace que, debido al alto factor de ramificación con el que se caracterizan la mayoría de juegos, el tiempo de cómputo para un número grande de turnos sea excesivamente grande. Haciendo uso de un módulo de replanificación no solo optimizamos este tiempo de cálculo, sino que permitimos al agente adaptarse con mayor facilidad a las características no determinísticas de los juegos. De esta manera conseguimos que sea substancialmente más fácil la resolución de objetivos en los juegos implementados.

De cara al funcionamiento, el proceso de replanificación se integra en el interior del framework GVGAI, definido como un agente dentro de este. El proceso que sigue el agente consiste en llamar repetidamente al planificador HTN recibiendo los movimientos para los siguientes turnos, hasta que el juego acabe. A partir de la salida del planificador, se cogen todos los movimientos devueltos que sean propios del agente, y se almacenan para realizarlos en los siguientes turnos.

En caso de no quedar movimientos para la iteración actual, el módulo transforma el estado del juego en la matriz 2D de caracteres que representa un nivel en VGDL. Esto resulta sencillo pues la representación del estado del juego actual en GVGAI sigue la misma conceptualización que en VGDL. Con este nuevo nivel, y junto a la descripción del juego, el módulo llama al proceso de compilación para producir un nuevo dominio y un nuevo problema. Una vez obtenidos los dos archivos, llama al planificador y continua con la ejecución del plan.

Capítulo 5

Implementación

El código fuente, junto a toda la información sobre instalación y uso del software implementado, se puede encontrar en la página [Vellido,].

5.1. Gramática ANTLR

La gramática, mostrada en la figura 5.1, representa la sintaxis y los diferentes conceptos que se incluyen dentro de la descripción de un juego en VGDL. Definida en el formato de ANTLR, permite producir la estructura básica del listener, que la recorrerá extrayendo los tokens necesarios para el proceso de compilación.

Existen ciertos puntos de relevancia para su comprensión:

- INDENT y DEDENT indican respectivamente el aumento y el decremento de indentación. De cara a controlar la jerarquía de sprites es necesario incluir código adicional para saber el nivel de indentación en todo momento.
- Las etiquetas de las reglas se definen con el carácter #, que facilita la forma de referenciarlas en el listener.
- Mediante el carácter = se permite asignar nombres a conjuntos de tokens dentro de las parser rules.
- El comando (− > skip) indica que se ignora el token encontrado.
- WS indica cualquier tipo de espacio en blanco. Por la forma de tratar las indentaciones, se prioriza la detección de estas ante el resto de espacios.
- El orden en el que se definen las reglas es relevante. ANTLR permite escribir gramáticas ambiguas, priorizando en ese caso la primera regla definida.
- En spriteDefinition, si la línea no contiene un segundo token WORD, nos indica que es un sprite hoja.

```
... // (Comandos para manejar la indentación en el SpriteSet)

// ----------------- Parser rules -----------------
basicGame
: 'BasicGame' parameter* nlindent
    (spriteSet | levelMapping | interactionSet | terminationSet
     | DEDENT | nlindent | NL )* DEDENT
;

// ----------------------------
spriteSet
: 'SpriteSet' nlindent (NL* spriteDefinition NL*)+ DEDENT
;

spriteDefinition
: name=WORD WS* '>' WS* spriteType=WORD?
    parameter* nlindent (WS* spriteDefinition NL?)+ DEDENT
    #RecursiveSprite
| name=WORD WS* '>' WS* spriteType=WORD? parameter*
    #NonRecursiveSprite
;

// ----------------------------
levelMapping
: 'LevelMapping' nlindent (NL* LEVELDEFINITION NL*)+ DEDENT
;

// ----------------------------
interactionSet
: 'InteractionSet' nlindent (NL* interaction NL*)+ DEDENT
;

interaction
: sprite1=WORD WORD+ WS* '>' interactionType=WORD parameter*
;

// ----------------------------
terminationSet
: 'TerminationSet' nlindent (NL* terminationCriteria NL*)+ DEDENT
;

terminationCriteria
: WORD parameter* 'win='(TRUE | FALSE) parameter*
;

// ----------------------------
parameter
: left=WORD '=' (WORD | TRUE | FALSE ) #NonPathParameter
| left=WORD '=' (WORD '/' WORD)  #PathParameter
;

nlindent    // Nueva línea seguida de indentación
: NL INDENT
;
```

Figura 5.1

```
// ----------------- Lexer rules -----------------

COMMENT // Se ignoran los comentarios
: '#' ~[\r\n]* -> skip ;

NL  // Nueva línea
| ( '\r'? '\n' | '\r' | '\f' ) SPACES?
)

WS  // Se ignoran los espacios en blanco
: [ \t]+ -> skip ;

TRUE
: [Tt] [Rr] [Uu] [Ee]
;
FALSE
: [Ff] [Aa] [Ll] [Ss] [Ee]
;

WORD
: CHAR (CHAR | UNDERSCORE | NUMBER)*
| NUMBER
;

LEVELDEFINITION
: WS* SYMBOL WS* '>' WS* (WORD WS*)+
;

ANY     // Se ignora todo aquello no reconocido
: .
;

// ----------------- Fragments -----------------

fragment SPACES
: [ \t]+ ;
fragment SYMBOL
: ~[ ] ;
fragment CHAR
: [a-zA-Z/.] ;
fragment UNDERSCORE
: '_' ;
fragment NUMBER
: '-'? [0-9]+ ([.,] [0-9]+)? ;  // Cualquier número real
```

Figura 5.1: Gramática ANTLR resumida.

Puesto que la descripción de un nivel en VGDL sigue una conceptualización bastante sencilla, es posible parsearla manualmente sin necesidad de definir ninguna gramática.

5.2. Listener

El listener consta de una clase responsable de recorrer el árbol generado a partir del archivo de descripción de juego y extraer la información de relevancia para el proceso de compilación.

Gracias a la gramática ANTLR, este listener se genera automáticamente con métodos que se activan cuando se entra y se sale de cada nodo del árbol. Estos nodos van asociados a cada parser rule definida en la gramática, y junto a las etiquetas y la asociación de tokens resulta fácil de acceder a las distintas partes de cada regla.

Además de las entidades de juego, detalladas más adelante, se extrae la siguiente información:

- **ShortTypes**: Lista de los caracteres que pueden aparecer en un archivo de nivel.
- **LongTypes**: Tipos que pueden aparecer en un archivo de nivel.
- **Stypes**: Tipos definidos en el SpriteSet. Este conjunto es mayor que el de LongTypes, pues incluye a los padres y a los tipos de estos.
- **Hierarchy**: Diccionario con los hijos asociados a cada padre.
- **TransformTo**: Objetos que pueden transformase en otros. Son los afectados por la interacción *transformTo*.

Para sacar alguno de estos elementos el listener analiza el nivel del juego asociado al problema.

5.2.1. Entidades de juego

Como se muestra en la figura 5.2, existen cuatro clases que almacenan la información de cada línea del archivo de descripción de juego. Mayoritariamente de la clases *Sprite* e *Interaction* se sacan las entidades de juego, elementos que serán instanciados en los templates.

Estas clases no implementan funcionalidad por sí misma, sino que encapsulan la información de cara a facilitar el acceso al resto de clases.

```
Sprite
|-- Name            : Nombre del objeto
|-- Type            : Tipo del objeto
|-- Parent          : En caso de que tenga, padre
|__ Parameters      : Parámetros del objeto

LevelMap
|-- Char            : Carácter que representa al objeto
|__ Sprites         : Tipo del objeto representado

Interaction
|-- SpriteName      : Nombre del objeto que produce la interacción
|-- PartnerName     : Objeto que recibe los efectos
|-- Type            : Tipo de interacción
|__ Parameters      : Parámetros de la interacción

Termination
|-- Type            : Criterio de parada
|-- Win             : Si el agente gana o pierde la partida
|__ Parameters      : Parámetros del criterio de parada
```

Figura 5.2: Clases de las entidades de juego y sus atributos.

```
AvatarHPDL
|-- AvatarActions
|-- AvatarLevelPredicates
|__ AvatarPredicates

SpriteHPDL
|-- SpriteActions
|__ SpriteLevelpredicates

InteractionHPDL
|-- InteractionActions
|__ InteractionMethods
```

Figura 5.3: Clases que forman la base de conocimiento relacionadas con VGDL.

5.2.2. Base de conocimiento

La base de conocimiento se almacena como un conjunto de clases estructuradas en tres módulos (como se muestra en la figura 5.3). Estos módulos a su vez encapsulan diferentes clases, las cuales reciben las entidades de juego, y cada módulo devuelve templates instanciados de una parte del dominio/problema.

Aunque todos los módulos están adaptados para que puedan producir elementos de cualquier parte del dominio/problema, para la estado inicial del proyecto no es necesario de que cada uno haga uso de toda la funcionalidad disponible.

Los módulos se dividen en las siguientes clases:

5.2.3. AvatarHPDL

Recibe como entrada la instancia de la clase Sprite (desde las entidades de juego) que corresponde al avatar; la jerarquía; y en caso de poder usar algún objeto, la instancia Sprite correspondiente.

Para cada elemento devuelve información diferente:

- **Actions**: Según el tipo del avatar construye e instancia los templates de los movimientos disponibles. Corresponde a primitivas del dominio HPDL.
- **Predicates**: Devuelve predicados en base al tipo del avatar.
- **Methods**: Genera un método wrapper para cada primitiva.
- **Tasks**: Construye una única tarea con la estrategia básica, incluyendo todos los métodos en ningún orden concreto.
- **LevelPredicates**: Predicados asociados a un problema específicos de un avatar. Principalmente compuesto por predicados para comprobar si un movimiento está o no disponible.

5.2.4. SpriteHPDL

Se genera una instancia con cada tipo diferente de objeto distinto de avatar.
Recibe como entrada la instancia del objeto Sprite; la jerarquía; y el padre en caso de que tenga.

Para cada elemento devuelve información diferente:

- **Actions**: Construye y devuelve primitivas en base al tipo de objeto que sea.
- **LevelPredicates**: Predicados para el problema específicos del objeto. En la versión actual del software solo concierne a la orientación de los tipos de misiles.

5.2.5. InteractionHPDL

Se genera una instancia por cada tupla **(interacción, sprite, sprite)**, es decir, no solo por cada interacción sino por cada pareja de objetos asociada a ella.

Recibe como entrada la instancia de Interaction; las dos instancias Sprite de los objetos involucrados; y la jerarquía.

Para cada elemento devuelve información diferente:

- **Actions**: Acción asociada a la interacción.
- **Methods**: Genera el wrapper de la primitiva comprobando las condiciones para que se produzca, y modificando los predicados pertinentes para que se realice de nuevo la comprobación en el turno siguiente.

Estos módulos hacen uso de un conjunto de clases auxiliares que encapsulan las diferentes partes de un dominio HPDL. Estas clases corresponden a la estructura de tarea (Task), método (Method) y primitiva (Action). Se almacenan conjuntamente en un archivo llamado *typesHPDL* y permiten acceder con facilidad a las distintas partes que las forman.

5.3. Generación de salida

5.3.1. Domain y problem generators

El domain generator es una clase que, en base a las plantillas y a las estructuras obtenidas a partir del listener, termina de construir cada parte del dominio.

Concretamente se encarga de:

- Definir la sección **:types** reflejando la estructura jerárquica del juego.
- Incluir los predicados recibidos por la base de conocimiento.
- Construir los predicados genéricos de posición actual y posición previa. Añadir predicados necesarios para la evaluación de interacciones.
- Crear la tarea *Turn* y sus subtareas, incluyendo los métodos obtenidos de la base de conocimiento donde corresponda.
- Añadir las primitivas en el dominio.

Como salida devuelve, en forma de cadena de caracteres, cada una de las diferentes estructuras que forman el dominio HPDL (predicates, functions, etc.).

El problem generator actua de manera similar, recibiendo información tanto del listener como de la base de conocimiento. También se preocupa de:

- Definir las instancias del nivel y asignarle los predicados de posición.
- Inicializar los predicados para la primera evaluación de interacciones.
- Inicializar los contadores de cada tipo de objeto.

Además, el problem generator cuenta con una clase auxiliar, *LevelObjects*, conteniendo información sobre cada instancia que aparece en el nivel. Esta clase facilita el acceso a la información sobre las instancias a la hora de generar los predicados necesarios para el problema.

5.3.2. Writers

Por último, existe dos scripts encargados de escribir la salida de los generators en pantalla/disco. Estos scripts construyen completamente el dominio o problema de salida, en base a las diferentes partes de cada uno devueltas por los generators.

Principalmente, se encargan de darle el formato necesario del lenguaje HPDL (paréntesis, requisitos, etc.).

5.4. Replanificación

Para la replanificación creamos un nuevo módulo conteniendo el framework GVGAI. En su interior definimos un nuevo agente que, cada vez que haya pasado el número de turnos deseado, genera un nuevo archivo VGDL de nivel en base al estado del juego y llama al proceso de parseo. El nuevo dominio y problema recibidos son invocados mediante el planificador HTN para obtener un nuevo plan, del que se desechan todas las acciones no producidas por el agente (solo se cogen sus movimientos) y se almacenan para realizarlas en los siguientes turnos.

Este módulo cuenta con un archivo de configuración que permite elegir el juego y el nivel sobre el que funcionar, el cuál se lee antes de arrancar el entorno GVGAI.

5.5. Estructuración

Para una mejor organización y de forma que se facilite la extensibilidad del proyecto, se ha optado por la siguiente estructura de directorios, mostrada en la figura 5.4.

Esta estructura permite extender las capacidades del software de manera sencilla, pues en el directorio *planners* se podría incluir nuevos planificadores, y cada nuevo lenguaje se añadiría al directorio *src* con su propia base de conocimiento.

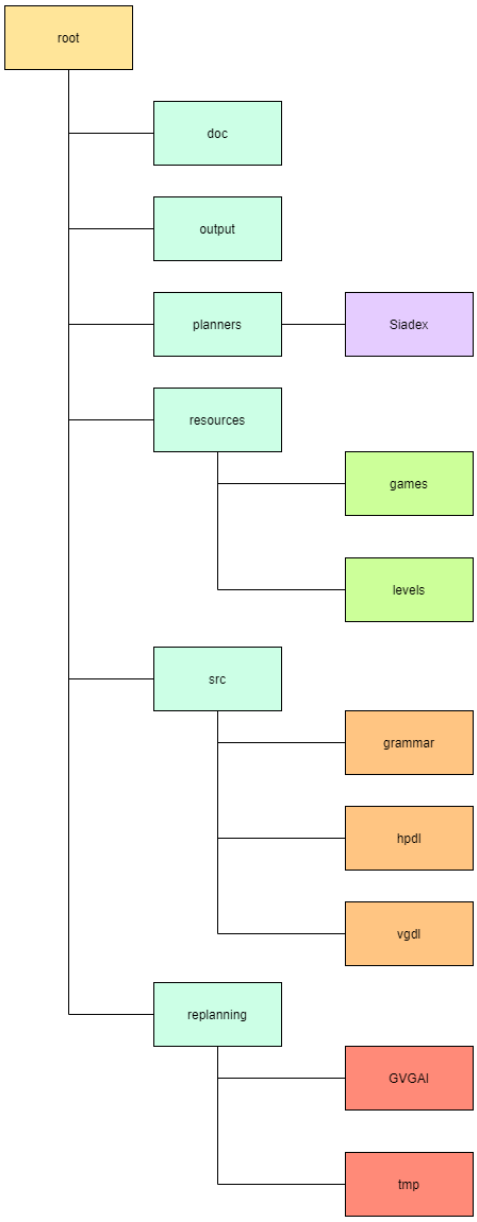


Figura 5.4: Estructura de directorios.

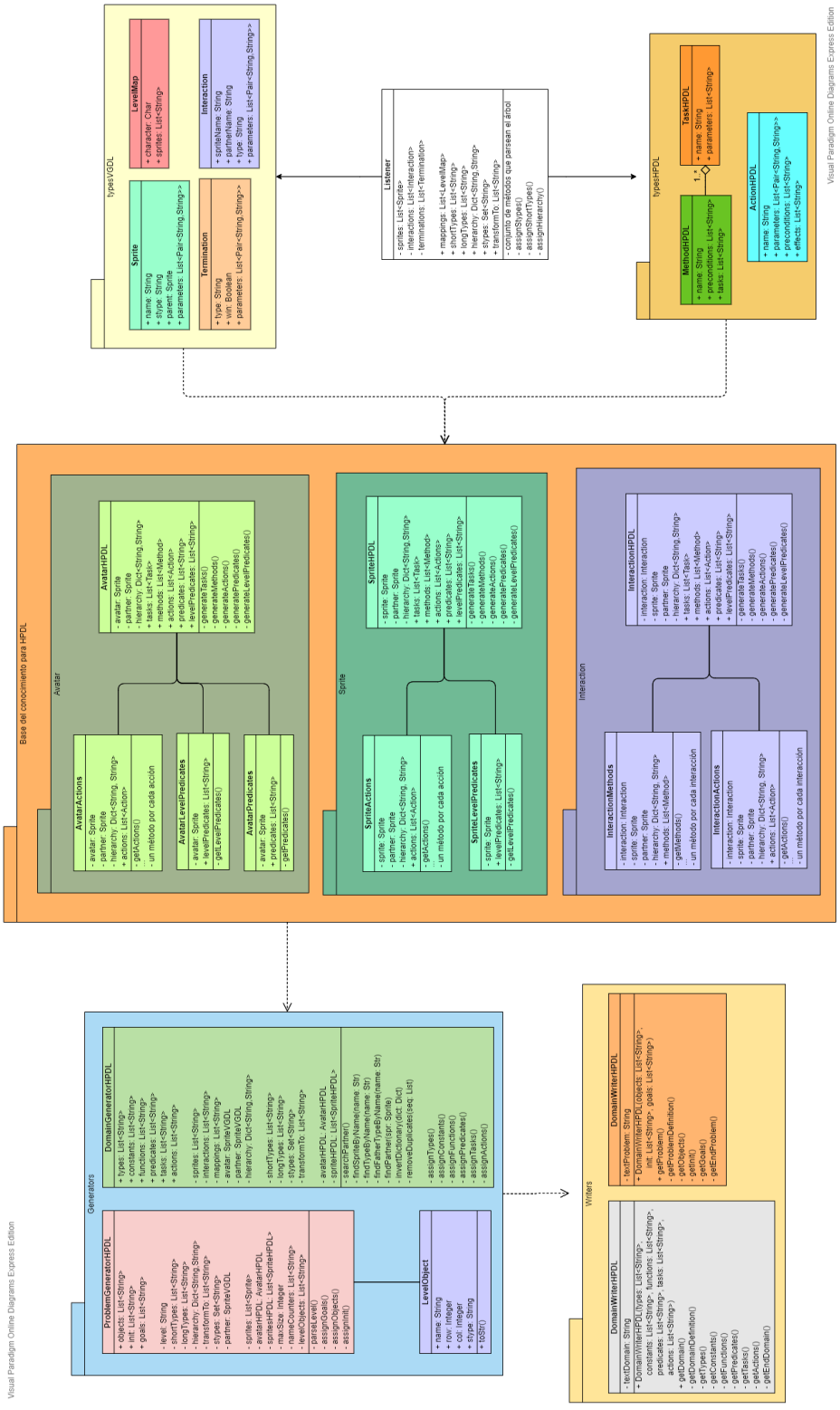


Figura 5.5: Diagrama de clases.

Se detalla el contenido de cada directorio:

- **doc**: Documentación del código asociado al proyecto.
- **output**: Carpeta donde almacenar el dominio y el problema de salida.
- **planners**: Directorio dónde almacenar los planificadores a utilizar. En este proyecto actualmente solo se sitúa Siadex.
- **resources**: Contiene una subcarpeta con las descripciones VGDL de juegos, y otra con niveles asociados a ellos.
- **src**: Código del programa, separado en base al lenguaje que se asocia. Se cuenta con la gramática ANTLR y el listener; clases asociadas a la producción de información HPDL; y clases con conocimiento exclusivo de VGDL.
- **replanning**: Módulo de replanificación, que cuenta con el framework GVGAI, dónde se define el agente, y una carpeta temporal auxiliar al proceso.

En la figura 5.5 se muestran el conjunto de módulos y las correspondientes clases que componen el proyecto, siguiendo los detalles de implementación explicados en los apartados anteriores.

5.6. Ejecución

El formato de ejecución del parser queda de la siguiente manera:

```
$ main.py [-h] -gi GAMEINPUT [-li LEVELINPUT]
           [-go GAMEOUTPUT] [-lo LEVELOUTPUT] [-vh]

Argumentos opcionales:
    -h,              --help                          Mensaje de ayuda
    -gi GAMEINPUT,   --gameInput   GAMEINPUT         Archivo de juego de entrada
    -li LEVELINPUT,  --levelInput  LEVELINPUT        Archivo de nivel de entrada
    -go GAMEOUTPUT,  --gameOutput  GAMEOUTPUT        Archivo de juego de salida
    -lo LEVELOUTPUT, --levelOutput LEVELOUTPUT       Archivo de nivel de salida
    -vh,             --verboseHelp                   Muestra información adicional
```

Aunque se proporciona un archivo Makefile para facilitar el uso el software.

Para el proceso de replanificación, la ejecución se realiza desde dentro del framework GVGAI, definiendo previamente en el archivo de configuración proporcionado el juego y el nivel que se desean probar.

Capítulo 6

Experimentación

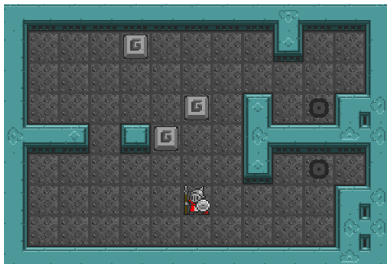
Para comprobar la calidad de los dominios producidos y del proceso de ingeniería del conocimiento, se ha experimentado con cuatro juegos diferentes, dos de puzles y dos reactivos, incrementando gradualmente la complejidad en la representación.

Estos juegos son:

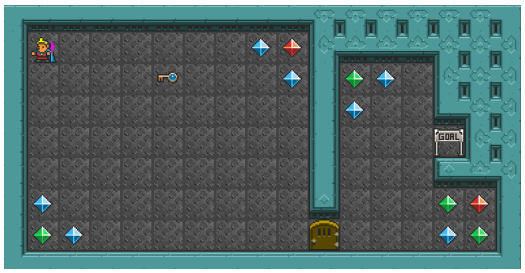
- **Sokoban**
  Juego de puzles en el cuál el agente debe empujar todas las cajas en casillas específicas del mapa, sobre las que caen y desaparecen. El avatar no puede agarrar ninguna de las cajas, debe empujarlas en la misma dirección en la que se mueve. El juego no incluye ningún tipo de NPCs ni no-determinismo, volviéndolo el más simple de aquellos con los que se ha experimentado.

- **Brainman**
  Otro juego de puzles consistente en hacer que el agente llegue a la salida. En un estilo similar al Sokoban, debe empujar una serie de llaves repartidas por el mapa hacia las puertas para poder abrirlas. Estas llaves no pueden ser cogidas por el agente y se comportan como misiles, lo que implica que cuando se empujan mantienen la misma dirección hasta que colisionan con algún objeto. Adicionalmente existen gemas como recursos para incrementar la puntuación del agente.

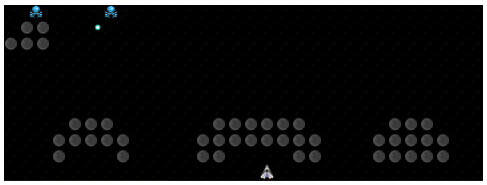
- **Aliens**
  Adaptación del clásico juego de arcade, en dónde el avatar debe matar a todos los aliens que aparecen sobre él. Para defenderse, el agente solo dispone de movimientos horizontales y de disparos en sentido ascendente. En esta versión los aliens se mueven de manera aleatoria hacia los laterales y lanzan bombas hacia el avatar, volviéndolo altamente no determinista y difícil de representar.



(a) Sokoban.



(b) Brainman.



(c) Aliens.

(d) Boulderdash.

Figura 6.1: Imágenes de los cuatro juegos con la interfaz gráfica de GVGAI.

- **Boulderdash**
  El objetivo en este juego es recoger nueve gemas en el mapa y moverse hacia la salida. Existen multitud de enemigos y rocas para dificultar el movimiento del agente en su camino.

  Existen dos tipos de terreno, excavado y vacío. Las rocas se desplazan hacia abajo cuando bajo ellas no se encuentra ningún objeto "rígido" (es decir, un agente, un NPC o terreno excavado). Para abrirse paso, el personaje tiene a su disposición una pala que le permite cavar el terreno, pasando este de excavado a vacío. En ocasiones el uso de esta pala es forzosamente necesario para resolver los niveles, aunque su uso se debe hacer con precaución pues puede hacer caer rocas en posiciones no deseadas.

  Boulderdash se caracteriza por ser fuertemente no-determinista, multi-objetivo, y en ocasiones irresoluble, lo que lo hace el más complejo de estos cuatro juegos.

Todos los juegos fueron tratados con el mismo procedimiento: se realiza un proceso de compilación desde VGDL a HPDL, partiendo de la descripción del juego y produciendo una representación de las dinámicas del juego en forma de primitivas, tareas..., e incluyendo una estrategia simple para el agente. Esta estrategia selecciona la primera acción disponible para el avatar, sin considerar objetivos. De la misma manera se produce una instancia de un problema a partir de una descripción de nivel, principalmente con uno de los niveles usados en la competición de GVGAI.

```
Sokoban:
:action (AVATAR_MOVE_UP A0)
:action (BOX_AVATAR_BOUNCEFORWARD Y2 A0)
:action (BOX_HOLE_KILLSPRITE Y2 X6)
-----------------------------------------------------------------
Brainman:
:action (AVATAR_MOVE_UP A0)
:action (KEYM_MOVE)
:action (AVATAR_WALL_STEPBACK A0 w17)
-----------------------------------------------------------------
Aliens:
:action (AVATAR_USE A0 partner)
:action (MISS_MOVE)
:action (ALIEN_PRODUCE)
:action (PORTALSLOW_PRODUCE)
:action (PORTALFAST_PRODUCE)
:action (AVATAR_BOMB_KILLSPRITE A0 partner)
-----------------------------------------------------------------
Boulderdash:
:action (AVATAR_MOVE_UP A0)
:action (BOULDER_MOVE)
:action (DIRT_AVATAR_KILLSPRITE z17 A0)
:action (BOULDER_DIRT_STEPBACK o2 w25)
:action (BOULDER_DIAMOND_STEPBACK o0 x1)
```

Figura 6.2: Ejemplos de planes devueltos para un turno en cada juego.

```
:action (AVATAR_MOVE_UP A0)                          ┐
:action (BOULDER_MOVE)                               │
:action (DIAMOND_AVATAR_COLLECTRESOURCE x3 A0)       ├ Primer turno
:action (BOULDER_DIRT_STEPBACK o2 w25)               │
:action (BOULDER_DIAMOND_STEPBACK o0 x1)             ┘

:action (AVATAR_MOVE_UP A0)                          ┐
:action (BOULDER_MOVE)                               │
:action (DIRT_AVATAR_KILLSPRITE z17 A0)              ├ Segundo turno
:action (BOULDER_DIRT_STEPBACK o2 w25)               │
:action (BOULDER_DIAMOND_STEPBACK o0 x1)             ┘

…
```

Figura 6.3: Porción del plan devuelto por Siadex, representando dos turnos en *Boulderdash*. Las dos primeras primitivas de cada turno en este plan indican los movimientos del avatar y del resto de objetos. Las tres siguientes corresponden a las interacciones producidas

	Elementos sintácticos							Elementos semánticos			
	Primitives	Tasks	Methods	Types	Supertypes	Predicates	Functions	Determinismo	Objetos movibles	Acciones del agente	Interacciones
Sokoban	14	5	20	5	3	10	14	Yes	No	4	5
Brainman	33	5	38	11	5	11	22	Yes	Yes	4	23
Aliens	17	5	19	10	9	9	25	No	Yes	3	9
Boulderdash	28	5	33	10	9	12	27	No	Yes	5	17

Figura 6.4: Características sintácticas y semánticas de los cuatro dominios producidos.

	Turnos	Acciones	Expansiones	Tiempo (s)
Sokoban	10	14	2075	0.00999
	20	24	3945	0.01748
	50	54	9555	0.04111
Brainman	10	21	6049	2.08517
	20	41	12279	3.49277
	50	101	30969	8.73218
Aliens	10	50	411	0.00499
	20	100	821	0.01111
	50	250	2051	0.02489
Boulderdash	10	126	844178	36.4963
	20	196	1244760	55.2729
	50	257	1737375	72.6214

Figura 6.5: Resultados de ejecución con Siadex. Cada dominio se lanzó con 10, 20 y 50 turnos.

La experimentación consiste en verificar visualmente la correctitud de los planes devueltos, ya sea mediante la salida del planificador Siadex o mediante el entorno GVGAI. Para ello se repite el proceso anterior con múltiples niveles diferentes y se comprueba la correctitud en cada uno de ellos. En la figura 6.2 se pueden ver ejemplos de planes devueltos para cada uno de los casos de estudio realizados

Como se muestra en la figura 6.3, el planificador devuelve una serie de acciones incluyendo los movimientos del avatar y del resto de objetos, además de una acción para cada interacción que se haya producido en cada turno. Para transformar este plan a GVGAI solo es necesario coger las acciones correspondientes al movimiento del avatar, puesto que el resto únicamente actualizan el estado del mundo en el planning state (representación del mundo para el planificador en forma de predicados).

La figura 6.4 muestra detalles sobre los dominios producidos, incluyendo la cantidad de tareas, predicados, etc. producidas. También se recogen características semánticas de los juegos. Adicionalmente, en la figura 6.5 mostramos resultados de acciones, expansiones y el tiempo requerido para distintas ejecuciones de

los dominios, y para niveles de similar complejidad entre juegos (concretamente el primer nivel de cada uno en el framework GVGAI).

A partir de las tablas podemos ver que cuando la complejidad del juego aumenta (esto se aprecia en el número de interacciones y de tipos distintos) las dimensiones del dominio se incrementan de manera similar.

Puesto que el no-determinismo no se representa en los dominios, vemos que los juegos de puzles pueden acabar siendo computacionalmente más costosos que algunos de los reactivos.

Por ejemplo, a partir de los resultados de 6.5, apreciamos que Boulderdash conlleva mayor tiempo de cómputo pues es el único donde en cada turno se calculan las nuevas posiciones de los proyectiles. Existen otros juegos con proyectiles, las llaves en Brainman y los disparos en Aliens, pero estos juegos no reflejan este comportamiento.

Esto se debe a que en Brainman hay que esperar a que el agente empuje a una llave, mientras tanto permanecerá estática. En el caso de Aliens, debido a que los NPCs disparan aleatoriamente los proyectiles no se actualizan a no ser que ya hayan sido definidos al principio del nivel (puesto que el hecho de disparar, al ser no-determinista, no está reflejado).

Adicionalmente, cada roca en Boulderdash en la mayoría de casos interacciona con la casilla de abajo, añadiendo aún más cómputo para el planificador.

Todo esto nos indica que la representación del no-determinismo es importante, aunque la dinámica de este tipo de eventos haga los planes grandes inútiles. Por ello creemos que una aproximación reactiva con replanificación puede producir resultados de mucha mejor calidad a la hora de cumplir objetivos.

6.1. Estrategia manual

Además de seguir la misma experimentación que con el resto de juegos, para el juego *Boulderdash* se definió manualmente una estrategia al agente para comprobar cómo funcionaría un dominio de estas características con una estrategia de mayor calidad.

La estrategia, tal y como se muestra en la figura 6.6, sigue el siguiente procedimiento:

1. Comprobar si se ha elegido una gema como objetivo. En caso de que no haya ninguna, se calcula la distancia Manhattan del agente a cada una de ellas, y se elige la que se encuentre a menor distancia.

2. Se afirman como predicados los movimientos posibles en ese turno para el avatar. Un movimiento es posible si, en el caso de que el agente se moviera en esa dirección, no colisionaría con ningún enemigo, roca o muro.

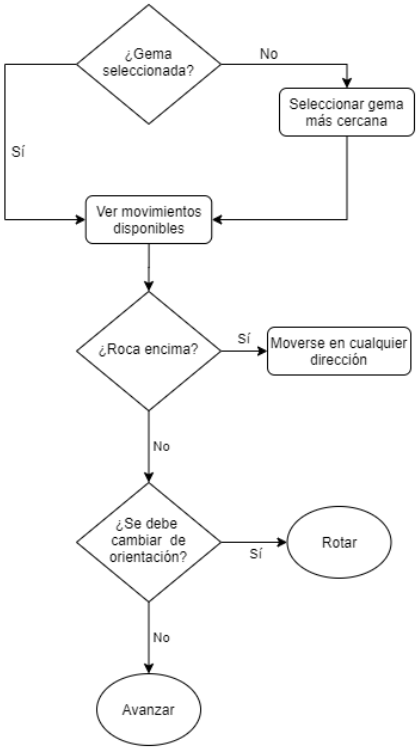


Figura 6.6: Estrategia definida a mano para el agente de Boulderdash.

3. Se comprueba si hay una roca en la casilla inmediatamente superior al avatar. Si existe, es necesario moverse en cualquier dirección, pues sino la gema mataría al agente en ese turno.

4. En otro caso se busca aquel movimiento de los disponibles que más reduce la distancia Manhattan hacia la gema. Para ello se mira si es necesario rotar (no se está en la misma fila/columna que la gema), y se elige el movimiento de forma apropiada.

Se pudo comprobar que el agente era capaz de resolver los objetivos, alcanzando gemas mientras esquivaba rocas durante su camino, tal y como se muestra en la figura 6.7 a través de uno de los niveles de GVGAI.

Consideramos esta estrategia simple para la complejidad del juego, pero viéndo cómo es capaz de conseguir gemas en zonas fáciles tenemos la certeza de que una estrategia de mayor elaboración podría completar el juego con muy buenos resultados.

Nota: Todos los resultados obtenidos de la experimentación se pueden encontrar en la página [Vellido,].

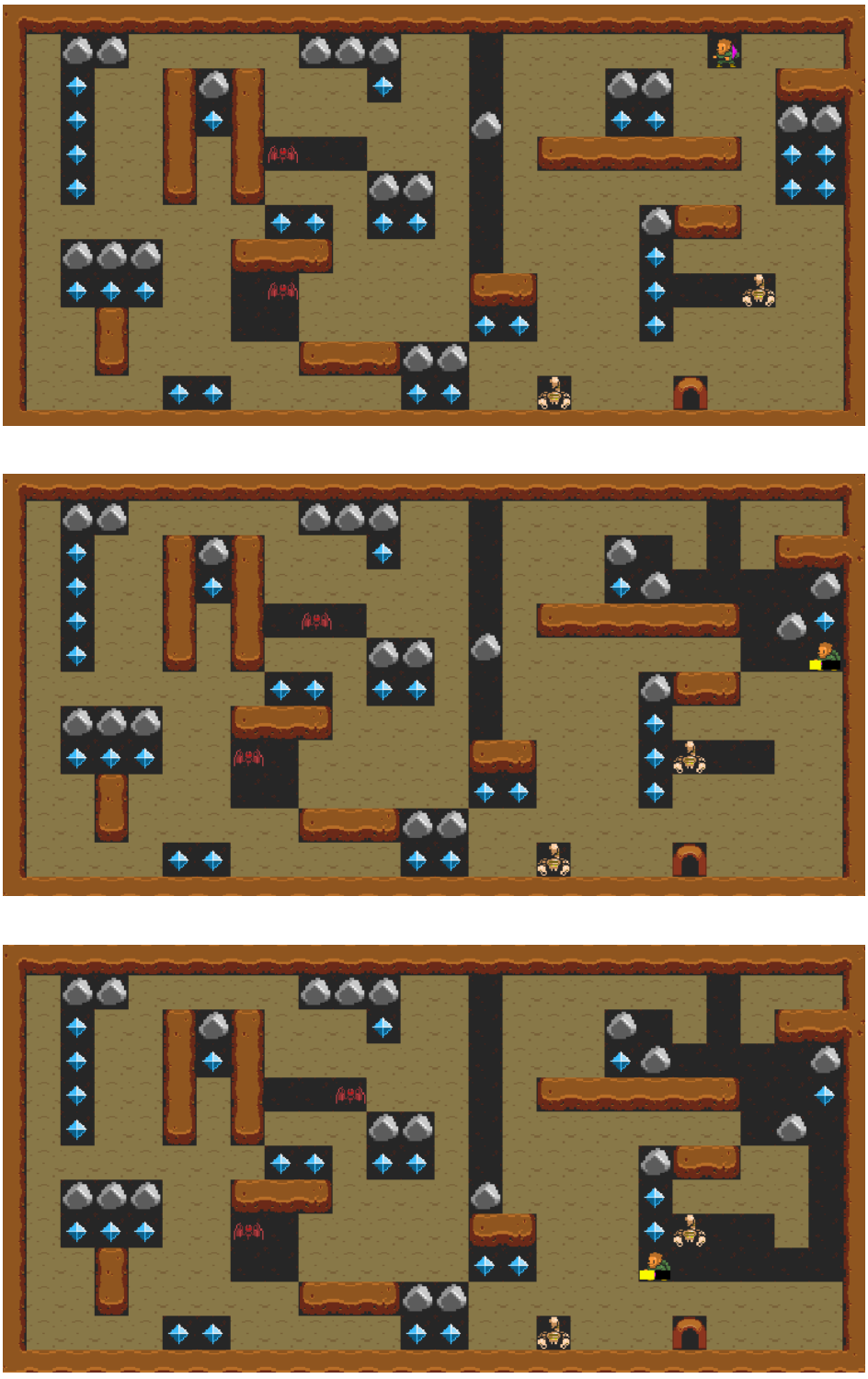


Figura 6.7: Ejemplo del comportamiento del agente siguiendo la estrategia manual.

Capítulo 7

Conclusiones

Con este trabajo pretendemos reabrir una pregunta aún sin resolver: ¿Es posible simplificar la definición de dominios de planificación complejos de manera automática?

Hemos visto que sí es posible. Combinando una base de conocimiento y un proceso de compilación, junto al uso de un lenguaje simple pero expresivo, podemos crear un procedimiento automático capaz de definir una gran variedad de dominios de planificación diferentes.

Por otro lado, la sencilla estructura en la que se basan los dominios permite modificar fácilmente la estrategia del agente de manera que puedan ser utilizados para la resolución de juegos que se adapten a las características de VGDL.

Vemos adicionalmente que este software podría emplearse en la enseñanza, de forma que los estudiantes se introduzcan en la planificación. Pudiendo interactuar con GVGAI a partir de un dominio complejo, ellos mismos podrían definir la estrategia del agente para entrenarse. Esto les permitiría comprobar visualmente la calidad de la estrategia implementada, cosa más sencilla de comprender que un plan devuelto por el planificador.

7.1. Trabajos futuros

A pesar de todo lo logrado en el proyecto, se aprecia que existe un número de puntos en los que este puede mejorar, y podrían ser considerados como trabajos futuros:

- **Representación del no-determinismo**. Este tipo de situaciones aparecen frecuentemente no solo en videojuegos, sino en problemas reales. En algunos casos un proceso continuo de replanificación y una estrategia reactiva puede ser suficientes para enfrentarse a este tipo de problemas. Aunque, por otra parte, si pretendemos crear un único dominio versátil y autosuficiente, creemos que la mejor aproximación sería incorporar técnicas de planificación probabilística.

- **Una estrategia para el agente de mayor complejidad**. Aunque somos capaces de representar las dinámicas en los videojuegos, vemos complicaciones a la hora de integrar una estrategia aprendida para el agente, capaz de resolver objetivos, dentro de los dominios generados por el proceso de conocimiento expuesto.

  Esta tarea no es sencilla, puesto que ya no estamos considerando un único proceso de parseo. Es necesario analizar las mecánicas del juego y comprenderlo. Creemos posible que el descubrimiento automático de estrategias de juego mediante minería de procesos, como las usadas en [Segura-Muros et al., 2017], las cuales toman como entrada trazas de planes de ejecuciones reales del juego, pueden ser una buena manera de afrontar este problema.

- **Mayor cantidad de lenguajes**. Queda claro que los lenguajes usados en el proceso de compilación son muy influyentes en la calidad de los resultados. VGDL y HPDL han sido propuestos como candidatos por su fácil integración con el entorno GVGAI. Este proceso se puede generalizar para otros lenguajes HTN y otras descripciones de juegos e interacciones que mantengan características similares a las mencionadas en este proyecto.

  Como trabajo futuro se puede afrontar la producción de estos dominios de planificación en lenguajes como HDDL [Höller et al., 2019] y SHOP [Nau et al., 2003].

Bibliografía

[Castillo et al., 2010] Castillo, L., Morales, L., González-Ferrer, A., Fdez-Olivares, J., Borrajo, D., and Onaindia, E. (2010). Automatic generation of temporal planning domains for e-learning problems. *Journal of Scheduling*, 13:347–362.

[Couto Carrasco, 2015] Couto Carrasco, M. (2015). Creació d'un controlador automàtic pel concurs gvg-ai. http://hdl.handle.net/10230/25487.

[Fdez-Olivares et al., 2011] Fdez-Olivares, J., Castillo, L., Cózar, J. A., and García Pérez, O. (2011). Supporting clinical processes and decisions by hierarchical planning and scheduling. *Computational Intelligence*, 27(1):103–122.

[Fdez-Olivares et al., 2006] Fdez-Olivares, J., Castillo, L., García-Pérez, Ó., and Palao, F. (2006). Bringing users and planning technology together. experiences in siadex. pages 11–20.

[Fox and Long, 2003] Fox, M. and Long, D. (2003). Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124.

[Ghallab et al., 2016] Ghallab, M., Nau, D., and Traverso, P. (2016). *Automated Planning and Acting*. Cambridge University Press, USA, 1st edition.

[González-Ferrer et al., 2013] González-Ferrer, A., Fernández-Olivares, J., and Castillo, L. (2013). From business process models to hierarchical task network planning domains. *The Knowledge Engineering Review*, 28(2):175–193.

[GVGAI,] GVGAI. java-vgdl. https://github.com/GAIGResearch/GVGAI/wiki/VGDL-Language.

[Haslum et al., 2019] Haslum, P., Lipovetzky, N., Magazzeni, D., and Muise, C. (2019). An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187.

[Höller et al., 2019] Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D., and Alford, R. (2019). Hddl – a language to describe hierarchical planning problems.

[McCluskey et al., 2016] McCluskey, T. L., Vaquero, T., and Vallati, M. (2016). Issues in planning domain model engineering.

[Nau et al., 2003] Nau, D. S., Au, T. C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20:379–404.

[Parr and Quong, 1995] Parr, T. J. and Quong, R. W. (1995). Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

[Perez-Liebana et al., 2016] Perez-Liebana, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S. M., Couëtoux, A., Lee, J., Lim, C., and Thompson, T. (2016). The 2014 general video game playing competition.

[Schaul,] Schaul, T. A video game description language (vgdl) built on top of pygame. https://github.com/schaul/py-vgdl.

[Schaul, 2013] Schaul, T. (2013). A video game description language for model-based or interactive learning. pages 1–8.

[Segura-Muros et al., 2017] Segura-Muros, J. A., Pérez, R., and Fernández-Olivares, J. (2017). Learning htn domains using process mining and data mining techniques.

[TIN2015-71618-R,] TIN2015-71618-R. Plan miner: Integración de planificación automática y minería de procesos para el aprendizaje de dominios de planificación jerárquica a partir de la experiencia almacenada en registros de actividad.

[Vellido,] Vellido, I. Vgdl to htn parser. https://github.com/IgnacioVellido/VGDL-to-HTN-Parser.

[Vereecken,] Vereecken, R. Vgdl 2.0. https://github.com/rubenvereecken/py-vgdl.