UNIVERSIDAD POLITÉCNICA DE MADRID

**DEPARTAMENTO DE AUTOMÁTICA, INGENIERÍA
ELECTRÓNICA E INFORMÁTICA INDUSTRIAL**

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES

**DISEÑO E IMPLEMENTACIÓN DE UN
PLANIFICADOR PARA UN AGENTE AUTÓNOMO**

TESIS DOCTORAL

Autor: Cristóbal Tapia García

Directores: Ramón Galán López

Doctor en Informática

Pablo San Segundo Carrillo

Doctor Ingeniero Industrial

2017

Tribunal nombrado por el Magnífico y Excmo. Sr. Rector de la Universidad Politécnica de Madrid, el día ____ de _____ de 2017.

Presidente: _____

Secretario: _____

Vocal: _____

Vocal: _____

Vocal: _____

Suplente: _____

Suplente: _____

Realizado el acto de defensa de la Tesis el día ____ de _____ de 2017 en la E.T.S.I.I.

EL PRESIDENTE

LOS VOCALES

EL SECRETARIO

AGRADECIMIENTOS

Es de justicia dedicar los primeros agradecimientos de esta tesis a aquellos que más la han sufrido: a mi mujer Marta y a mis hijos: Pablo y Alicia. A ellos les dedico este trabajo.

También al resto de mi familia: mis padres, mis hermanos y sus esposas, y mis sobrinos. Incluyo aquí a mis suegros y a mi cuñado, cuyo apoyo ha sido inestimable.

A mis tutores de tesis: Ramón Galán y Pablo San Segundo. Así como al grupo de investigación al que pertenezco (Control Inteligente), y al resto de profesores y personal de DISAM, incluyendo a Rosa, a Tere y a Carlos.

Asimismo, a mis compañeros en el antiguo departamento ELAI, ahora incluido en el Departamento de Ingeniería Eléctrica, Electrónica, Automática y Física Aplicada de la ETSIDI.

Quiero mencionar también a los integrantes del grupo de planificación y aprendizaje en la UC3M.

Por último, a sabiendas de que me dejo a muchos y muchas en el tintero, a todos aquellos que han ayudado a completar esta tesis, mi agradecimiento.

ÍNDICE DE CONTENIDOS

Índice de contenidos	v
Índice de figuras.....	ix
Índice de tablas	xi
Resumen.....	xiii
Abstract	xv
1. INTRODUCCIÓN	1
1.1. Marco y Contexto de la tesis	1
1.2. Objetivos.....	2
1.3. Principales aportaciones	3
1.4. Estructura de la tesis	4
2. ESTADO DEL ARTE.....	5
2.1. Robots Autónomos	5
2.2. Inteligencia Artificial.....	7
2.3. Planificación automática.....	8
2.3.1. Planificación clásica.....	8
2.3.2. STRIPS.....	9
2.3.3. Planificadores y el espacio de planes	10
2.3.4. Graphplan	10
2.3.5. Planificación HTN.....	11
2.3.6. Planificación basada en SAT.....	11
2.3.7. Planificación con heurísticas	12
2.3.8. Madagascar.....	12
2.3.9. Fast Downward	13
2.3.10. Portfolios	13
2.4. Planificación con SAT.....	13
2.4.1. Problema SAT	13
2.4.2. SAT en planificación.....	15
2.4.2. Codificación SAT.....	16

2.4.3. Tipos de codificación de acciones.....	21
2.5. PDDL.....	22
2.5.1. Lenguaje PDDL.....	22
2.5.2. Versiones de PDDL.....	24
2.5.3. Editores PDDL	26
2.6. Otras técnicas.....	32
2.6.1. Grafos, cliques y problemas combinatorios.	32
2.6.2. Bitstrings	32
3. ENTORNO PARA PLANIFICACIÓN AUTOMÁTICA	35
3.1. Arquitectura de un agente inteligente para planificación	35
3.2. Analizador de lenguaje PDDL.....	40
3.2.1. Gramática de PDDL	41
3.2.2. Analizador de lenguaje PDDL	43
3.2.3. Analizador de PDDL basado en PEG	47
3.2.4. Estructuras de almacenamiento de PDDLs	48
3.2.5. Exportación e importación a formato JSON	51
3.3. El compilador PDDL-SAT	52
3.3.1. Introducción	52
3.3.2. Esquema del compilador SAT en el contexto del sistema	53
3.3.3. Notación	53
3.3.4. Etapas del compilador SAT.....	56
3.4. Estudio de incompatibilidad entre acciones	67
3.4.1. Tipos de incompatibilidad entre acciones	67
3.4.2. Clasificación de incompatibilidad entre acciones	69
3.5. Codificación híbrida de acciones.....	74
3.5.1. Ejemplo	75
3.5.2. Resultados	79
3.6. SAT Solver	80
3.7. Analizador de soluciones de planificación	82
4. ENTORNO DE SIMULACIÓN	85
4.1. Introducción.....	85

4.2. Simulador de escenarios 3D	86
4.2.1. Funcionalidades generales.....	86
4.2.2. Integración entre Simulador y otros módulos	87
4.2.3. Simulador - desktop	87
4.2.4. Simulador - Online	90
4.3. Editor Estructurado de PDDL (PDDL Builder)	92
4.3.1. Introducción	92
4.3.2. Funcionalidades del Editor.....	93
4.4. Espacio de usuario en la nube.....	100
5. ESCENARIOS PARA LA SIMULACIÓN Y PRUEBAS	101
5.1. Descripción de escenarios	101
5.1.1. Escenarios ya existentes	101
5.1.2. Problemas de logística aips	102
5.1.3. Escenario A	104
5.1.4. Escenario B	110
5.1.5. Escenario C	113
5.1.5. Escenario con modificadores emocionales	119
6. CONCLUSIONES Y TRABAJOS FUTUROS.....	123
6.1. Conclusiones.....	123
6.2. Trabajos futuros	124
7. REFERENCIAS.....	125

ÍNDICE DE FIGURAS

Fig. 1. Robot urbano.	6
Fig. 2. Expresiones en la "cara" de Urbano.	7
Fig. 3. Diagrama del planificador SAT.....	16
Fig. 4. Diagrama de estados y acciones entre estados.	18
Fig. 5. Editor Notepad++ mostrando un fichero de PDDL.....	27
Fig. 6. Captura del IU de PDDL Studio. Fuente: PDDL Studio web site.....	28
Fig. 7. Editor Sublime mostrando coloreado de sintaxis sobre un dominio en PDDL.	29
Fig. 8. Página principal del sitio Planning.Domains.....	30
Fig. 9. Editor de Planning.Domains.....	30
Fig. 10. Captura de pantalla del editor PLAGUE.	31
Fig. 11. Esquema general de la arquitectura de un agente.....	36
Fig. 12. Esquema de la arquitectura del agente propuesto.....	37
Fig. 13. Esquema de bloques de la arquitectura del agente inteligente.	38
Fig. 14. Diagrama de Componentes del sistema del analizador de PDDL.	41
Fig. 15. Diagrama de Clases del analizador PDDL.	44
Fig. 16. Máquina de estados del reconocedor de tokens (scanner).....	45
Fig. 17. Detalle de la clase PddlParser.....	46
Fig. 18. Diagrama de elementos de dominio en PDDL.	48
Fig. 19. Diagrama de elementos de problema en PDDL.	49
Fig. 20. Diagrama de clases para almacenar un dominio PDDL.	51
Fig. 21. Conversor SAT en el entorno del sistema.	53
Fig. 22. Diagrama de fases en la compilación a SAT.....	56
Fig. 23. Grafo de incompatibilidades G_c del ejemplo.	77
Fig. 24. Diagrama del SAT solver en el contexto del sistema.....	81
Fig. 25. : Diagrama de los módulos relacionados con el análisis y generación de problemas y planes.....	82
Fig. 26. Diagrama de contexto del Simulador.	86

Fig. 27. Captura del interfaz del simulador.....	88
Fig. 28. Captura del interfaz del simulador online.....	90
Fig. 29. Interfaz general del editor PDDL Builder.	93
Fig. 30. Sección de Requirements del PDDL Builder.	94
Fig. 31. Detalle del Interfaz de usuario de PDDL Builder, en la edición de predicados.	95
Fig. 32. Sección de acciones en PDDL Builder.	96
Fig. 33. Diseño del IU de PDDL Builder, edición de precondiciones.....	97
Fig. 34. Sección del árbol de proyecto de PDDL Builder.	99
Fig. 35. Ilustración para ejemplo de escenario A.....	105

ÍNDICE DE TABLAS

Tabla 1. Ejemplo de instanciación de un predicado.	17
Tabla 2. Ejemplo de instanciación de una acción.	17
Tabla 3. Ejemplo de uso de números de etapa.	18
Tabla 4. Ejemplo de uso de codificación para etapas.	19
Tabla 5. Agrupación de módulos por funciones relacionadas.	39
Tabla 6. Tabla de incompatibilidad entre acciones.	69
Tabla 7. Lista de acciones generadas en el dominio del ejemplo.	76
Tabla 8. Codificación bitwise de las acciones del conjunto K1.	78
Tabla 9. Codificación bitwise de las acciones del conjunto K2.	78
Tabla 10. Comparativa de codificación de varios problemas de planificación y reducción del número de bits necesarios empleando la codificación híbrida propuesta.	80

RESUMEN

La planificación automática es un área dentro de la inteligencia artificial que está en pleno desarrollo.

En esta tesis se muestra el trabajo realizado relacionado con la planificación automática, planteando un entorno de desarrollo para problemas de planificación que facilite su uso, integrando, en un sistema, herramientas de planificación, simulación y de visualización 3D, además de una utilidad para la edición de PDDL (lenguaje empleado en planificación). El sistema planteado incluye además la posibilidad de análisis de resultados de planificación, que pueden usarse en un nivel más alto de razonamiento junto con bases de conocimiento.

Asimismo se expone un planificador basado en lógica booleana, incluyendo un traductor de problemas de planificación a problemas de lógica, que pueden ser resueltos por un demostrador lógico. Este traductor incluye aportaciones en la técnica de conversión y codificación de esos problemas.

Se aportan además un conjunto de problemas de planificación, de distinta complejidad, aplicados a la robótica, sirviendo como ejemplos para posteriores desarrollos.

ABSTRACT

Automated planning is a part of Artificial Intelligence on current development.

This thesis exhibits the work done in relation with automated planning, defining a framework for automated planning that makes developping planning problems easier by integrating tools for planning, simulation, and 3D viewing. The framework also includes a PDDL (planning domain definition language) editor that facilitates writing planning problems. Moreover, the system is also capable of analyzing planning results, which can be used on a higher reasoning level together with an adequate knowledge base.

In addition, a planning solver based on Boolean logic is described and implemented. The solver contains a compiler from PDDL to a logic problem, solvable by a logic demonstrator program. This compiler has a number of improvements with respect to state-of-the-art compilers.

Finally, a set of original planning problems concerning robotics and of varying degrees of difficulty are presented. We hope these benchmarks can serve as useful examples for the IA planning community.

1. INTRODUCCIÓN

1.1. Marco y Contexto de la tesis

Este trabajo se presenta como tesis doctoral dentro del Programa de Doctorado en Automática y Robótica en la ETSII de la UPM (con Mención Excelencia) y has sido realizado formando parte del Grupo de Control Inteligente y del Centro de Robótica y Automática del CSIC. Dentro del ámbito del control inteligente se emplean diversas técnicas de inteligencia artificial, en nuestro caso aplicada a robots principalmente, pero también a otras facetas. En este trabajo concretamente, se profundiza en planificación automática, así como en el uso de problemas de satisfacción de restricciones lógicas para obtener planes de actuación. Dentro de la diversidad en algoritmos, técnicas y lenguajes para expresar los dominios de planificación, se han escogido algunos de los que representan un aspecto importante entre los investigadores del tema.

La tesis se ha realizado participando en los siguientes proyectos de investigación del Plan Nacional:

- Un robot interactivo autónomo capaz de razonar en un entorno dinámico (ARABOT, Autonomous RAtional RoBOT). DPI2010-21247-C02-01.
- Navegación Asistida mediante Lenguaje Natural (NAVEGASE) 2014-2017, DPI 2014-53525-C3-1-R.

1.2. Objetivos

El objetivo de esta tesis es explorar y desarrollar un entorno de planificación automática, que proporcione un control de alto nivel para un agente inteligente y así le permita llevar a cabo una misión descrita mediante una lista de tareas simbólicas: en cada instante el agente será capaz de deducir las mejores acciones para realizar la misión encomendada. Estas acciones se declaran de manera implícita en una base de conocimiento del dominio. Para ello, debe tratarse de un sistema modular que permita conectar con una base de conocimiento.

Se plantea que el robot sea capaz de detectar problemas en la realización de un plan y re-planificar en consecuencia. Esto significa analizar el plan trazado antes de su realización, evaluar la ejecución del mismo para detectar inconsistencias o discrepancias con el plan teórico, y poder re-planificar.

Dada la importancia del proceso de validación, se ha introducido un objetivo adicional: disponer de un simulador de planes, incluyendo un entorno gráfico para poder visualizar entornos y analizar los planes a realizar.

El planificador tiene carácter general, pudiendo establecer el dominio mediante un lenguaje estándar (PDDL). De esta forma, tanto el planificador como el simulador de planes pueden emplearse en distinto ámbitos, más allá de la robótica, que puedan modelarse en PDDL, como por ejemplo la industria, logística o informática.

El planificador que se planteó está basado en lógica booleana, por lo que también constituyó un objetivo de la tesis explorar las características de la planificación basada en SAT (*satisfiability problem* en inglés, y de ahora en adelante, SAT-planning) y desarrollar un compilador para permitir la resolución de los planes usando SAT.

Todo ello se enmarca en una propuesta de arquitectura de agentes inteligentes, que incluye planificador, gestor de conocimiento, análisis de planes, aprendizaje y otras características.

En resumen, los objetivos concretos han sido:

- Plantear una arquitectura de agente inteligente basado en planificación
- Desarrollar un analizador del lenguaje PDDL
- Desarrollar un compilador de PDDL a SAT, y su correspondiente decodificador.
- Desarrollar un entorno de simulación, pruebas y modelador para problemas en PDDL.
- Crear una serie de escenarios de prueba con problemas de planificación relacionados con robots y sus operaciones.

1.3. Principales aportaciones

En el trabajo que se expone, se ha desarrollado un entorno, o marco de trabajo (*framework*) para planificación automática y simulación de los planes solución obtenidos, denominado LPS (*Logic Planning Simulator*). Este *framework* facilita el modelado PDDL, y supone una herramienta para planificación que es una solución completa (modelado, obtención de planes y simulación de los mismos). Por otra parte, tiene un diseño modular y permite sustituir sus módulos por otros con similares funciones pero distinta implementación, sujeto a respetar las interfaces entre módulos. Asimismo, deja abierta la posibilidad de análisis de resultados en la planificación y permite la conexión con otros módulos adicionales. Por último, cabe destacar que los módulos están claramente documentados, lo que permite su estudio y facilita su modificación. Este entorno fue presentado en un congreso en 2015 [62] .

Dentro del *framework* se incluye un analizador del lenguaje PDDL (versión 1.2), programado en C++. Es un código general que puede ser reutilizado en cualquier otro proyecto que implique ficheros PDDL. El módulo con su código fuente y comentarios queda a disposición de los interesados.

Para el tipo de planificación proporcionado (*SAT planning*), se aportan dos técnicas para mejorar el rendimiento bien del compilador SAT, bien de la obtención de la solución. Éstas son:

- El uso de cadenas de bits (o *bitarrays*), para aprovechar las características de la CPU tanto en el solver SAT como en la codificación. Una aplicación de esta técnica fue expuestas en el congreso ICTAI 2008 de la IEEE [58].
- Un método de codificación binario híbrido, basado en la compactación de bits, que no pierde la posibilidad de representar la concurrencia de acciones. Este método se presentó como ponencia en el congreso MMCTSE en 2017. Se inspira en algoritmos para problemas combinatorios que se expusieron en *Computers & Operations Research* en 2014 [57].

En cuanto al simulador, se diseñaron sus funcionalidades, estableciendo sus especificaciones. Como resultado se dispone de un simulador con visualización en 3D, para entornos PC, y otro simulador de similares características accesible desde un navegador web. Además se ha desarrollado un editor (llamado PDDL Builder) con un interfaz de usuario específicamente diseñado para facilitar el modelado y escritura de los dominios en PDDL. Este editor funciona sobre un navegador web.

Finalmente, se han creado un conjunto de escenarios ambientados en temáticas de robots para ser empleados como ejemplos de dominios y problemas en PDDL, y como escenarios 3D para el simulador.

1.4. Estructura de la tesis

En este capítulo se ha mostrado una introducción al trabajo presentado.

En el Capítulo 2 se comenta el estado del arte relativo a planificación automática en general, y a SAT planning en particular.

El Capítulo 3 se centra en una descripción general del entorno desarrollado para planificación, y describe con mayor detalle el analizador del lenguaje y el compilador a problemas SAT. Asimismo, se describen algunas técnicas que se recogen como aportaciones.

Posteriormente, el Capítulo 4 describe el entorno de simulación, tanto en modo paso a paso como en el visualizador 3D de escenarios, así como PDDL Builder, el entorno de edición de PDDL desarrollado.

En el Capítulo 5 se comentan varios escenarios creados para la simulación y pruebas y sus características.

Finalmente, las conclusiones de este trabajo se encuentran en el Capítulo 6, junto con trabajos y posibles líneas de investigación futuras.

2. ESTADO DEL ARTE

Uno de los grandes aspectos en la construcción de robots autónomos es la inteligencia artificial y sus aplicaciones. Si se pretende que un robot (o un agente inteligente) exhiba un cierto grado de autonomía, se precisa aportar la capacidad de decisión respecto a las acciones que realizará el robot dentro de su entorno. Un aspecto importante es la capacidad de planificación de series de acciones para lograr un objetivo o misión.

Otro aspecto importante a considerar son las herramientas disponibles para el desarrollo de esas capacidades e incorporarlas en el agente inteligente, incluyendo lenguajes, programas, técnicas y entornos de desarrollo y simulación dedicados.

2.1. Robots Autónomos

La robótica, como rama tecnológica que se ocupa del diseño, construcción, programación y aplicación de robots, ha tenido un gran desarrollo. Los robots que se han creado, inicialmente limitados en cuanto a las decisiones que podían tomar al realizar las tareas programadas, han ido aumentando en capacidad de percepción del entorno en el que se desenvuelven, y en capacidad de proceso de dicha información, aumentando por tanto su flexibilidad, y su capacidad de adaptarse a condiciones diversas. Se les ha ido dotando de capacidad de decisión y por tanto, de autonomía. Para ello el control del robot ha ido incorporando cada vez más técnicas provenientes de la inteligencia artificial.

2. ESTADO DEL ARTE

En nuestro grupo de investigación tomamos como modelo el robot Urbano [55]: un robot de servicio, dedicado a servir de guía en exposiciones y museos. Su foto se muestra en la Fig. 1. Para realizar sus funciones, Urbano debe ser capaz de realizar varias funciones que lo clasifican como un robot con alto nivel de autonomía:

- Obtiene información de su entorno a través de distintos tipos de sensores (cámara, láser, ultrasonidos).
- Puede realizar sus tareas durante gran parte del tiempo sin necesidad de un operador humano.
- Puede moverse, evitando obstáculos próximos, incluso móviles, y utilizando un mapa de su entorno, acceder a distintas zonas. Puede conocer su ubicación relativa, para guiar a grupos de personas.

Fig. 1. Robot urbano.

Además de esto, Urbano dispone de un módulo de comportamiento emocional [1]. Esto es muy importante en robots que mantienen una interacción con humanos como parte de sus tareas. Urbano dispone de una “cara” robótica que puede expresar emociones, como se muestra en la Fig. 2.

Fig. 2. Expresiones en la "cara" de Urbano.

Urbano tiene una arquitectura software que ha servido de inspiración para el trabajo realizado en esta tesis, y así dotarle de mayor capacidad de planificación que aumente su grado de autonomía.

2.2. Inteligencia Artificial

La Real Academia Española [71] de la lengua define "Inteligencia artificial" como "Disciplina científica que se ocupa de crear programas informáticos que ejecutan operaciones comparables a las que realiza la mente humana, como el aprendizaje o el razonamiento lógico". Es bastante exacto, aunque puede precisarse que la "inteligencia artificial" es más bien la inteligencia exhibida por máquinas en general, y en particular la "inteligencia computacional" la inteligencia exhibida por programas informáticos.

Suele citarse a McCarthy 1956 como inventor del término "*Artificial Intelligence*" (Inteligencia artificial) o AI (IA) [56] pág. 17.

Actualmente entre los investigadores en inteligencia artificial se habla de Agentes Inteligentes (ya no se suele indicar que es artificial), puesto que es a través de éstos como se manifiesta la inteligencia, y engloba tanto las técnicas clásicas de IA como consideraciones acerca del entorno en el que se encuentra y su interacción.

Se han propuesto diversas arquitecturas de agentes inteligentes, para una introducción general véase [56], en el caso multiagente ver [70], y cabe citar [37], [12], [48] entre otras referencias, dentro de las cuales se indican el almacenamiento de la información y el conocimiento, el razonamiento automático, aprendizaje, e incluso emociones y autoconciencia.

Uno de los procesos o partes que aportan inteligencia a un agente es la planificación.

2.3. Planificación automática

La Real Academia Española [71] de la lengua define "Planificación" como

1. Acción y efecto de planificar
2. Plan general, metódicamente organizado y frecuentemente de gran amplitud, para obtener un objetivo determinado, tal como el desarrollo armónico de una ciudad, el desarrollo económico, la investigación científica, el funcionamiento de una industria, etc.

En este texto nos referimos al proceso de Planificación Automática, con un significado más restringido. Así, el llamado **problema de planificación** consiste en encontrar una secuencia de **acciones** que, al realizarlas, permitan pasar de una situación **actual** a lograr un **objetivo** o meta, es decir, un resultado deseado.

En el inicio del libro de Ghallab et al. [21], se define: “*Planning is the reasoning side of acting. It is an abstract, explicit deliberation process that chooses and organizes actions by anticipating their expected outcomes. This deliberation aims at achieving as best as possible some pre stated objectives. Automated planning is an area of Artificial intelligence (AI) that studies this deliberation process computationally*”.

Las aplicaciones de la planificación automática son muy amplias, abarcando la logística, montaje de maquinaria compleja, proyectos con muchas etapas, etc. Por supuesto también como parte de un agente inteligente.

Dentro del campo de la planificación automática encontramos distintas técnicas y métodos. Inicialmente se acotó el problema, imponiendo una serie de restricciones de forma que fuese abordable. Eso constituyó la planificación clásica, como se verá a continuación.

Posteriormente se fueron relajando esas restricciones, es decir, permitiendo ciertas situaciones que se encuentran en los problemas reales y ampliando las aplicaciones de la planificación automática. Algunas de esas mejoras son el uso de restricciones temporales, planificación interactiva, uso de acciones con efectos estocásticos, etc.

Se considera que los primeros programas para la resolución de problemas de planificación en general aparecieron alrededor de 1960, junto con lenguajes para expresar formalmente el modelo de planificación. A partir de entonces han aparecido diversos programas y técnicas, como se comenta a continuación.

2.3.1. Planificación clásica

Pueden nombrarse como los primeros planificadores a GPS (*General Problem Solver*) [42], [43]. Los primeros planificadores se basaban en demostración de teoremas y lógica de predicados.

Para abordar estos problemas, se establecen una serie de restricciones o suposiciones al modelo, de forma que la amplia variedad de problemas de planificación queda acotada. Dichas suposiciones son:

- Mundo Finito (A0): En el modelo está descrito todo lo que es de importancia para la planificación, y tiene un número finito de estados.
- Completamente observable (A1): El agente tiene información completa de todo el escenario del problema.
- Mundo determinista (A2): No existen efectos estocásticos, es decir, el agente al actuar no experimenta ningún tipo de azar. Las acciones aplicadas siempre tienen el efecto esperado.
- Mundo estático (A3): No hay ningún agente en el mundo que pueda cambiar el estado de éste, excepto el agente que realiza el plan.
- Objetivos simples (sin restricciones) (A4): La meta que se desea alcanzar se establece, sin ninguna consideración de estados intermedios.
- Planes secuenciales (A5): El agente realiza una sola acción cada vez, una tras otra hasta alcanzar el objetivo.
- Tiempo implícito (no hay duración) (A6): No se tiene en cuenta el tiempo que tarda en realizarse una determinada acción. Simplemente existe una secuenciación de una acción tras otra.
- Planificación *offline* (A7): Se establecen 2 fases, una de planificación, en la que el agente delibera y obtiene un plan válido, y una segunda fase en la que el agente realiza las acciones obtenidas en el plan. No se intercalan ambos procesos.

El código entre paréntesis corresponde a la clasificación expuesta en Ghallab et al. [21], págs. 9 a 13, como modelos restringidos (planificación clásica) y extendidos.

Aunque puede parecer que estas restricciones dan lugar a escenarios excesivamente rígidos, es un buen punto de partida y además, algunos problemas pueden adaptarse a estas restricciones obteniendo soluciones válidas. Algunas de estas restricciones pueden relajarse con bastante facilidad, ampliando el tipo de problemas que pueden resolverse con un planificador.

2.3.2. STRIPS

STRIPS es un planificador desarrollado por Richard Fikes and Nils Nilsson en 1971 [16]. Estaba realizado con la finalidad de resolver problemas de planificación. También se denomina así al lenguaje que emplea para definir el problema de planificación concreto.

La representación STRIPS estaba basada en lógica de primer orden. El estado se compone de un conjunto de predicados, los cuales pueden tener variables, y las acciones establecen los cambios en el estado del mundo, qué condiciones

debe cumplir una acción para poder realizarla (precondiciones) y qué cambios produce sobre el mundo (efectos).

Un problema de planificación en STRIPS es un sistema basado en estados y se define como una tupla $P = \langle I, G, A \rangle$, donde I es un conjunto de hechos (predicados) que se cumplen en el estado inicial, G es el objetivo (conjunto de hechos que deben cumplirse), y A es un conjunto de acciones. Cada acción A tiene unas precondiciones $pre(A)$, y unos efectos, que pueden ser añadir hechos al estado, o quitarlos, y se indican por $add(A)$, y $del(A)$ respectivamente.

Se demostró que resolver la planificación en un dominio STRIPS es un problema PSPACE-completo [8].

Tras STRIPS aparecieron varios planificadores que mejoraban o ampliaban las características de STRIPS, como ADL.

ADL es también un programa para planificación automática, concebido para su uso en robótica. Fue propuesto en 1987 Pednault [44]. Se basa en STRIPS, con algunos cambios: permite efectos condicionales, y se basa en el principio de mundo abierto, es decir: el estado del mundo no es necesariamente conocido a priori y permite la existencia de incertidumbre en los predicados.

2.3.3. Planificadores y el espacio de planes

Si bien los planificadores anteriores producían una solución como una secuencia perfectamente ordenada de acciones para llegar al objetivo, los planificadores de orden parcial (*partial-order planning*) obtienen un plan en el cual algunas de las acciones no están ordenadas, cuando el orden de éstas no importa. También se denominan planificadores de "menor compromiso" [68].

Algunos planificadores de este tipo son UCPOP, SNLP y UNPOP.

2.3.4. Graphplan

Graphplan es un programa para planificación automática, cuya característica principal era la construcción de un grafo con acciones y predicados para obtener el plan. En el grafo se tienen en cuenta los predicados incompatibles, para reducir el espacio de búsqueda. Fue desarrollado por Avrim Blum y Merrick Furst en 1995 [5]. Posteriormente fue ampliado para permitir planificación con probabilidades [6].

Partiendo de un modelo de problema en lenguaje STRIPS, Graphplan calcula una secuencia de acciones para alcanzar el estado objetivo, si eso es posible. La novedad de Graphplan consistía en construir un grafo (*planning graph*) para reducir el número de posibilidades en la búsqueda de la solución.

El grafo se construye por capas de nodos. En la primera capa se crea un nodo por cada predicado que se cumple en el estado inicial. En la siguiente capa se

crean nodos para representar acciones posibles, realizables en el primer paso del plan. Los predicados de una etapa y las acciones posibles se van disponiendo en capas alternas. Los enlaces en el grafo indican:

- Relación entre una acción y un predicado, en la capa anterior, que aparece como precondition en la acción.
- Relación entre una acción y el predicado que queda afectado por un efecto de la acción.

También se crean unas listas para anotar predicados que no pueden ser ciertos al mismo tiempo (incompatibles), y de las acciones que son incompatibles.

El grafo, una vez construido, representa sucesivas etapas en el plan, hasta una última capa en la que las condiciones del objetivo se cumplen. De este grafo puede obtenerse, mediante un algoritmo de búsqueda hacia atrás, la secuencia de acciones que constituyen el plan para lograr el objetivo.

Actualmente la idea del *planning graph* se usa en algunos otros planificadores como un pre-procesado que permite reducir el número de acciones posibles en cada etapa.

2.3.5. Planificación HTN

Hierarchical Task Network (HTN) o Red Jerárquica de Tareas es una técnica para planificación automática en la que se describen las tareas que puede realizar el sistema y su descomposición en sub-tareas, a varios niveles hasta lograr primitivas que pueden ser ejecutadas directamente. El planificador emplea esa descomposición para obtener un plan a partir de tareas de alto nivel. Existen dependencias entre tareas y sus sub-tareas, expresadas en forma de red (*network*). La descomposición de tareas en sub-tareas al planificar sigue unas condiciones definidas en la jerarquía de tareas. El formalismo más extendido para expresar la jerarquía de tareas emplea TAEMS [18].

Algunos planificadores de este tipo son O-PLAN, UMCP, ABSTRIPS, SHOP, SIADEX.

2.3.6. Planificación basada en SAT

En lógica, SAT (*Satisfiability* o *satisfiability problem*) se refiere al problema de asignación de valores de verdad en una expresión lógica proposicional de forma que se cumpla (satisfaga) la expresión resultando el valor “verdadero”. Los programas que pueden resolver estos problemas se suelen denominar *SAT solvers*.

Una forma de resolver un problema de planificación es convertirlo en un problema SAT equivalente, cuya solución, mediante un SAT solver se descodifica posteriormente para obtener un plan de acciones válido.

Esta técnica ha tenido mucho éxito debido a la existencia de SAT solvers muy eficientes, y a la existencia de buenas técnicas de conversión entre el lenguaje de formalización de problemas de planificación y el formato usado en SAT.

En este trabajo se emplea este enfoque para el planificador propuesto, por lo que comentará con más detalle en el apartado 2.4.

2.3.7. Planificación con heurísticas

Se denominan planificadores heurísticos (o que usan planificación con heurísticas) a aquellos planificadores cuya búsqueda de la solución se centra en el empleo de algún tipo de función heurística como guía. La función debe ser independiente del dominio, si se desea un planificador de propósito general. Algunos planificadores de este tipo usan un algoritmo en el que se relajan ciertos aspectos en el dominio, produciendo un problema de menor complejidad, por ejemplo considerando sólo una parte de los efectos de las acciones [39].

Estos planificadores supusieron avances frente a Graphplan, siendo más rápidos y permitiendo problemas de mayor tamaño. Algunos planificadores de este tipo son HSP [7] o FF [28].

2.3.8. Madagascar

Jussi Rintanen desarrolló Madagascar [49]. Se trata de un planificador basado en SAT, incorporando algunas técnicas importantes indicadas en [67], [51], [53], entre otras referencias.

Una de las propuestas en Madagascar es relajar las condiciones en las que las acciones son incompatibles y deben realizarse una tras otra, permitiendo considerar que varias acciones se realizan en una misma etapa si existe alguna manera de secuenciarlas de forma que sean compatibles. Esto permite obtener, en muchos casos, planes con un horizonte de tiempo menor, mejorando el rendimiento.

También determina que el horizonte de tiempo en SAT planning es muy importante para el rendimiento, y propone un método para lanzar en paralelo varias instancias de problemas SAT, con distintos horizontes de tiempo, comparando el resultado. Además, expone el uso de horizontes de tiempo progresivamente mayores, en vez de simples incrementos unitarios.

Variantes de Madagascar se clasificaron en los primeros puestos en la competición de planificación de ICAPS en 2014.

2.3.9. Fast Downward

Fast Downward es un planificador desarrollado inicialmente por Malte Helmert [23]. Puede considerarse como un planificador de tipo heurístico, pero se basa en la conversión del problema, inicialmente expresado en PDDL, al formalismo SAS+ [4].

Este planificador construye grafos causales, y convierte conjuntos de predicados relacionados en variables, que toman como valores distintos objetos del dominio. La descripción de su funcionamiento puede encontrarse en [24],[26].

El formalismo SAS+ es también reducible a SAT, según muestra Huang [30], [29], [31].

2.3.10. Portfolios

Uno de los enfoques de mayor éxito en las últimas competiciones de planificación automática es el uso en paralelo de varios planificadores, con distintas características y algoritmos, ponderando el tiempo de uso de procesador para cada uno, a fin de conseguir el mejor rendimiento posible para distintos tipos de problemas de planificación. La justificación se basa en que distintos planificadores tienen cada uno fortalezas y debilidades según la clase de problema que debe resolver. Ejemplos de estos "planificadores portfolio" pueden consultarse en [65] y [9].

2.4. Planificación con SAT

En los años 90 (del siglo pasado), se describe por primera vez un método para resolver problemas de planificación automática basado en convertir el problema concreto de planificación a una instancia del problema de satisfacibilidad booleana (también llamado SAT). La solución del problema SAT, mediante cualquier programa (*solvers*) para SAT, se decodifica posteriormente como un plan solución del problema de planificación. Dado que en esa época ya existían SAT solvers muy eficientes, resultó un método muy aceptado.

2.4.1. Problema SAT

En lógica booleana, se denomina "*Satisfiability Problem*" (o SAT, para abreviar) al problema de determinar si existe (o no) una asignación de valores de verdad (verdadero o falso) a las variables o términos de una fórmula de la lógica proposicional, de forma que dicha fórmula tome el valor "verdadero". En caso afirmativo, suele ser también útil conocer esa asignación de variables (al menos una, de entre muchas soluciones posibles). Se denomina modelo a una

asignación de valores tal que se cumpla la expresión. En caso de no encontrar tal asignación, puede ayudar disponer de alguna pista acerca de las razones por las cuales no es posible tal asignación; es decir, saber qué parte de la expresión lógica provoca una contradicción y qué variables lógicas están implicadas en dicha contradicción (suele suceder que sólo algunas de las variables son las que generan la imposibilidad de satisfacción de la expresión). El problema SAT pertenece a la categoría NP-Completo [10].

Un ejemplo sería la expresión:

$$(p \vee q) \wedge (r \vee \neg s)$$

Siendo T= Verdadero y F=Falso, las siguientes asignaciones son modelos:

$$\{p = F, q = T, r = T, s = F\}$$

$$\{p = T, q = F, r = F, s = F\}$$

Sin embargo la siguiente asignación no es un modelo, puesto que el resultado de la expresión es Falso:

$$\{p = F, q = F, r = F, s = T\}$$

Un ejemplo de fórmula que no puede ser Verdadero nunca es:

$$p \wedge \neg p$$

Es frecuente en SAT especificar la expresión lógica a satisfacer en el formato normalizado CNF (Conjunctive Normal Form) [46], que consiste en una conjunción de clausulas. Se denomina clausula a una disyunción de literales o variables lógicas, bien de forma negada o bien sin negar (forma directa).

Existen varios algoritmos y técnicas para resolver problemas SAT [66],[40],[14],[3]: los más usados para su resolución exacta son DPLL [11] y sus variaciones modernas, o bien algoritmos heurísticos basados en búsquedas aleatorias como WalkSAT [72], [59] o inspirados en algoritmos genéticos como MASK [22].

DPLL obtiene su nombre de sus autores (Davis-Putnam-Logemann-Loveland). Es un algoritmo completo (es decir, siempre encuentra la solución, si existe) basado en búsqueda exhaustiva con backtracking. En versiones posteriores se han añadido técnicas que mejoran el rendimiento, como backtracking no cronológico, aprendizaje de cláusulas, propagación con "*two watched literals*", o reinicios aleatorios.

La salida de un SAT solver es una etiqueta, indicando si existe al menos una solución que satisfaga la expresión y además se espera un modelo (una serie de variables con la asignación de valores de verdad, que satisface la expresión; nótese que dicha asignación no tiene porqué ser necesariamente completa). Alternativamente, se obtendrá una etiqueta indicando que no existe ninguna asignación posible que cumpla; se dice entonces que el problema es

insatisfacible (o no satisfacible). Algunos SAT solvers pueden dar una explicación acerca de por qué no se puede satisfacer la fórmula, proporcionando por ejemplo un conjunto de variables que son las que provocan un conflicto y no puede satisfacerse la fórmula.

El problema SAT tiene un gran interés para los informáticos, ya que la lógica booleana forma parte de los principios de funcionamiento de los ordenadores, y su uso es básico en los lenguajes de programación. Muchos problemas algorítmicos pueden traducirse a un problema lógico SAT.

2.4.2. SAT en planificación

En nuestro caso, el interés de SAT proviene de su posible uso para codificar y resolver problemas de planificación. Se denominan SAT planners a los planificadores que realizan una traducción a SAT para resolver el problema de planificación, véase [21] capítulo 7, y [50]. Dadas las características de la codificación a SAT, estos solvers suelen tener algunas limitaciones respecto a los tipos de problemas que pueden resolver, pero este esquema ha generado planificadores muy rápidos.

Los trabajos de Kautz y Selman [32], [33] se consideran los primeros SAT planners. Posteriormente se desarrolló Blackbox [34],[35], que incorporaba más técnicas.

Un asunto muy importante en un planificador basado en SAT es la conversión del problema de planificación (expresado por ejemplo en PDDL) a un problema SAT. Existen varios métodos (o técnicas de conversión) de un problema de planificación a SAT. En los comienzos, algunos autores realizaron conversiones manuales a SAT, pero esto sólo es práctico con problemas muy sencillos. Actualmente se usan conversiones automáticas de PDDL a SAT, siguiendo unos esquemas de traducción fijos [15]. Un programa o procedimiento que realiza esta conversión se denomina compilador SAT (*SAT compiler*). En la Fig. 3 se muestra un diagrama de un SAT solver.

El compilador SAT parte de la descripción del problema de planificación, es decir, tendrá en cuenta el estado inicial, el conjunto de acciones con sus efectos, la expresión del objetivo o meta y un horizonte de tiempo (número de etapas o pasos). Con ello construye una fórmula lógica de manera que la fórmula sea satisfecha si y sólo si hay un plan válido asociado con ese horizonte de tiempo. Puede entonces encontrarse un plan probando con diferentes horizontes, de forma incremental.

Al pasar el problema de planificación a SAT, es necesario expresar el número de pasos o etapas en las que se realiza el plan. En SAT planning se utilizan distintas variables lógicas para expresar el estado de cada predicado o acción en cada etapa. Suele prepararse una conversión a SAT de un número fijo de etapas (*steps*), y comprobar si existe solución con ese número de etapas. Si se

comienza con 1 etapa, y se incrementa en 1 cada vez, se obtendrá un plan con el menor número de etapas posible, N. Los planes con menos etapas se habrán probado no satisfacibles. Esto significa que SAT planner debe convertir e intentar resolver un SAT varias veces, antes de encontrar una solución.

Fig. 3. Diagrama del planificador SAT.

Después de encontrar una solución, un SAT planner debe convertir la solución SAT a la solución en planificación equivalente. Esta traducción suele ser sencilla, buscando en una simple tabla inversa, generada a la vez que se realizó la conversión a SAT.

2.4.2. Codificación SAT

Al emplear planificación basada en SAT, un problema de planificación automática clásica se expresa en una fórmula de lógica proposicional, de forma que los "modelos" o soluciones de esta fórmula se corresponden con una solución del problema de planificación y constituyen por tanto un plan válido de ese problema. Un modelo es una asignación de valores de verdad a las variables de la fórmula lógica de forma que el valor de verdad de la fórmula es verdadero. El problema de satisfacibilidad es el problema de encontrar si una

fórmula tiene algún modelo, y en ese caso decimos que es satisfacible. La asignación de valores en el modelo se corresponderá con un plan válido.

Los dos elementos principales para convertir un problema de planificación a una expresión lógica son los estados y las transiciones entre estados.

Estados

Los estados en el problema de planificación suelen expresarse (en PDDL es así) como un conjunto de predicados de lógica de primer orden, aplicables a objetos definidos en el problema. Para expresarlos en SAT, se deben convertir a lógica de predicados aplicando un proceso de instanciación de objetos en los predicados (*grounding*), es decir, sustituir los parámetros presentes en los predicados, por todas las posibles opciones, como se ve en el ejemplo mostrado en la Tabla 1.

Predicado	Objetos	Proposiciones
ROBOT-AT(room)	room1 room2	robot-at_room1 robot-at_room2

Tabla 1. Ejemplo de instanciación de un predicado.

Transiciones

En un modelo determinista y estático, como sucede en planificación clásica, el agente provoca cambios en el estado del mundo a través de sus acciones. Es además el único modo de cambiar el estado, es decir, no existen otros agentes que afecten al estado.

En el problema de planificación, las acciones se describen mediante unas precondiciones y unos efectos, que son expresiones lógicas de primer orden. Estas acciones se deben codificar como expresiones lógicas proposicionales también, de forma que cada acción con sus argumentos instanciados, será una variable lógica, como se muestra en el ejemplo de la Tabla 2.

Acción	Objetos	Proposiciones
MOVE (from, to) PRE: ROBOT-AT(from) EFF: ROBOT-AT(to) AND -ROBOT-AT(from)	room1 room2	move_room1_room1 move_room1_room2 move_room2_room1 move_room2_room2

Tabla 2. Ejemplo de instanciación de una acción.

Etapas o longitud del plan

El estado del mundo va a ir cambiando, etapa a etapa, durante la aplicación de un plan. Ese estado del mundo está constituido por el conjunto de predicados definidos en el problema, por tanto se debe diferenciar o indicar de alguna forma el estado o instante al que se refiere cada una de las proposiciones. Las acciones también deben estar referidas a una etapa o instante de tiempo concreto. Para esto, se añade un argumento a cada uno de los predicados, que indica el número de etapa en el que se aplica. En el caso de las acciones, se añade también un argumento que indica el número de etapa I en el que se comienza la acción, provocando unos efectos que se aplican al estado de la etapa $I+1$. En el paso a proposiciones, se tendrá una variable lógica distinta para cada etapa, y para cada proposición. En la Tabla 3 se muestra con un subíndice el número de etapa:

$t = 1$	$t = 2$	$t = \dots$
robot-at_room1 ₁	robot-at_room1 ₂	...
robot-at_room2 ₁	robot-at_room2 ₂	...

Tabla 3. Ejemplo de uso de números de etapa.

En planificación clásica los instantes de tiempo se expresan como una secuencia de números enteros, comenzando en 0 (estado o etapa inicial). Se considera que las acciones tienen una duración real irrelevante, pero sí es importante el orden en que se efectúan. Algunos autores usan números pares para indicar los distintos estados, y números impares para los instantes en los que se aplican acciones, de forma sucesiva. En nuestro caso usamos una numeración consecutiva para los estados, como se muestra en la Fig. 1, de forma que el estado inicial es el 0, y las acciones toman el número del instante I en el que se producen, dando lugar a un estado $I+1$. El estado final, en un horizonte de tiempo N , sería el estado N .

Fig. 4. Diagrama de estados y acciones entre estados.

En nuestro caso empleamos el símbolo especial @ como separador para indicar el número de etapa, como se muestra en la Tabla 4.

t = 0	t = 1	t = ...
robot-at_room1@0	robot-at_room1@1	...
robot-at_room2@0	robot-at_room2@1	...

Tabla 4. Ejemplo de uso de codificación para etapas.

En la construcción de las expresiones lógicas, debe tenerse en cuenta la posible existencia de modelos no deseados (*unintended models*), es decir, situaciones en las que la asignación a las variables lógicas es correcta, pero dan lugar a soluciones de planificación que no son posibles.

Puede usarse acciones nulas (*dummy actions*), que no tienen precondiciones ni efectos. De esta forma pueden realizarse planes cuyas acciones efectivas (no nulas) se realicen en menos etapas de las indicadas en el horizonte. Esta técnica puede ser útil ya que normalmente no se conoce a priori el número de etapas necesario para obtener un plan válido para un problema de planificación.

Fórmulas

En el esquema clásico de codificación a SAT, la fórmula de lógica proposicional que corresponde al problema de planificación se construye como la conjunción (operación AND lógica) de varios grupos de expresiones:

1. Estado inicial: es una expresión que incluye a todo el estado inicial s_0 , y aparecen por tanto todos los predicados, con sus argumentos instanciados. Los predicados que no se cumplen (tiene valor de verdad falso) en estado inicial, aparecen negado (operador NOT).

$$\bigwedge_{p \in s_0} p_0 \wedge \bigwedge_{p \notin s_0} \neg p_0$$

2. El estado objetivo (o conjunto de estados objetivos): se expresa como la conjunción de predicados que deben cumplirse en la etapa N, por estar especificados en el objetivo g . Deben indicarse tanto los predicados en forma directa g^+ como negada g^- . El objetivo es un estado especificado parcialmente, no es necesario que aparezcan todos los predicados si no son parte del objetivo.

$$\bigwedge_{p \in g^+} p_{n0} \wedge \bigwedge_{p \notin g^-} \neg p_n$$

3. Acciones: Cada acción a_i provoca un cambio de estado, debido a sus efectos, produciendo unos cambios en los predicados de la etapa siguiente. Además, sólo pueden producirse si se cumplen sus precondiciones. Se indica el número de etapa con el subíndice i .

$$a_i \Rightarrow \left(\bigwedge_{p \in \text{precond}(a)} p_i \wedge \bigwedge_{e \in \text{effects}(a)} e_{i+1} \right)$$

4. En planificación clásica, lo único que puede producir un cambio en los predicados es la aplicación de una acción, y esto debe establecerse como expresiones lógicas. En este punto pueden usarse dos formulaciones: *classical frame actions* o *explanatory frame axioms*. Expondremos las segundas por resultar más compactas, expresadas con las fórmulas:

$$\neg p_i \wedge p_{i+1} \Rightarrow \left(\bigvee_{a \in A / p_i \in \text{effects}^+(a)} a_i \right) \wedge$$

$$p_i \wedge \neg p_{i+1} \Rightarrow \left(\bigvee_{a \in A / p_i \in \text{effects}^-(a)} a_i \right)$$

5. Exclusión entre acciones. En planificación clásica, en cada etapa sólo puede realizarse una acción (*complete exclusion axiom*). Si se desea permitir más de una acción en cada etapa, en vez de esta expresión deberá añadirse otra que establezca la incompatibilidad entre acciones que no puedan realizarse a la vez (*mutual exclusion*). La exclusión mutua completa sigue la siguiente expresión:

$$a_i, b_i \in A, a_i \neq b_i, \bigwedge_{0 \leq i \leq n} (\neg a_i \vee \neg b_i)$$

Este esquema de traducción y algunas variantes se propusieron en [15]. Un problema en SAT planning es la proliferación de variables y cláusulas a medida que los problemas de planificación son mayores. Varias técnicas efectivas para reducir el generación de las expresiones lógicas se indican en [27], [67], [52]

y [54] entre otros. En [60] se exploran distintas opciones para la traducción a SAT y se comparan sus resultados. Huang propone un conjunto de expresiones diferente [30], [31] pero partiendo del formalismo SAS+.

2.4.3. Tipos de codificación de acciones

Como se ha comentado, los predicados y las acciones son convertidos a variables lógicas (o literales) para SAT. Para realizar ese paso, los predicados y las acciones deben instanciar sus parámetros a todos los distintos objetos posibles establecidos en el problema. Además, deberá añadirse como parámetro la etapa a la que se refiere el literal.

Respecto a las acciones, hay varias formas de codificación, a fin de realizar una notación más compacta (con uso de más o menos variables lógicas). Los 4 tipos fundamentales son:

- *Regular encoding*: una variable booleana para cada acción instanciada. Permite paralelismo múltiple.
- *Single operator splitting*: Para cada acción, descomponer los parámetros, y usar varias variables booleanas para indicar la acción por un lado, los parámetros por otro, de forma que se reducen las variables necesarias para codificar una acción concreta. Permite paralelismo de entre acciones que sean distintas.
- *Overloaded operator splitting*: Codificar cada posible parámetro, de forma que todas las acciones y predicados comparten la misma codificación de parámetros, para reducir el número de variables booleanas necesarias. No permite paralelismo.
- *Bitwise*: Se codifica cada acción instanciada como una combinación de variables booleanas. Es la codificación que requiere menos variables lógicas, pero genera una expresión lógica más complicada. No permite paralelismo.

Además de estos tipos, se han sugerido otros tipos híbridos combinando algunos de estos tipos de codificación para distintos grupos de acciones, o de algunos de sus parámetros. En nuestro trabajo se explora una técnica para ello que se explicará en el capítulo 3..

Una vez codificados los predicados y acciones, aplicado el número de etapa y convertido el problema a expresiones lógicas, suele convertirse toda la expresión a formato CNF (*Conjunctive Normal Form*) [46], que es un formato estandarizado utilizado por la mayoría de los SAT solvers. El formato CNF consiste en reducir la fórmula lógica a una conjunción de cláusulas, siendo una cláusula una disyunción de literales o variables lógicas, que pueden estar negadas o no. Toda fórmula lógica puede expresarse en CNF, aunque algunas pueden ser muy amplias por las características de la fórmula.

2.5. PDDL

PDDL son las siglas de *Planning Domain Definition Language* (Lenguaje de Definición de Dominios para Planificación). Es un intento de estandarizar la definición los problemas de planificación de forma general. La primera versión fue definida en 1998 [38]. Surgió inicialmente a raíz de una competición (*International Planning Competition*) que se celebraron en 1998 y 2000. Establece un formalismo que permite a los programas de planificación disponer de un estándar, y así permitir una comparación y competición entre diversos planificadores y los resultados que obtienen.

Tiene una sintaxis parecida a Lisp, basada en STRIPS y ADL.

STRIPS [16] es un programa de planificación automática de tipo general. El dominio y problema a resolver se definen en unos ficheros de texto externos, empleando un lenguaje de declaración denominado también STRIPS, que ha inspirado a otros lenguajes para el mismo propósito. (Suele clasificarse como un lenguaje de acciones, o de transición de estados (*state transition systems*). Se creó en 1971.

ADL [45] surgió posteriormente, ampliando las posibilidades de STRIPS con tipos, efectos condicionales, precondiciones negadas, etc.

PDDL se ha modificado posteriormente generando varias versiones, en sucesivas convocatorias de competiciones. También existen algunas extensiones y variantes, que generalmente intentan incluir aspectos menos usados en los dominios de planificación. Para adaptarse a distintos tipos de planificadores, algunos elementos del lenguaje PDDL son opcionales, y se incluye una especificación de los requisitos u opciones necesarios por una planificador para poder trabajar con un dominio concreto.

2.5.1. Lenguaje PDDL

El lenguaje define un modelo de la realidad para formalizar un problema de planificación. El lenguaje soporta dos tipos de ficheros: uno con la definición general del modelo del problema (*domain*), y otro con los elementos específicos de un problema concreto (*problem*). Por dominio se entiende el conjunto de tipos de objetos que encontramos, los predicados o funciones, y las acciones posibles en el entorno del problema. El problema en sí define los objetos concretos, la situación inicial (que predicados son ciertos en el instante de comienzo), y el objetivo o *goal* (los predicados que queremos que sean ciertos, y para los que necesitaremos realizar una serie de acciones).

Ejemplo de dominio en PDDL:

```
(define (domain robot_domain_A)
  (:requirements :typing :negative-preconditions)
```

2. ESTADO DEL ARTE

```
(:types room box) ; no case sensitive
(:predicates
  (ROBOT-AT ?r - room) ; robot is in room
  (ROBOT-LOADED) ; BOOLEAN
  (BOX-AT ?b - box ?r - room) ; a box can only be on 1 place
  (PATH ?x ?y - room) ; a path between room and room
  (ROBOT-CARRIES ?b - box) ; robot is carrying this box
)
(:action move :parameters (?x - room ?y - room )
  :precondition ( and ( ROBOT-AT ?x )
    ( PATH ?x ?y ) )
  :effect (and (ROBOT-AT ?y)
    (not (ROBOT-AT ?x))
  )
)
(:action load :parameters (?r - room ?b - box)
  :precondition (and (BOX-AT ?b ?r )
    (ROBOT-AT ?r )
    ( not (ROBOT-LOADED) )
  )
  :effect (and (ROBOT-LOADED)
    (not (BOX-AT ?b ?r ))
    (ROBOT-CARRIES ?b )
  )
)
(:action unload :parameters (?r - room ?b - box)
  :precondition (and (ROBOT-AT ?r )
    (ROBOT-LOADED)
    (ROBOT-CARRIES ?b )
  )
  :effect (and (not (ROBOT-LOADED))
    (BOX-AT ?b ?r )
    (not (ROBOT-CARRIES ?b ))
    (not (ROBOT-LOADED))
  )
)
)
```

Ejemplo de problema de ese dominio:

```
(define (problem robot_problem_A_P01)
  (:domain robot_domain_A)
  (:objects rm1 rm2 rm3 rm4 - room
    bx1 bx2 bx3 bx4 - box)
  (:init
    (ROBOT-AT rm1)
    (BOX-AT bx1 rm1) (BOX-AT bx2 rm2)
    (BOX-AT bx3 rm3) (BOX-AT bx4 rm4) ; 1 --- 2
    (PATH rm1 rm2) (PATH rm1 rm3 ) (PATH rm1 rm4) ; |\ /|
    (PATH rm2 rm1) (PATH rm2 rm3 ) (PATH rm2 rm4) ; | x |
    (PATH rm3 rm1) (PATH rm3 rm2 ) (PATH rm3 rm4) ; | / \ |
    (PATH rm4 rm1) (PATH rm4 rm2 ) (PATH rm4 rm3) ; 3 --- 4
```

```
)  
(:goal  
  (and (ROBOT-AT rm3) (not (ROBOT-AT rm1) )  
        (BOX-AT bx2 rm4)  
  )  
)  
)
```

El resultado de resolver el problema será una secuencia de acciones (expresadas en la misma nomenclatura en que PDDL, con parámetros instanciados) ordenada (puede suceder que algunas de las acciones pueden realizarse al mismo tiempo, al no existir conflicto entre ellas, o que pueden realizarse en un orden distinto) que al ser aplicadas trasladan al sistema del estado inicial al estado objetivo. Puede no ser posible conseguir el objetivo, en ese caso el planificador debe indicarlo.

PDDL es un lenguaje de descripción de un problema de planificación. Es un lenguaje de tipo declarativo, por tanto no indica el modo de resolución de dicho problema (aunque a partir de la versión 3.0 se incluyen *preferences*, que pueden usarse como reglas de control). Puesto que distintos planificadores pueden usar distintos métodos y algoritmos de planificación, y sus correspondientes estructuras de datos, algunos de estos planificadores pueden trasladar la definición del problema de planificación fácilmente, o puede que parte de la descripción requiera una transformación compleja (o incluso que no sea posible adaptar a la estructura de un planificador concreto). Generalmente puede transformarse el propio PDDL a una versión equivalente, pero que permite su uso a planificadores que no soportan ciertas opciones, aunque esto suele suponer un incremento en la longitud de la descripción del problema (por ejemplo, el uso de tipos puede ser convertido a un dominio sin tipos, creando predicados para especificar la restricción de objetos a ciertos conjuntos, e incluir esos predicados en las precondiciones de las acciones, y quitando los tipos de todos los objetos del problema. Pero puede ocasionar una mayor extensión de la descripción del problema, y puede ser menos eficiente a la hora de buscar una solución).

2.5.2. Versiones de PDDL

Desde su creación, PDDL ha ido admitiendo cambios, que suelen coincidir con la celebración de las competiciones IPC, para incluir opciones disponibles en distintos planificadores y adaptarse a los avances en los programas de planificación.

Las versiones más importantes de PDDL son:

- PDDL 1.2 [38]: Versión original, empleada en las competiciones de 1998 y 2000. Establece la sintaxis y semántica del lenguaje.

- PDDL 2.1 [17]: Definido en la tercera competición IPC de 2002. Introduce variables numéricas, métricas y acciones continuas o con duración en el tiempo. Suponen cambios bastante importantes en la definición de problemas de planificación.
- PDDL 2.2 [13]: Empleado en la cuarta competición IPC en 2004. Introduce predicados "derivados" (o calculados a partir de otros predicados). También se definen predicados que toman un valor en un determinado instante de tiempo, no necesariamente en el estado inicial.
- PDDL 3.0 [19, 20]: Se añaden restricciones para especificar trayectorias de estados, y preferencias, que sirven para el control del algoritmo de planificación.
- PDDL 3.1 [25, 36]: Se usó en las competiciones IPC en 2008 y 2011. Introdujo *object-fluents*, variables que pueden tomar valores de cualquier tipo. Suponía un cambio de bastante importancia semántica.

La versión actualmente usada en las competiciones IPC- ICAPS es la 3.1 (año 2016).

Algunas extensiones de PDDL son:

- NDDL (New Domain Definition Language): Definido por la NASA, usado en Europa2, está basado en variables e intervalos, y usa reglas o restricciones entre actividades.
- PDDL+: parte de PDDL 2.1 y fundamentalmente añade procesos (acciones en tiempo continuo, con duración) y eventos (instantáneos, producidos por el entorno) .
- PPDDL (Probabilistic PDDL): Apareció en 2004, a partir de PDDL 2.1, en las competiciones de IPC como lenguaje estándar para soportar efectos probabilísticos, y variables fluents para contabilizar “recompensas” de acciones realizadas y metas intermedias.
- APDL (Abstract Plan Preparation Language) : Apareció en 2006 como añadido a NDDL, incorpora acciones y mantiene un nivel más abstracto que PDDL y NDDL. Ideado para poder realizar análisis formales de los dominios y problemas de planificación, es más restringido que PDDL.
- MA-PDDL (Multi Agent PDDL): Extensión para PDDL 3.1, surgida en 2012, para planificación multiagente. Permite diferencias entre distintos agentes, y distintas acciones según el agente, metas diferentes para los agentes. Tiene en cuenta la concurrencia de acciones y la interacción entre ellas debido a distintos agentes actuando. Se le han añadido posteriormente las opciones de “observabilidad parcial” y efectos probabilísticos.

2.5.3. Editores PDDL

El lenguaje PDDL puede escribirse en archivos de texto plano (por ejemplo notepad o vi), al igual que se puede escribir la mayoría de los lenguajes de programación.

En muchos lenguajes de programación, sobre todo en los lenguajes de tipo genérico y de mayor difusión, hay disponible algún programa para editar dicho lenguaje que añade opciones que facilitan escribir código en ese lenguaje. Dentro de esos editores se pueden distinguir al menos dos categorías:

- Editores de texto plano, con opciones para resaltar partes del código según el lenguaje del que se trate (suelen poder ser configurados y así soportar muchos lenguajes distintos). Por ejemplo Notepad++, Scitex, ...
- Entornos de desarrollo integrados (IDE o Workbench), que además de editar y resaltar partes del código, añaden una serie de herramientas que facilitan las fases de escritura, compilación, pruebas y depuración. Suelen centrarse en un solo lenguaje concreto. Ejemplos: VisualStudio, CodeWarrior, Netbeans, etc.

En el caso de PDDL, no existen muchas opciones específicamente diseñadas, pero sí algunas. En el caso de editores de texto multi-lenguaje se pueden emplear plantillas de realce de sintaxis de Lisp, ya que PDDL sigue una sintaxis parecida. En caso de desear mayor ajuste a PDDL, no suele ser difícil modificar estas plantillas para adaptarlas a PDDL.

Estos editores facilitan la escritura de código de programación, por ejemplo, ayudan a detectar comienzo y fin de paréntesis (muy útil dada la sintaxis de PDDL) u otros delimitadores. Algunos editores pueden colapsar/expandir bloques de código, esto es, partes del código que queda delimitado por paréntesis, y que por tanto va agrupado, como se muestra en la Fig. 5.

Esto es útil para ver por ejemplo toda la descripción de las acciones, o bien ver solo la primera línea de cada acción, que proporciona un vistazo general, y desplegar el resto de una acción cuando se quiere examinar. En el ejemplo inferior se muestra esa opción. En el dominio, varias acciones están ocultas, mostrando sólo su primera línea, que indica el nombre de la acción, y hay una acción desplegada mostrando la definición completa de esa acción. Esta opción suele indicarse con un símbolo más (para expandir) o menos (para colapsar y esconder) dentro de un cuadrado pequeño, en el margen izquierdo del editor. También se suelen encontrar opciones en el menú del editor, para colapsar o expandir todos los bloques de código encontrados.

2. ESTADO DEL ARTE

```
1 (define (domain barman)
2   (:requirements :strips :typing)
3   (:types hand level beverage dispenser container - object
4     ingredient cocktail - beverage
5     shot shaker - container)
6   (:predicates (ontable ?c - container)
7
8   (:action grasp
9
10  (:action leave
11
12  (:action fill-shot
13
14  (:action refill-shot
15
16  (:action empty-shot
17
18  (:action clean-shot
19    :parameters (?s - shot ?b - beverage ?h1 ?h2 - hand)
20    :precondition (and (holding ?h1 ?s)
21                      (handempty ?h2)
22                      (empty ?s)
23                      (used ?s ?b))
24    :effect (and (not (used ?s ?b))
25                (clean ?s))
26  )
27
28  (:action pour-shot-to-clean-shaker
29
30  (:action pour-shot-to-used-shaker
31
32  (:action empty-shaker
33
34  (:action clean-shaker
35
36  (:action shake
37
38  (:action pour-shaker-to-shot
39
40  )
41  )
42  )
43  )
44  )
45  )
46  )
47  )
48  )
49  )
50  )
51  )
52  )
53  )
54  )
55  )
56  )
57  )
58  )
59  )
60  )
61  )
62  )
63  )
64  )
65  )
66  )
67  )
68  )
69  )
70  )
71  )
72  )
73  )
74  )
75  )
76  )
77  )
78  )
79  )
80  )
81  )
82  )
83  )
84  )
85  )
86  )
87  )
88  )
89  )
90  )
91  )
92  )
93  )
94  )
95  )
96  )
97  )
98  )
99  )
100 )
```

Fig. 5. Editor Notepad++ mostrando un fichero de PDDL.

A continuación se comentan varias herramientas útiles para el desarrollo específicamente en PDDL. Todos ellos permiten editar ficheros de PDDL de dominio y de problemas, realizan coloreado de sintaxis, identificando las etiquetas propias de PDDL (**:requirements**, **:predicate**, **:action**, etc.), y detectan apertura y cierre de paréntesis, entre otras características.

2.5.3.1. PDDL Studio

PDDL Studio es una aplicación de tipo IDE, especialmente dedicada para este lenguaje. Fue desarrollada por Miroslav Chomut y Tomas PLch [47].

Puede encontrarse aquí: <http://amis.mff.cuni.cz/PDDLStudio/> y es posible descargarlo mediante petición.

Se trata de una herramienta bastante completa, con editor de PDDL con realce de sintaxis, autocompletar, detecta una serie de errores en los ficheros PDDL, tanto de tipo sintáctico como de tipo semántico, y un gestor de proyecto. Su interfaz de usuario es similar a otros IDEs (Fig. 6).

Fig. 6. Captura del IU de PDDL Studio. Fuente: PDDL Studio web site.

2.5.3.2. Editor myPDDL

MyPDDL [61] es un proyecto para el desarrollo de herramientas que faciliten trabajar con PDDL e ingeniería del conocimiento. Dentro del proyecto, se han desarrollado 2 editores de PDDL. Uno online, basado en el editor ACE, y otro offline, basado en el editor Sublime (Fig. 7).

Permite escribir en PDDL con la ayuda de coloreado de sintaxis, y uso de *templates* (pequeños trozos de código que se usan frecuentemente).

```
1 (define COFFEE
2
3   (requirements
4     :typing)
5
6   (:types room - location
7           robot human _ agent
8           furniture door - (at ?l - location)
9           kettle ?coffee cup water - movable
10          location agent movable - object)
11
12   (:predicates (at ?l - location ??o - object)
13               (have ?m - movable ?a - agent)
14               (hot ?m - movable) = true
15               (on ?f - furniture ?m - movable))
16
17   (:action boil
18     :parameters (?m - movable $k - kettle ?a - agent)
19     :preconditions (have ?m ?a)
20     :effect (hot ?m))
21
```

Line 20, Column 22 Spaces: 2 PDDL

Fig. 7. Editor Sublime mostrando coloreado de sintaxis sobre un dominio en PDDL.

2.5.3.3. Planning.Domains

Planning.Domains [41] es una iniciativa de Christian Muise. Se trata de un sitio web (Fig. 8) que integra un repositorio de problemas PDDL, con un editor de PDDL, y con planificadores en servidores. Fue presentada en 2016. El editor está desarrollado en Java Script y usa el editor ACE.

Fig. 8. Página principal del sitio Planning.Domains.

El editor de Planning.Domains (Fig. 9) está basado en el editor para JavaScript ACE. El editor permite coloreado por sintaxis, templates, detección de paréntesis, etc.

Fig. 9. Editor de Planning.Domains.

Los enlaces para acceder son:

<http://planning.domains/>

<http://editor.planning.domains/>

La ventaja de este entorno es permitir usar problemas en PDDL ya existentes, muy conocidos entre los expertos del área, y poder modificar y obtener soluciones, sin necesidad de instalar entornos o herramientas, independientemente del SO, siempre que se disponga de un navegador web compatible y conexión a Internet.

2.5.3.4. Editor PLAGUE

PLAGUE [64] se inició como un Trabajo de Fin de Grado de la universidad de Málaga, desarrollado por José Luis Usero Vílchez.

Se trata de un editor que permite editar ficheros PDDL de dominio y de problemas (Fig. 10). Es bastante sencillo, puesto que tiene fines pedagógicos. Emplea PDDL4J y JavaGP, para poder proporcionar solución a los problemas planteados.

Fig. 10. Captura de pantalla del editor PLAGUE.

Existe un canal de YouTube.es con unos videos del editor PDDL: PLAGUE, explicando su funcionamiento y características.

Enlace: <https://www.youtube.com/watch?v=NbBrgTbWBpo>

2.6. Otras técnicas

2.6.1. Grafos, cliques y problemas combinatorios.

En este trabajo se emplean algoritmos y técnicas que están relacionados con problemas en grafos, por lo que se comenta brevemente en qué consisten. Se considera un grafo [63], [69] como un conjunto de elementos denominados vértices o nodos, normalmente representados por círculos, unidos por líneas denominadas aristas o arcos. Cada círculo puede tener en su interior un identificador: un número o una letra. Son objeto de estudio de la teoría de grafos, una disciplina de las matemáticas.

Los grafos sirven como abstracción para abordar numerosos problemas matemáticos. Hay diversos tipos de grafos y nos centraremos en grafos finitos, no dirigidos y sin pesos.

Un clique es un subgrafo en los que todos sus vértices están conectados con todos. Estos subconjuntos tienen interés en diversos campos. Encontrar el clique de mayor cardinalidad (número de nodos que lo compone) en un grafo, se conoce como el problema del máximo clique, y es un problema NP-Difícil.

Los problemas NP son un conjunto de problemas para los que no se ha encontrado solución en un tiempo polinomial. Esto significa que cuando aumenta el tamaño del problema, los algoritmos para resolverlo necesitan un número exponencial de pasos para resolverlos. Se denominan problemas combinatorios a aquellos que, para encontrar una solución, deben comprobarse diferentes combinaciones de variables.

Otro problema combinatorio muy conocido es el coloreado mínimo de un mapa o grafo, consistente en encontrar el menor número de colores necesario para asignar un color a cada territorio del mapa (o nodo del grafo) de forma que ninguno tenga un vecino adyacente de igual color (dos nodos del grafo son vecinos si existe una arista que los conecta).

Determinados problemas pueden convertirse, como abstracción, en un problema de búsqueda de un clique en el grafo, o en un problema de encontrar un máximo clique, como es el caso de la propuesta de codificación híbrida que se comenta en el capítulo 3.

Es importante resaltar las aportaciones de Pablo San Segundo (uno de los directores de esta tesis) en este campo.

2.6.2. Bitstrings

Se denominan bitstrings o cadenas de bits a una estructura de datos que permite almacenar y operar con una secuencia de bits de longitud arbitraria.

Los ordenadores actuales trabajan con registros de tamaño de palabra (*word*) típicamente de 32 o 64 bits. Eso indica su capacidad de direccionamiento de memoria y también determina la cantidad de datos básicos con los que puede operar. Así, un procesador de 64 bits puede realizar operaciones como AND, NOR, NOT, sumas y restas, etc. con operandos de 64 bits, lo que significa el doble de capacidad que un procesador de 32 bits.

Casi siempre es necesario trabajar con un mayor número de bits que el proporcionado por el tamaño de la palabra del procesador, por lo que se emplean tipos de datos y estructuras de datos, que son bien conocidas por los programadores informáticos, por ejemplo listas, vectores (*arrays*), matrices, etc.

Los bitstrings son estructuras que tiene la particularidad de almacenar un conjunto de bits ordenados. Suelen almacenarse como un array de words. Estas estructuras, al ser binarias, pueden aprovechar algunas de las características de los procesadores para realizar ciertas operaciones de forma muy eficiente. Por ejemplo, las operaciones lógicas, que están presentes en los procesadores, pueden realizarse muy rápidamente al aprovechar el paralelismo de la palabra del procesador, y de las estrategias de acceso y escritura a memoria, que favorecen la lectura/escritura de bytes o palabras de/desde la memoria.

Un uso más avanzado consiste en el uso de hardware, generalmente tarjetas graficas “especiales” que permiten el procesamiento de datos de forma general, no sólo para gráficos (véase CUDA [73]).

- Ventajas:
 - Son estructuras muy compactas, para almacenar bits.
 - Al aprovechar el paralelismo del tamaño de palabra, las operaciones básicas son muy rápidas y usan eficientemente la memoria.
 - No usan compresión y no tiene pérdidas de información.
- Desventajas:
 - Sólo almacenan bits individuales (otros tipos de datos requieren codificación a bits explícitamente).
 - En el caso de poca densidad de unos, desaprovecha enormemente la memoria.
 - En algunos lenguajes, no existe esta estructura o es difícil de usar, no es fundamental.

Aplicaciones de los bitstrings

En el caso de necesitar almacenar grupos de bits individuales, es probablemente la mejor opción. Los bitstring pueden ser usados como una secuencia ordenada de bits, o simplemente como un conjunto de bits en el que no importa el orden.

Suelen usarse para representar un conjunto de flags. Se emplean en estructuras de colas con prioridad, y en indicadores de uso de memoria, discos, inodes, etc.

También se usan en compresión de datos y criptografía.

En este trabajo se ha usado una biblioteca de bitstrings, desarrollada por uno Pablo San Segundo, para proporcionar mejoras de rendimiento. Esta biblioteca está desarrollada usando C++ estándar. Aunque es preferible su uso con sistemas de 64 bits, puede usarse en distintas plataformas de 32 y 64 bits.

Sus ventajas son:

- Usa *intrinsics* para mejorar el rendimiento en sistemas Intel (opcional).
- Implementa algoritmos optimizados para el cálculo de operaciones lógicas.
- Implementa algoritmos optimizados para detectar operaciones de búsqueda (primer bit a 1, primer bit a 0 (*find first 1/0*), contar bits a 1, contar bits a 0 (*population 1/0*), etc.
- Dispone de un modo “*sparse*”, con el que se puede ahorrar memoria si se conoce que el número de unos almacenados será bajo respecto al tamaño del bitstring.

En este trabajo se emplean esas estructuras para mejorar el rendimiento del clique, del compilador a SAT y su formato lógico o CNF, y del SAT planner. Esta biblioteca está accesible al dominio público en [74]

En nuestro trabajo su uso se justifica plenamente, puesto que gran parte de los procesos a realizar tratan datos booleanos, y muchos, sobre los cuales se aplican muchas operaciones de tipo lógico, véase por ejemplo [58].

3. ENTORNO PARA PLANIFICACIÓN AUTOMÁTICA

Como parte de esta tesis se ha implementado un entorno o marco de trabajo y desarrollo (*framework*) para planificación automática, basado en planificación clásica. Está concebido para ser integrado en la arquitectura de un agente inteligente autónomo, y se han dispuesto herramientas de edición de dominios y simulación de escenarios de planificación. Parte del trabajo consiste en un compilador para *SAT planning*.

3.1. Arquitectura de un agente inteligente para planificación

En este trabajo se parte de una propuesta de una arquitectura para un agente inteligente, poniendo énfasis en planificación automática y simulación del entorno. La arquitectura tiene en cuenta también una base de conocimiento de alto nivel y aprendizaje.

En el diagrama de la Fig. 11 se muestra al agente en relación con el entorno, a través de sus sensores y actuadores. Todo lo que el agente conoce del mundo, se obtiene a través de un conjunto de sensores y de su modelo del mundo (que puede estar incluido en su base de conocimiento, o bien aparecer de forma

separada). El agente puede efectuar acciones que afectan al mundo (o a la situación del agente en ese mundo) mediante un conjunto de actuadores.

Fig. 11. Esquema general de la arquitectura de un agente.

Este es un esquema general de agente inteligente, que puede tener un cierto nivel de autonomía. Se muestran algunas funciones internas o partes del software del agente que se han incluido a fin de aportar un comportamiento inteligente, como el módulo de aprendizaje o el de planificación de acciones. En este agente se tienen en cuenta los objetivos o motivación del robot para actuar, para hacer algo en el mundo, así como un módulo que aporta aspectos emocionales (reconocer características emocionales del mundo, y también tener respuestas con carácter emocional). Este último módulo es especialmente

3. ENTORNO PARA PLANIFICACIÓN AUTOMÁTICA

importante en el caso de robots o agentes inteligentes que van a interactuar con frecuencia con personas.

Este esquema es parecido a otras arquitecturas, como por ejemplo la del robot Urbano [55], que actualmente dispone de módulos de conocimiento, emocionales, etc.

En este trabajo el enfoque se centra en la planificación, y en la validación de la misma, incluyendo un entorno de simulación en la que realizar pruebas (Fig. 12).

Fig. 12. Esquema de la arquitectura del agente propuesto.

Para realizar el software del agente, se plantearon una serie de módulos tanto internos (planificador, base de conocimiento, evaluador para aprendizaje) como externos (para la simulación e interfaces de usuario), que permitiesen un entorno integrado y modular en el que poder desarrollar y probar variaciones a la programación del agente.

3. ENTORNO PARA PLANIFICACIÓN AUTOMÁTICA

Este sistema se descompone a su vez en una serie de módulos con diferentes funciones, que se muestran en el siguiente diagrama (Fig. 13) :

Fig. 13. Esquema de bloques de la arquitectura del agente inteligente.

3. ENTORNO PARA PLANIFICACIÓN AUTOMÁTICA

En él se encuentran una serie de módulos, agrupados por funciones (Tabla 5):

Razonador o "alto nivel"	Representa un nivel cognitivo y una base de conocimiento. Gestiona las "misiones" y el conocimiento de las acciones, propiedades del mundo, etc.
Planificador	Un planificado basado en SAT plan, que acepta problemas en PDDL y obtiene planes validos de los problemas. Puede sustituirse por otro planificador.
SAT <i>solver</i>	Un solver SAT empleado por el planificador.
Interfaz de Usuario	Incluye un simulador para mostrar la aplicación de un plan, permitir configurar el sistema y visualizador 3D.
Análisis y Evaluación	Un módulo análisis de los problemas y planes, que extrae conclusiones para proporcionar realimentación a la base de conocimiento o alto nivel.

Tabla 5. Agrupación de módulos por funciones relacionadas.

Como parte de este trabajo se han creado módulos y técnicas para emplear en la mayoría de esos grandes bloques en los que se divide la arquitectura.

Los componentes principales del sistema propuesto, tal como se muestra en el esquema son los siguientes:

- Parser PDDL: Se encarga de leer ficheros con una descripción del dominio y del problema (PDDL Domain y PDDL Problem), en lenguaje PDDL y generar una estructura (PDDL Tree) que permita analizar y trabajar con esos elementos,
- Conversor SAT: convierte un problema expresado en PDDL a un problema SAT, expresado en clausulas de lógica proposicional. Este componente genera asimismo una Tablas de Símbolos que establece la equivalencia entre la nomenclatura PDDL y las variables usadas en el problema SAT.
- SAT Solver: Es un módulo independiente, que admite como entrada un conjunto de clausulas lógicas y determina una posible asignación de valores de verdad para satisfacer todas las clausulas. Si no es posible, debe indicarlo. Esa asignación equivaldrá a una solución del problema de planificación.

- **Decoder SAT:** se trata de un componente auxiliar al Conversor SAT, cuya función es obtener las acciones que componen la solución a partir de las variables que forman parte de la solución del problema SAT. Emplea para ello la Tabla de Símbolos construida previamente.
- **Simulador:** Permite mostrar de forma gráfica los elementos de un entorno, y permite simular la ejecución de un plan válido, obtenido previamente. Permite mostrar mediante gráficos 3D los objetos existentes en el escenario del problema de planificación. Emplea ficheros auxiliares y de modelos 3D para la representación gráfica.
- **Scheduler/Executor:** Es el componente encargado de realizar la secuenciación de las acciones que forman la solución del problema de planificación, modificando el estado del mundo.
- **PDDL Editor:** Puede considerarse parte del interfaz de usuario, tiene como función facilitar la creación y modificación de ficheros PDDL por parte del usuario.
- **IU Simulador:** Se compone de una serie de componentes visuales que complementan la visualización del simulador.
- **Evaluador:** Se trata de un módulo que analiza la ejecución de un plan en el simulador, y lo compara con la descripción del dominio. Este evaluador puede descubrir problemas en inconsistencias en los problemas o dominios de planificación.
- **Generador de Problemas:** Se encarga de crear variaciones en los problemas de planificación, teniendo en cuenta el conocimiento almacenado en el alto nivel, que servirán como problemas de aprendizaje.
- **PDDL Coder:** Está encargado de codificar los problemas creados por el Generador de Problemas a lenguaje PDDL válido.

En el resto de apartados de este capítulo y el siguiente se comentan los distintos módulos creados y las aportaciones a cada uno de ellos.

3.2. Analizador de lenguaje PDDL

Uno de los módulos del sistema es un analizador sintáctico (*parser*) de lenguaje PDDL (Fig. 14). Este módulo lee ficheros de texto, conteniendo bien una definición de dominio o bien una definición de problema. Normalmente, al intentar resolver un problema de planificación, se leerán dos ficheros: uno de dominio y otro de problema, y éste último debe estar basado (y hacer referencia) al dominio que se está usando. En nuestro caso la lectura de uno u otro fichero pueden realizarse de forma independiente, puesto que en ocasiones no se pretende obtener un plan a un problema, sino analizar las acciones o las características del problema, para comprobar su corrección o la existencia de

posibles errores, elaborar información complementaria a un plan, crear variaciones de los problemas, etc. Habitualmente lo que se pretende es resolver un problema concreto de planificación y para ello deben usarse ambos ficheros.

En la arquitectura del agente, el *parser* PDDL, el conversor a SAT y el SAT *solver* son una parte importante del módulo de planificación, y por tanto del agente.

Fig. 14. Diagrama de Componentes del sistema del analizador de PDDL.

3.2.1. Gramática de PDDL

Los parser desarrollados en este analiza dominios y problemas en versión PDDL v1.2 con las opciones de *requirements* *:strips*, *:types*, y *:negative-preconditions*. Si bien no contempla todas las opciones de la última versión de PDDL (v3.1), estas características engloban algo más que el lenguaje STRIPS. Esta restricción reduce los tipos de problemas que puede tratar este sistema, pero gran parte de los sistemas desarrollados en torno a PDDL se ciñen a STRIPS y tienen como punto de partida estas mismas restricciones. La inclusión de la posibilidad de usar tipos es muy conveniente para facilitar la clasificación de los objetos existentes en un problema en PDDL y reducir en parte las posibilidades en la instanciación de parámetros (*grounding*).

La gramática de PDDL, en sus diversas versiones está referenciada en el apartado 2.5.

La función del *parser* es leer un fichero de texto, que contenga una definición de dominio o bien de problema en PDDL (como se indicó, suele realizarse la lectura de un fichero de problema y de otro fichero con el dominio correspondiente) y obtener a partir de ellos un conjunto de datos organizado en

estructuras de datos, (en nuestro caso objetos declarados como clases de C++), que tienen una relación directa con los elementos que se pueden definir en un fichero PDDL.

El *parser* se ha construido como un analizador descendente recursivo a partir de la gramática BNF propuesta por los creadores del lenguaje PDDL. En otro apartado relativo al PDDLBuilder, se verá otra técnica de análisis basada en expresiones regulares.

El lenguaje PDDL reconocido tiene la expresión, en notación BNF:

```
<domain> = ( define ( domain <identif> )
              <domain-defs>* )
<domain-defs> = ( :requirements <requirements> )
                  | ( :types <types> )
                  | ( :predicates <predicates> )
                  | ( :action <action> )
<requirements> = <requirem-labels>*
<types> = <identif>*
<predicates> = ( <a-predicate>* )
<a-predicate> = <identif> <typedvarlist>
<typedvarlist> = <variable>* - <identif>
<identif> = ([A..Z][a..z]) ([A..Z][a..z][0..9]_)*
<variable> = ?<identif>
<action> = <identif> :parameters ( <typedvarlist> )
           :precondition ( <goal-descrip> )
           :effect ( <effect-descrip> )
<goal-descrip> = and ( <goal-descript> )
                  | not ( <identif> <atomic-formula> )
                  | <atomic-formula>
<atomic-formula> = (<identif> | <variable> ) *
<problem> = ( define ( problem <identif> )
              ( :domain <identif> )
              <problem-defs>* )
<problem-defs> = ( :objects <objects> )
                  | ( :init <init> )
                  | ( :goal <goal> )
<objects> = ( <identif>* [- <identif> ] ) *
<init> = <identif> [ <name-list> ]
<name-list> = <identif>*
<goal> = <goal-descrip>
<requirem-labels> = :strips | :typing
                  | :negative-preconditions
```

En la primera versión no se considera el operador OR, ya que puede reducirse a usar varias acciones similares para contemplar los distintos casos que generan las disyunciones.

En los efectos no debe aparecer el operador OR, ya que daría lugar a acciones no deterministas.

3.2.2. Analizador de lenguaje PDDL

3.2.2.1 Analizador

La primera versión desarrollada se basa en la técnica del análisis descendente recursivo. Esa es la técnica habitual para la generación de analizadores de gramáticas independientes del contexto, como los lenguajes de programación. Se usó el lenguaje de programación C++. Al estar escrito en ese lenguaje nos permite alcanzar potencialmente la mayor eficiencia y velocidad posibles, si se optimiza el uso de estructuras.

El analizador se implementa fundamentalmente en 2 niveles: El parser (analizador sintáctico) y el scanner (analizador léxico). El parser se ocupa de la gramática y la estructura del análisis, mientras que el scanner realiza las tareas de nivel inferior: obtener caracteres del flujo de entrada, identificarlo como perteneciente a un token (elementos más pequeños de la gramática, indivisibles) o a una palabra clave, y generar los datos de los tokens o elementos del lenguaje para proporcionarlos al parser. También se mantiene información del nivel de anidado de los paréntesis, información de la línea de texto que se está analizando, etc.

La gramática que se emplea es una versión simplificada de la gramática de PDDL 2.1 (y compatible con el resto de las versiones de PDDL posteriores), adecuada para un planificador basado en SAT.

el funcionamiento habitual en este tipo de analizadores es el siguiente: el parser inicia el análisis, y realiza llamadas al scanner cada vez que pretende analizar un elemento del fichero de entrada. Se intercalan por tanto ambos niveles de análisis (sintáctico y léxico).

Para la implementación se definieron dos clases, una se denomina PddlParser y otra PddlScanner (Fig. 15). El scanner está implementado siguiendo un modelo de máquina de estados. El parser por su parte está estructurado en distintos métodos, como suele ser habitual en este tipo de analizadores, en los cuales se define una función (o método) para cada símbolo no terminal presente en la gramática BNF, y cada método es responsable de reconocer la regla asociada a ese símbolo no terminal.

3. ENTORNO PARA PLANIFICACIÓN AUTOMÁTICA

Fig. 15. Diagrama de Clases del analizador PDDL.

3.2.2.2. Scanner

El Scanner implementa una maquina de estados que facilita la identificación de los elementos terminales, y tiene en cuenta la línea de texto que se está analizando. El esquema de la máquina de estados se muestra en la Fig. 16:

Fig. 16. Máquina de estados del reconocedor de tokens (scanner).

El scanner reconoce los siguientes *tokens*:

- Palabras clave (*keywords*), como `:types`, `:requirements`, etc.
- Identificadores, que incluyen nombres de objetos, de tipos, de predicados y de acciones.
- Variables, usadas como parámetros (son identificadores que comienzan con el carácter `'?'`).
- El símbolo menos `'-'`, usado como separación entre variables y tipos.
- Paréntesis abiertos y cerrados, que establecen la estructura del resto de elementos.

En PDDL no se distinguen mayúsculas y minúsculas, y los identificadores pueden componerse de varios caracteres, incluyendo letras (alfabeto inglés, no se admiten acentuadas o la ñ, ç, etc.), números (pero no pueden aparecer en la

primera posición), y los separadores '-' (menos) y '_' (barra baja). Los espacios, tabuladores y cambios de línea no forman parte de los identificadores.

3.2.2.3. Parser

El parser se implementa en la clase PddlParser (Fig. 17). Este parser identifica los elementos de un fichero PDDL (con la ayuda del scanner) y generará un conjunto de estructuras de datos en memoria, en las que se distinguen objetos, predicados, acciones (con sus parámetros, precondiciones y efectos), el estado inicial y el estado (o estados) objetivos.

Fig. 17. Detalle de la clase PddlParser.

Básicamente, el parser pide tokens (es decir, elementos terminales del lenguaje) de forma sucesiva al scanner, el cual se los facilita en orden con información

relativa al token (tipo, texto del token, etc.). En esa información también se incluye la línea y la posición de ese token en el fichero de texto, puesto que son datos útiles la gestión de errores.

El parser sigue una estructura de analizador de lenguaje descendente recursivo. En ese esquema, cada elemento no terminal de la gramática se implementa usando una función que comprueba la validez de la regla asociada a ese elemento no terminal, produciendo un error en caso de no cumplirse. Según se van comprobando las reglas, se va generando una estructura interna de información en la que se almacenan los elementos de PDDL definidos en los ficheros. Estas estructuras en nuestro caso se implementan en C++ como objetos de varias clases, que se explican con detalle más adelante.

Si el parser detecta un error sintáctico o léxico en el fichero PDDL que está leyendo, se interrumpe en ese punto, produciendo un mensaje de error que indique al usuario el tipo de error, la línea y posición del elemento que se ha detectado erróneo, de forma que el usuario tenga una buena información acerca del problema encontrado.

Algunas reglas de validez de los ficheros PDDL que realiza el parser se sitúan a nivel semántico, y en algunos casos implican a ambos ficheros (el de dominio y el de problema), por ejemplo: comprobar que el nombre de dominio que se especifica en el fichero de problema, se corresponde con el del dominio que se está usando. El parser desarrollado tiene la opción de identificar sólo un fichero de dominio, o sólo de problema, o bien identificar primero uno de dominio y después otro de problema.

3.2.3. Analizador de PDDL basado en PEG

Esta versión del analizador se realizó para el editor PDDL Builder. Está programado en Javascript, y se basa en el generador de analizadores de lenguaje tipo PEG (*parsing expression grammar*) disponible en <https://pegjs.org/>

Los analizadores PEG realizan el análisis sintáctico y léxico, son parecidos a un comprobador de reglas de estructura, o de expresiones regulares cuyos símbolos son los tokens definidos en el lenguaje.

Esta versión usa más memoria que la anterior, y al estar programado en Javascript es menos eficiente, pero esto no suele ser un problema puesto que los problemas de planificación son, tanto en términos de extensión del fichero como en el de elementos en el PDDL, bastante reducidos. La ventaja estriba en estar realizado en el mismo lenguaje que el resto de PDDL Builder con lo que queda perfectamente integrado.

3.2.4. Estructuras de almacenamiento de PDDLs

El análisis de los ficheros PDDL de un problema de planificación, generan como resultado una serie de elementos que se deben guardar para realizar la planificación o realizar cualquier análisis con ellos.

Los elementos del lenguaje PDDL empleado se expresa en el esquema para el dominio (Fig. 18):

Fig. 18. Diagrama de elementos de dominio en PDDL.

El apartado Requisitos se indica en otro color por ser un elemento de configuración de las opciones del lenguaje PDDL, más que del problema de planificación definido.

En estas estructuras se almacenan, una vez identificados *tokens* y símbolos no terminales correspondientes, los siguientes elementos:

Dominio

- Nombre del dominio.
- Requerimientos (opciones del lenguaje).
- Declaración de los tipos (si hay) indicando el tipo padre al que pertenecen.
- Constantes.
- Predicados:
 - Nombre.
 - Cardinalidad y nombre de los parámetros, indicando su tipo.
- Acciones:
 - Nombre.
 - Nombre de los parámetros, indicando su tipo.
 - Precondiciones, indicando los predicados y las variables en ellos.
 - Efectos, indicando los predicados, y las variables en ellos.

Las precondiciones y los efectos de las acciones se deben almacenar en función de los parámetros de la acción, y ambos son expresiones lógicas que pueden incluir los operadores lógicos Y (and) y Negado (not).

Esquema de elementos correspondiente para el problema (Fig. 19):

Fig. 19. Diagrama de elementos de problema en PDDL.

En la definición del problema concreto, especificado en PDDL en el fichero de *problem*, se identifican los elementos:

Problema

- Nombre del problema.
- Nombre del dominio (debe coincidir con el dominio anterior).
- Objetos (constantes, etiquetas que los designan), indicando su tipo.
- Estado inicial (una serie de predicados que se cumplen), indicando:
 - Nombre.
 - Parámetros instanciados.
- Objetivo (conjunto de predicados que deben cumplirse), indicado:
 - Nombre.
 - Parámetros instanciados.

El estado inicial es un conjunto de predicados instanciados (*grounded*), es decir, con sus parámetros sustituidos por objetos o constantes. Los objetos deben estar definidos en el fichero de problema y las constantes deben estar definidas en el fichero del problema.

El objetivo se expresa como una conjunción de predicados instanciados, que deben cumplirse (o no cumplirse, si aparecen negados).

Estos datos se almacenan en un conjunto de clases en C++, mostradas en el diagrama de la Fig. 20.

En estas estructuras puede almacenarse adecuadamente problemas y dominios de PDDL limitados a las opciones STRIPS + *typing*.

3.2.4.1. Descripción de las clases

PddlDomain: Clase contenedora del resto de elementos pertenecientes a un dominio de PDDL. Incluye algunos métodos para acceder y facilitar operaciones con listas completas de elementos PDDL.

Pddl Problem: Clase contenedora de los elementos de un problema de PDDL.

PddlGoal: Clase que almacena una lista de predicados. Aunque se denomina Goal, también sirve en general para otras funciones.

Pddl Predicate: Clase que almacena los datos de un predicado: nombre, y lista de parámetros.

PddlTypedVarList: Clase que contiene una lista de identificadores. Se emplea para los distintos parámetros de un predicado o una acción.

PddlAction: Clase contenedora de los elementos de una acción. Indica el nombre de la acción, la lista de parámetros, y listas de predicados para las precondiciones y los efectos.

PddlTypedVar: Clase básica para trabajar con variables o parámetros de PDDL. Puede tener un tipo, útil si se ha definido `:typing` en el dominio.

Fig. 20. Diagrama de clases para almacenar un dominio PDDL.

En este esquema, se usa **PddlGoal** para almacenar expresiones lógicas de predicados que aparecen en las acciones, tanto las precondiciones como sus efectos. Esto es posible, aunque realmente los datos en uno y otro caso no son exactamente los mismos, pero la estructura de almacenamiento es válida.

3.2.5. Exportación e importación a formato JSON

Una utilidad incluida en el desarrollo es la exportación de la estructura de datos tanto de un problema como de un dominio a formato JSON.

El formato JSON (JavaScript Object Notation) [75] es un formato para intercambiar datos en forma de texto, es un formato ligero y de sintaxis sencilla, semejante a los lenguajes Java, Javascript, C, etc. JSON define dos tipos de estructuras de datos: objetos y listas. Los objetos se componen de pares nombre-valor, mientras que las listas son una secuencia ordenada de valores. Los valores, por su parte, pueden ser cadenas de caracteres, números, o también objetos o listas, de forma que puede haber anidamiento jerárquico de

estructuras, permitiendo almacenar estructuras de datos muy complejas mediante reglas sencillas.

La estructura en JSON sigue el mismo esquema jerárquico presente en las estructuras de objetos en C++ (y por tanto similar al de los ficheros PDDL originales) sólo que los valores y sus relaciones se expresan en formato de texto ASCII JSON. Puesto que existen bibliotecas, para muchos lenguajes de programación, que permiten exportar e importar estructuras de datos siguiendo el formato JSON Esta conversión facilita compartir las estructuras PDDL entre lenguajes, por ejemplo entre C++ y JavaScript, sin necesidad de escribir lectores de PDDL para ambos lenguajes.

3.3. El compilador PDDL-SAT

3.3.1. Introducción

Una forma de resolución de un problema de planificación consiste en transformar el problema en un problema SAT equivalente, tal y como se ha visto en el capítulo 2.

En este trabajo se ha acotado el problema en dos sentidos:

- Como se indicó en apartados anteriores, se considera parte del formato completo de PDDL, incluyendo solamente STRIPS y tipos de objetos.
- Por otro lado, en la codificación a SAT se consideran sólo algunas opciones, entre las muchas disponibles. Se han escogido aquellas que a priori pueden producir menos variables lógicas y cláusulas, para reducir problemas de explosión combinatoria, y que permitan una codificación compacta (por ejemplo, permitiendo paralelismo de acciones en una misma etapa, si es posible).

A pesar de esas limitaciones, se ha buscado diseñar e implementar una codificación que sea perfectamente válida y completa, es decir que sea efectivamente correcta para resolver problemas de planificación.

La conversión que se propone recoge algunas técnicas existentes y algunas novedades, usándolas en la implementación realizada, intentando que la conversión sea rápida a la vez que mantenga un sentido entre ambos problemas (planificación y SAT) que permita ser comprendido fácilmente por otros programadores.

Como se ha comentado, la técnica para emplear SAT para la resolución de un problema de planificación en N etapas consiste en intentar resolver un problema SAT en el que se ha codificado un problema de planificación en i etapas, y si no es posible, intentarlo con un número de etapas $i+1$, de forma sucesiva, hasta lograr encontrar una solución. En este trabajo se ha tenido en cuenta esta

necesidad de generar de forma repetida el mismo problema de planificación pero con distinto horizonte de tiempo, incorporando una optimización para esos casos.

El resultado de la implementación ha sido un compilador SAT propio, usado en nuestro sistema, que nos permite modificar la codificación entre diversas opciones y explorar codificaciones híbridas.

3.3.2. Esquema del compilador SAT en el contexto del sistema

La conversión a problema SAT se realiza partiendo de las estructuras de PDDL que ya se han obtenido por el parser PDDL. El conversor realizará la traducción y generará por un lado un fichero DIMACS con el problema SAT en formato CNF, y preparará una lista estableciendo la correspondencia entre las variables lógicas y los distintos elementos de PDDL (predicados y acciones). En el esquema de la Fig. 21 se muestra el conversor y su relación con otros módulos del sistema.

Fig. 21. Conversor SAT en el entorno del sistema.

3.3.3. Notación

En este apartado se emplean expresiones de lógica proposicional con una serie de variables que hacen referencia a elementos del problema de planificación (normalmente expresado en PDDL o STRIPS). Para dar un nombre que podamos representar algo para los humanos, se siguen los siguientes criterios:

- Las variables que representan predicados, se nombran comenzando con el nombre del predicado, seguido de los nombres de los parámetros.

- Las variables que representan acciones, se nombran comenzando por el nombre de la acción, y a continuación los nombres de los parámetros.
- Se emplea el símbolo ‘_’ (subrayado o barra baja) para separar el nombre de predicado o acción, y cada uno de los parámetros.
- El nombre acaba en un símbolo ‘@’ (arroba) y un numero entero, que representa el numero de etapa o instante de tiempo en el que se considera ese predicado o acción.
- Los números de etapa para los predicados comienzan en 0 (estado inicial), representan que valor de verdad de ese predicado en un instante de tiempo.
- Los números de etapa de las acciones comienzan en 0 y representan si se realiza o no esa acción en un instante de tiempo n , siendo el estado anterior el que tiene numero de etapa n , y provocando efectos reflejados en el estado de la etapa $n+1$.
- Se emplea el símbolo ‘-’ (menos o guión) delante del nombre para indicar su negado, o que se cumple el valor de verdad FALSO para esa variable.

Así, por ejemplo los predicados en PDDL siguientes:

ROBOT-AT(room1)
BOX-AT(box3,room4)

Darán lugar a unas variables lógicas en forma:

robot_at_room1@0
robot_at_room1@1
...
box_at_box3_room4@0
...

En este ejemplo, la primera variable es el valor en la etapa de tiempo 0 (inicio), y la segunda variable se refiere al valor en el instante de tiempo siguiente.

El número de etapas de tiempo vendrá dado por el controlador que se encarga de llamar al SAT solver, indicando el numero de etapas en los que se busca solución.

Los predicados siguen la notación:

NOMBRE_PREDICADO_param1_param2..._paramN@numetapa

Las acciones se convierten igualmente a variables lógicas, así estas acciones:

MOVE(room1, room2)

PICK-BOX(box3, room2)

MOVE(room2, room3)

Suponiendo que se producen en los instantes de tiempo 0, 1 y 2, se convierten a las variables lógicas con nombres:

move_room1_room2@0

pick_box_box3_room2@1

move_room2_room3@2

En general la notación sigue el esquema:

NOMBRE-ACCION_param1_param2..._paramN@numetapa

Se usa el símbolo menos (-) como operador de negación (not). Así podemos indicar que un predicado no se cumple, o tiene un valor de verdad *falso*, anteponiendo el símbolo menos. También se usa cuando se quiere expresar la condición negada. Por ejemplo:

-robot_at_room1@0

-box_at_box3_room1@0

Estos predicados indican que el robot no está en la habitación room1 en la etapa 0, y que la caja box3 no está en room1, en la etapa 0 igualmente.

3.3.4. Etapas del compilador SAT

En la conversión del problema de planificación (en PDDL) a SAT, se deben realizar una serie de fases o pasos, como se indica en el diagrama (Fig. 22):

Fig. 22. Diagrama de fases en la compilación a SAT.

3.3.4.1. Fase 1 - Lectura y análisis de ficheros PDDL

El compilador a SAT propuesto usa el analizador de PDDL expuesto anteriormente, y por tanto los datos de entrada se encuentran disponibles como una estructura de clases de C++ ya comentada. El analizador debe haber sido llamado en modo de lectura de ambos ficheros: el de dominio y el de problema, para poder disponer de las estructuras con los datos correctos. A estas

estructuras se les añaden algunas otras para facilitar la conversión, y para almacenar estados intermedios y datos de trabajo del compilador.

3.3.4.2. Fase 2 – Conversión de expresiones lógicas

Una vez obtenidas las estructuras de datos que reflejan los ficheros PDDL, se realiza una transformación sobre las expresiones lógicas según donde se encuentran:

- Las precondiciones de las acciones: debe contener solo el operador AND y el operador NOT afectando únicamente a predicados. Si no es así, debe transformarse la expresión de forma que el operador NOT solo afecte a predicados. Esta transformación se realiza aplicando las reglas de Morgan sobre la expresión. Los operadores OR deben desplazarse a la raíz de la expresión. Con ello se generarán distintas acciones aplicables.
- Los efectos de las acciones: Los efectos sólo deben contener el operador AND y el operador NOT afectando únicamente a predicados. Si no es así, se generarían efectos opcionales, es decir, no deterministas. En nuestro modelo no los aceptamos.
- El estado inicial: El estado inicial se transforma a una expresión AND en la que aparecen todos los predicados instanciados que se cumplen en el inicio, y los que no están definidos se deben codificar como predicados que no se cumplen.
- El estado objetivo: El estado objetivo (goal) se transforma a una expresión AND en la que deben cumplirse todos los predicados que se han definido (conservando el operador NOT en su caso). Si aparece el operador OR, se debe desplazar a la raíz de la expresión.

Algunas de estas conversiones se indican en [27].

Las expresiones lógicas se van a almacenar en unas estructuras en C++ que permiten seguir manipulando estas expresiones de acuerdo con las necesidades de la conversión completa a SAT.

3.3.4.3. Fase 3 – Detección de predicados estáticos e inhábiles

Se denomina predicados estáticos o invariantes a aquellos predicados, cuyo estado, o valor de verdad, no cambia durante la realización de un plan. En algunos casos hay predicados que no precisan cambiar porque el problema no lo requiere, pero hay predicados que no pueden cambiar nunca al no existir ninguna acción que los modifique.

Estos predicados tiene la particularidad de permanecer siempre en el mismo estado, es decir, se cumplen (o no se cumplen) y no pueden cambiar a lo largo de un plan. Por tanto estos predicados pueden considerarse constantes, y se puede usar esta circunstancia para simplificar la generación de acciones

posibles. Cuando estos predicados aparecen en las precondiciones de las acciones, pueden descartarse (si se cumplen) o bien puede descartarse esa acción (si no se cumple el predicado estático), puesto que el predicado hace falsa la precondición.

La detección de predicados estáticos se realiza examinando las acciones del dominio, siguiendo la siguiente regla:

Si un predicado no aparece en los efectos de ninguna acción del dominio, ese predicado es estático.

El algoritmo para detectar estos predicados consiste en examinar, para cada predicado, si aparece en los efectos de alguna acción, recorriendo todas las acciones en el dominio. Si aparece el predicado, no es invariante. Si se llega al final y no se ha encontrado el predicado en ninguna acción, es invariante. Los predicados invariantes se marcan para poder procesarlos de forma distinta posteriormente.

Denominamos predicados inhábiles (o no considerados), a aquellos predicados que no se emplean en ninguna precondición. Estos predicados no sirven para tomar decisiones en cuanto a la posibilidad de realizar o no una acción, aunque en algunos casos pueden estar presentes en el estado objetivo.

Estos predicados inhábiles son candidatos a ser eliminados, aún cuando las acciones realicen efectos sobre ellos. Sin embargo, en ese caso, debe considerarse también el estado objetivo. Si no aparece tampoco en el objetivo, ese predicado puede eliminarse, y quitarse toda referencia a él en el problema de planificación puesto que no afecta en absoluto en la búsqueda de un plan válido.

Si un predicado es inhábil y es estático, puede eliminarse ese predicado y sus referencias en el problema de planificación.

Si un predicado es inhábil, no es estático, y no aparece en el estado objetivo (goal), puede eliminarse ese predicado y sus referencias en el problema de planificación

El algoritmo para detectar predicados inhábiles es parecido al anterior: Se examinan todos los predicados, y por cada uno se comprueban todas las precondiciones de las acciones. Si el predicado aparece en alguna precondición, no es inhábil. Si se llega al final de la exploración y no se ha encontrado el predicado, es un predicado inhábil, y puede marcarse como tal. aun así, se debe comprobar el estado objetivo.

Una vez realizada la detección de predicados estáticos, también puede aplicarse un análisis sencillo al objetivo, para comprobar que efectivamente puede ser alcanzable:

Si en el estado objetivo aparecen predicados estáticos, que han cambiado su valor respecto al estado inicial, el objetivo es inalcanzable.

La afirmación anterior indica que los objetivos que requieren un cambio de estado en predicados que no pueden cambiar, no pueden cumplirse y por tanto no puede obtenerse una solución al problema de planificación. Es decir, no existen acciones que puedan pasar del estado inicial a un estado objetivo variando un predicado estático. Es un caso infrecuente en los problemas de planificación, puesto que el usuario que prepara el problema suele (y debe) tener esto en cuenta. Pero puede ser un análisis útil para detectar errores al plantear problemas de planificación, y por esto se ha incluido en nuestra propuesta. También puede ser útil para problemas que son resultado de una generación automática del problema.

El análisis anterior (predicados estáticos e inhábiles, a excepción de la consideración del objetivo) se realiza sobre el dominio, y es válido para cualquier problema sobre ese dominio.

Una vez detectados predicados estáticos, puede usarse esa información para simplificar acciones, al realizar el grounding, como se verá. En esta parte, debe usarse información del problema, puesto que en el estado inicial aparecen los predicados estáticos que tienen valor verdadero.

Al realizar la sustitución de parámetros (grounding) de acciones, si en sus precondiciones aparecen precondiciones que son predicados estáticos, estos pueden sustituirse por su valor de verdad (establecido en el estado inicial). Puesto que estamos tratando con precondiciones tipo STRIPS sin operadores OR, si el predicado tiene un valor verdadero, el predicado puede quitarse sin alterar el problema de planificación. En caso de ser valor falso, la precondición no se cumple, y se puede eliminar la acción instanciada directamente. Esto es válido para cualquier número de etapa, puesto que el predicado estático no puede cambiar a lo largo de un plan.

3.3.4.3. Fase 4 - Reordenamiento de parámetros

La presencia de predicados estáticos o inhábiles, afecta a la generación de acciones posibles. En las acciones definidas en PDDL se definen una serie de parámetros (objetos y constantes) que se usan en las precondiciones y en los efectos de la acción. El orden de los parámetros puede afectar al orden en que se instancian esos parámetros y dan lugar a posibles acciones instanciadas (que son las que constituirán el plan). Asimismo, el tipo de codificación que se empleará en la conversión a problema SAT puede resultar más compacto o no, dependiendo de la agrupación de parámetros de la acción, y por tanto del orden en que aparecen.

En esta fase (que es opcional) se propone cambiar el orden de los parámetros antes de comenzar a instanciarlos (o cambiar el orden en el que se seleccionarán

cada uno de los parámetros para instanciarlos), de forma que los parámetros que se usan en los predicados estáticos se coloquen primero, y por tanto se comprueban antes. En el caso de detectar que no pueden cumplirse durante la generación de las posibles acciones aplicables, esas acciones se rechazan y no es necesario codificarlas, ni mantenerlas almacenadas para el cálculo del plan.

3.3.4.5. Fase 5 - Instanciación (*grounding*) de los parámetros

Todas las acciones emplean parámetros, que se usan en sus precondiciones y efectos como parámetros de los predicados de las expresiones lógicas. Para poder obtener un plan, y para la conversión a SAT del problema, es necesario sustituir los parámetros por valores válidos para esos parámetros, que son objetos y constantes. En ambos casos deben pertenecer al tipo establecido por la declaración de la acción y la declaración de los predicados.

En este paso se generan varias instancias de cada acción, sustituyendo los parámetros definidos en la acción por objetos y constantes válidos. Los objetos y constantes son válidos si pertenecen a un tipo compatible con el tipo indicado en el parámetro al que sustituye. Las constantes están definidas en el dominio, pero los objetos están definidos en el fichero de problema. Por tanto deben emplearse los datos del problema concreto para proceder a la sustitución de parámetros.

De la misma forma, se generan varias instancias de cada predicado, sustituyendo los parámetros del predicado por objetos o constantes que pertenezcan al tipo adecuado.

En esta etapa el uso de la opción *typing* permite reducir bastante el número de instancias de acciones generado, y por eso en nuestra propuesta incluimos desde el inicio la posibilidad de usar tipos. Si no se ha definido esa opción, todos los objetos y constantes son del mismo tipo (*object*).

Para la generación se emplea un algoritmo sistemático recursivo, que va asignando valores (teniendo en cuenta el tipo al que pertenece el parámetro, si se ha definido) primero al primer parámetro, a continuación al segundo, y así sucesivamente. Al usar este algoritmo, es conveniente colocar como primeros parámetros aquellos que son usados en predicados estáticos (y por tanto pueden indicar que la acción no puede realizarse, al detectar que no se cumple el predicado estático simplemente comparándolo con el estado inicial), y por tanto no tiene sentido seguir expandiendo la acción con ese valor de parámetro.

Al final de este proceso se genera una etiqueta por cada acción instanciada y por cada predicado instanciado, con una nomenclatura legible para los usuarios, así:

accion_param1_param2

3.3.4.6. Fase 6 – Generar expresiones para acciones incompatibles

Después de realizar la sustitución de parámetros en las acciones, se obtiene un gran número de acciones posibles. Cada una de ellas dará lugar a una variable lógica para SAT, que indica si esa acción se realiza en una etapa determinada del plan o no. Si sólo permitimos que se realice una acción en cada etapa (planificación clásica o STRIPS), al convertir el problema a SAT deberemos indicarlo con una serie de expresiones lógicas de forma que exista una sólo una variable lógica, de las que se refieren a las acciones, a valor verdadero, indicando que ésa es la acción a realizar en el plan en una etapa. Esto se denomina exclusión mutua de acciones, y puede expresarse con una conjunción de alternativas por pares de acciones:

$$\forall a_i \forall a_j / i \neq j, \quad \bigwedge \neg a_i \vee \neg a_j$$

Esta fórmula impide que se asignen 2 acciones. Esta expresión se incluye en cada etapa. Las cláusulas resultantes tendrían la forma, por ejemplo:

```
-move_room1_room2@0 V -move_room1_room3@0  
-move_room1_room2@1 V -move_room1_room3@1  
...  
-move_room1_room2@0 V -move_room1_room4@0  
-move_room1_room2@1 V -move_room1_room4@1  
...
```

Con esto se impiden incompatibilidades de acciones, pero la exclusión mutua de todas las acciones entre sí no permite ningún tipo de paralelismo de acciones.

Si se desea permitir que se realicen varias acciones a la vez (y esto puede ser interesante al reducir la longitud en pasos del plan resultante y presumiblemente el tiempo de cálculo de la solución del problema SAT) debe comprobarse que las acciones a realizar a la vez (en la misma etapa) son compatibles. Por ejemplo, un robot no puede ir a dos sitios a la vez. Suponiendo que se cumple en el estado de la etapa 1 los predicados:

```
ROBOT-AT(bot1, room1)  
PATH(room1, room2)
```

PATH(room1,room3)

...

En el plan podrían aparecer las acciones:

MOVE(bot1,room1,room2)@1

MOVE(bot1,room1,room3)@1

...

Podemos entender que esta situación no es posible, pero debe establecerse explícitamente en el problema SAT con una serie de expresiones que impidan que se asignen en la misma etapa acciones que no son compatibles.

Otras acciones sí pueden ser compatibles al no existir relación de incompatibilidad ni de interferencia entre ellas. En ese caso no se incluiría las cláusulas de exclusión mutua entre esas acciones. En el apartado 3.4. se analizan los tipos de incompatibilidad entre acciones con más detalle y se proponen alternativas.

3.3.4.7. Fase 7 – Codificar los nombres como Identificadores

Una vez realizado el grounding, para la conversión a problema SAT debe asignarse una variable booleana a cada acción y a cada predicado (instanciados). El valor verdadero en esas variables indicará que esa acción asociada se realizará en el plan, o que ese predicado asociado se cumple en un momento determinado durante la realización del plan. Las variables booleanas se identifican simplemente con un número, puesto que así resulta más eficiente para una programa de ordenador (que el empleo de etiquetas de texto). La relación entre esas variables booleanas y sus correspondientes acciones o predicados instanciados debe conservarse, y se guarda en una estructura de tabla de símbolos de doble entrada, de forma que es rápido acceder tanto por número de variable booleana como por etiqueta textual asignada a cada acción y predicado instanciado.

3.3.4.8. Fase 8 - Generación de expresiones lógicas SAT

El problema SAT es un problema de satisfacción de restricciones con variables booleanas, basado en operaciones lógicas AND, OR NOT.

Se debe convertir el problema de planificación a un problema SAT, y para ello se convierten los elementos de PDDL a unos conjuntos de expresiones lógicas, de forma que exista una relación y que la solución del SAT corresponda con una solución del problema de planificación.

Si bien hay varias formas de hacer esto, se ha considerado la transformación más usual, bastante compacta y correcta. Es decir, que si hay solución para el problema de planificación, el problema SAT lo encontrará. Si el SAT obtiene una solución, esa será una solución válida para el problema de planificación, y si SAT no encuentra ninguna solución, es seguro que no hay solución para el problema de planificación, al menos en ese horizonte de tiempo considerado. En todos los métodos de conversión del problema de planificación a SAT es necesario establecer un límite máximo de etapas o pasos o tiempo.

En el paso anterior se ha asignado una variable booleana a cada acción o predicado instanciado. En este paso se emplean ya esas variables para representar las acciones o predicados.

Los grupos de expresiones lógicas son:

- Init (estado inicial).
- Goal (estado objetivo).
- Causas de acciones.
- Efectos de las acciones.
- Axiomas marco.
- Exclusión de conflictos.
- Control de paralelismo.

Init

El estado inicial debe quedar completamente identificado. Se construye una conjunción lógica de forma que para todos los predicados que se cumplen en el estado inicial, se pondrá su variable booleana asociada a verdadero, y si no a valor falso (variable negada).

Esto se convierte en CNF a una cláusula por cada variable, y la variable aparecerá negada o no según el estado inicial del predicado que representa.

Estas cláusulas estarán etiquetadas con la etapa 0.

Goal

El estado final o de objetivo se indica con un conjunto de cláusulas, al igual que Init. En este caso se indican únicamente las variables booleanas que corresponde a los predicados que aparecen el elemento Goal del problema en PDDL.

Estas cláusulas deben etiquetarse con el número de la etapa final, que dependerá del horizonte de tiempo que se esté considerando en este problema.

Causa de Acciones

Son un conjunto de expresiones lógicas que indican que para realizar una acción, sus precondiciones deben cumplirse. En caso contrario no podría realizarse esa acción. Por ejemplo:

$$\text{mover}(\text{habit1}, \text{habit2})@i \rightarrow \text{esta_en}(\text{habit1})@i$$

Efecto de las acciones

Si se realiza una acción, sucederá algo en el mundo, modificando ciertos predicados. Esto se expresa con unas expresiones lógicas del tipo:

$$\text{accion}@i \rightarrow \text{efectos}@i+1$$

Indican que si una acción se realiza en una etapa, en la etapa siguiente nos encontraremos con que los efectos de la acción han ocurrido y han producido esos cambios. Los efectos pueden ser añadir un predicado, (un predicado se hace verdad), o quitarlo (se emplea aquí el signo '-' para indicar el símbolo de negación').

$$\text{mover}(\text{habit1}, \text{habit2})@i \rightarrow \begin{array}{l} \neg \text{esta_en}(\text{habit1})@i+1 \wedge \\ \text{esta_en}(\text{habit2})@i+1 \end{array}$$

Axiomas marco

Estos axiomas son expresiones lógicas que evitan que determinadas variables booleanas (que representan los predicados del estado del mundo) puedan tomar valores arbitrariamente y aparecer modelos accidentales (*unintended models*), los cuales no son soluciones válidas para la planificación. En un mundo determinista y no dinámico lo único que puede producir cambios en el estado del mundo son las acciones posibles. Esto debe incluirse explícitamente en el problema SAT para tenerlo en cuenta, si no, podrían dar lugar a soluciones que no serían correctas.

Hay dos formas típicas de establecer estas restricciones, una forma clásica (*classic frame axioms*) y otra forma denominada explicativa (*explanatory frame axioms*). En nuestro caso se emplea la forma explicativa, al resultar más compacta y flexible.

Estas condiciones se expresan mediante expresiones lógicas como la siguiente:

```
-esta_en(habit1)@i ^  
esta_en(habit1)@i+1 -> mover(habit1,habit2 )@i V  
mover(habit1,habit3)@i V ...
```

Si una variable (predicado) ha cambiado de valor, es porque necesariamente se ha realizado una acción en la etapa anterior, que produce ese cambio de valor.

Produciendo expresiones de ese tipo para todas las posibles acciones, se evitan los modelos accidentales.

Exclusión de conflictos

En la fase 6 se realizó un análisis de posibles incompatibilidades entre acciones, y se prepararon expresiones para evitar conflictos a causa de ello en un plan. Esas expresiones se reducen a CNF y se incluyen en el problema SAT.

Control del paralelismo

En nuestro caso se emplean técnicas para permitir paralelismo de acciones, pero si se quiere evitar esa posibilidad, es posible generar expresiones que eviten que se realice más de una acción en la misma etapa, mediante exclusión mutua de acciones. Este tipo de cláusulas se pueden generar opcionalmente en nuestro compilador SAT.

3.3.4.9. Fase 9 - Aplicar el número de etapa /tiempo

En la etapa anterior se construyeron una serie de expresiones lógicas y se convirtieron a notación CNF. Se dejó fuera el aspecto de la etapa o paso de tiempo a la que corresponde la acción para usar una técnica de copia y desplazamiento. Esta técnica permite generar el problema rápidamente a partir de la etapa anterior, y aporta una ventaja: una vez realizadas las etapas anteriores, se pueden generar el mismo problema pero con un número de etapas distinto sin necesidad de realizar las etapas anteriores nuevamente.

En esta etapa se añade la información de las etapas de tiempo a las cláusulas CNF.

La técnica consiste en crear un grupo de cláusulas general, que usan como referencias de tiempo un número de etapa relativo.

Hay 3 grupos de cláusulas según su procedencia:

- A. Estado inicial: se refieren a la etapa 0
- B. Objetivo o goal: se refieren a la última etapa del horizonte definido N.

- C. Resto de clausulas: las correspondientes a las precondiciones y efectos de las acciones y sus cambios sobre las etapas. Así como las restricciones por axiomas marco y resolución de conflictos.

Las clausulas de los grupos A y B se producen una sola vez, en el caso del grupo B se debe tener en cuenta cuál es el horizonte de tiempo, esto es, el número de etapas para el que se está buscando un plan.

Las clausulas del grupo C sin embargo deben repetirse para considerar la realización de acciones en unas etapas o en otras. Al generar estos grupos de clausulas se generaron de forma general, una sola vez, usando unos números de etapa genéricos: i , $i+1$. Pero las distintas acciones y el estado de los predicados deben concretarse para todas las etapas dentro del horizonte temporal. Por ejemplo, consideremos un horizonte temporal $N=10$, y la expresión lógica:

`mover_pos1_pos2@i -> -estaen_pos1@i+1 y estaen_pos2@i+1`

La codificación al aplicar el numero de etapa sería:

`mover_pos1_pos2@0 -> -estaen_pos1@1 y estaen_pos2@1`
`mover_pos1_pos2@1 -> -estaen_pos1@2 y estaen_pos2@2`
`mover_pos1_pos2@2 -> -estaen_pos1@3 y estaen_pos2@3`
`...`
`mover_pos1_pos2@9 -> -estaen_pos1@10 y estaen_pos2@10`

Cada una de esas variables lógicas (como `mover_pos1_pos2@1`) es una variable diferente, ya que corresponde a una etapa de tiempo distinta, y forma parte del estado total del problema.

La técnica aquí propuesta se basa en la copia y desplazamiento de numero de variables booleanas, conservando la estructura de las clausulas, para generar todas las expresiones lógicas necesarias para expresar el problema como SAT.

Mediante esta técnica, el horizonte temporal sólo afecta a esta última etapa, de forma que al usar SAT de forma repetida (con diferentes horizontes de tiempo) para buscar soluciones al problema de planificación, podemos realizar todas las etapas una sola vez, y después aplicar esta técnica para generar cuantas expresiones sean necesarias según el horizonte temporal.

El tamaño del problema SAT se incrementa al aumentar el espacio temporal. Cada etapa añadirá un numero de clausulas y de variables al problema.

En principio, esta técnica es compatible con el uso de un graphplan u otras técnicas para reducir el número de acciones posibles, por ejemplo debido a incompatibilidades entre acciones. Es decir, se puede aplicar un filtrado y eliminar algunas clausulas que se detecten como incompatibles o inaccesibles en cada etapa.

3.3.4.10. Fin de las Fases de conversión

Tras la última etapa, se obtiene un conjunto de expresiones lógicas, normalizadas en CNF, del problema SAT equivalente al problema de planificación. Cada acción y predicado, instanciados, se representa por una o más variables lógicas (dependiendo de la codificación empleada). El resultado se exporta a un fichero en formato DIMACS CNF. También puede exportarse a un formato binario. Es más eficiente sin embargo utilizar dentro del mismo programa algún método para invocar a un solver SAT (por ejemplo enlazar con una biblioteca de SAT), e invocar el solver a partir de estructuras de datos en C/C++. En cualquier caso, se genera el formato adecuado para poder usar un SAT solver (de varios disponibles). Tras la conversión, se mantienen las estructuras de datos tanto de PDDL como de la tabla de símbolos empleadas en la conversión a CNF, para poder deshacer la codificación SAT de la solución, y poder usar la como una solución expresada en predicados y acciones de PDDL.

3.4. Estudio de incompatibilidad entre acciones

Las acciones que son posibles realizar en un dominio pueden tener ciertas incompatibilidades de forma que no puedan realizarse a la vez, en una misma etapa. En algunos casos el dominio es tan restrictivo que sólo puede realizarse una acción en cada etapa. En otros dominios, sí es posible aprovechar paralelismo entre acciones, y eso puede suponer no solo una mejora en el plan obtenido, sino que puede ayudar a resolver el problema más rápidamente.

3.4.1. Tipos de incompatibilidad entre acciones

Hay varios tipos de incompatibilidad entre acciones, que pueden clasificarse según el tipo de predicados que generan esa posible incompatibilidad. En la Tabla 6, que se muestra a continuación, se realiza una clasificación de incompatibilidades basada en las precondiciones y efectos de las acciones. En la columna “Relación entre predicados” se distinguen entre predicados que aparecen negados o sin negar, y si son de distinto signo (es decir, un predicado aparece negado y el otro no). En la columna “Expresiones” se muestra la formula lógica general que debe cumplirse para producir ese tipo de incompatibilidad. En esa columna A_1 y A_2 se refieren a dos acciones distintas, y se usan precondiciones (directas o negadas, pre), y efectos que añaden (add) o eliminan (del). Ambas acciones pueden intercambiarse de lugar. El signo menos

se emplea para cambiar el signo de las precondiciones (de un predicado negado a sin negar, y viceversa).

Debe tenerse en cuenta que en PDDL las precondiciones negadas son opcionales, usando el requirement :negative-preconditions. Una forma de tratar las precondiciones negadas (la que usa Blackbox 2.0) es añadir un nuevo predicado por cada predicado negado en alguna acción. Así, en la acción A1:

```
(:action A1 (:parameters ?x - object)
  :precondition (and (not p1(x)) p2(x) )
  :effects (and (p1(x) p3(x) ) )
)
```

La precondición `p1` negada se convierte al nuevo predicado (sin negación) `not-p1`. Se deben añadir el nuevo predicado también a todos los efectos allí donde aparezca `p1` como efecto, para mantener la consistencia en el dominio.

La acción quedaría como (se señalan los cambios en otro color):

```
(:action A1 (:parameters ?x - object)
  :precondition (and (not-p1(x)) p2(x) )    ; conversion
  :effects (and (p1(x) p3(x)
    (not (not-p1(x)) )                      ; añadido
  )
)
```

De esta forma se eliminan los predicados con negación de las precondiciones.

3.4.2. Clasificación de incompatibilidad entre acciones

Tipo	Relación entre predicados	Compatibilidad	Expresiones
0	Distintos	Independientes	$\text{del}(A_1) \cap [\text{pre}(A_2) \cup \text{add}(A_2)] = \emptyset$ AND $\text{del}(A_2) \cap [\text{pre}(A_1) \cup \text{add}(A_1)] = \emptyset$
Entre precondiciones			
I	Iguales	Compatibles	$\text{pre}(A_1) \cap \text{pre}(A_2) \neq \emptyset$
II	Distinto signo	Excluyentes	$\text{pre}(A_1) \cap \neg \text{pre}(A_2) \neq \emptyset$
Entre efectos			
III	Iguales	Condición de carrera posible	$\text{add}(A_1) \cap \text{add}(A_2) \neq \emptyset$ $\text{del}(A_1) \cap \text{del}(A_2) \neq \emptyset$
IV	Distinto signo	Inconsistencia	$\text{add}(A_1) \cap \text{del}(A_2) \neq \emptyset$ $\text{del}(A_1) \cap \text{add}(A_2) \neq \emptyset$
Entre precondición y efecto			
V	Iguales	Sobre escritura	$\text{pre}(A_1) \cap \text{add}(A_2) \neq \emptyset$ $\neg \text{pre}(A_1) \cap \text{del}(A_2) \neq \emptyset$
VI	Distinto signo	Interferencia	$\text{pre}(A_1) \cap \text{del}(A_2) \neq \emptyset$ $\neg \text{pre}(A_1) \cap \text{add}(A_2) \neq \emptyset$

Tabla 6. Tabla de incompatibilidad entre acciones.

Tipo 0. Acciones independientes

Este tipo indica independencia entre acciones y no existe incompatibilidad entre ellas. Por tanto, dos acciones independientes pueden realizarse a la vez en una misma etapa sin ningún problema. También pueden secuenciarse, es decir, realizarse una detrás de la otra en cualquier orden en la misma etapa.

Tipo I. Precondiciones iguales

El hecho de existir precondiciones iguales en dos acciones distintas no establece ninguna incompatibilidad. En los ejemplos, las acciones de coger cajas, o mover, todas tienen como precondiciones que el robot está en una determinada habitación, pero no por ello son incompatibles.

Tipo II. Precondiciones iguales con distinto signo

En este tipo de relación entre predicados, en una acción aparece un predicado de forma directa (sin negar) y en la otra acción aparece negado. Para que se produzca esta situación, en primer lugar debe permitirse predicados negados, con la opción de PDDL :negative-preconditions.

En este ejemplo el único predicado repetido es p1, que aparece con distinto signo en las acciones:

```
(:action A1 (:parameters ?x - object)
  :precondition (and (p1(x) p2(x) )
  :effects (and (p3(x))
)
(:action A2 (:parameters ?x - object)
  :precondition (and (not (p1(x)) p4(x))
  :effects (and (p5(x))
)
```

En este caso, sólo una de esas acciones podrá realizarse en una determinada etapa, ya que las precondiciones pueden ser verdaderas o falsas, pero no ambas cosas, y por tanto una de esas acciones no validará sus precondiciones. Pueden considerarse incompatibles, por ese motivo, y generar cláusulas para contemplarlo, pero no es necesario incluirlas a efectos de validez del plan.

Sin embargo estas cláusulas pueden ser incluidas como pistas para que el solver SAT disponga de una serie de restricciones adicionales y eliminar posibilidades en la búsqueda de solución, al coste de incluir una serie de cláusulas adicionales al problema.

Tipo III. Entre efectos con igual signo

Dos acciones pueden tener entre sus efectos el mismo predicado. Aunque eso no produce una inconsistencia, sí puede indicar una interferencia entre acciones. En este trabajo, este tipo de situaciones puede detectarse, y la generación de cláusulas de exclusión mutua entre acciones de este tipo es opcional.

Ejemplo:

```
(:action A1 (:parameters ?x - object)
  :precondition (and (p1(x))
  :effects (and (p2(x))
)
(:action A2 (:parameters ?x - object)
  :precondition (and (p3(x))
  :effects (and (p2(x))
)
```

Al instanciarlas podríamos obtener las acciones y la precondition de sus efectos:

```
A1(Ob1) -> p2(Ob1)
A2(Ob1) -> p2(Ob1)
```

Tipo IV. Entre efectos de distinto signo

Dos acciones que tienen en sus respectivos efectos el mismo predicado pero en un caso lo establece y en la otra acción lo anula, se consideran incompatibles. Ese tipo de situación crea una interferencia entre ambas acciones, y es imposible determinar cuál es el estado resultante de aplicar las acciones (al menos en planificación clásica).

Ejemplo:

```
(:action A1 (:parameters ?x - object)
  :precondition (and (p1(x))
  :effects (and (p2(x))
)
```

```
(:action A2 (:parameters ?x - object)
  :precondition (and (p3(x))
  :effects (and (not p2(x))
)
```

Al instanciarlas podríamos obtener las acciones y la precondition de sus efectos:

```
A1(Ob1) -> p2(Ob1)
A2(Ob1) -> not p2(Ob1)
```

En el ejemplo mostrado, ambas acciones son incompatibles por tener el mismo predicado, con distinto signo. Aplicar ambas acciones a la vez dejaría al predicado p2 en un estado indeterminado en la etapa siguiente.

Tipo V. Entre una precondition y un efecto con igual signo

En este caso una acción define un predicado como efecto, que coincide con una precondition de la otra acción. El predicado debe estar cumpliéndose en esa etapa, de lo contrario no se podría aplicar la acción que lo incluye como precondition.

Ejemplo:

```
(:action A1 (:parameters ?x - object)
  :precondition (and (p1(x) p2(x))
  :effects ( p3(x) )
)
(:action A2 (:parameters ?x - object)
  :precondition (and (p4(x) p5(x) )
  :effects ( p1(x) )
)
```

Este caso es parecido al tipo III, puesto que se trata de sobrescribir el valor de un predicado, no en cambiar su valor.

Tipo VI. Entre una precondition y un efecto con distinto signo

Si una acción tiene definida una precondition, que se cumple en ese instante (si no, no podría aplicarse), y otra acción anula esa precondition, pueden considerarse incompatibles.

Ejemplo:

```
(:action A1 (:parameters ?x - object)
  :precondition (and (p1(x) p2(x))
  :effects ( p3(x) )
)
(:action A2 (:parameters ?x - object)
  :precondition (and (p4(x) p5(x) )
  :effects ( (not p1(x)) )
)
```

Este puede ser el caso, por ejemplo, en el que un robot pretende ir a dos lugares a la vez, la acción MOVE tendrá una precondition ROBOT-AT(room1) que es invalidada por la otra acción MOVE.

Otros tipos de incompatibilidad

Los posibles tipos de incompatibilidad descritos anteriormente pueden averiguarse partir del análisis del problema de planificación expresado en PDDL. Si se quieren expresar otro tipo de incompatibilidad entre acciones o entre predicados, deben usarse otros mecanismos para indicarlo explícitamente en el problema. Por ejemplo pueden usarse predicados para ello, o usar axioms (debe indicarse el pre-requisito :domain-axioms) u otras opciones de PDDL. En cualquier caso, el programador que escriba los ficheros PDDL debe tenerlo en cuenta al definir el modelo.

Una forma sencilla de incluir exclusión entre acciones es definir un predicado para indicar el conflicto, a modo de semáforo, de forma que las acciones comprueben en sus precondiciones el predicado, y en los efectos producirán ese predicado para marcar la situación como posible conflicto.

Implementación de detección de incompatibilidades entre acciones

Para programar la detección de acciones incompatibles y la posterior generación de cláusulas de exclusión mutua entre acciones, se empleó una matriz de relación entre acciones. Esta técnica se basa en construir una matriz en la que se anotan los tipos de inconsistencias entre cada par de acciones

instanciadas. Una vez construida, se recorre la matriz generando expresiones de exclusión mutua entre aquel par de acciones que tenga una anotación de inconsistencia.

Como alternativa, se desarrolló la generación de expresiones de exclusiones mutuas entre acciones sin usar la matriz de relación. Esto puede dar lugar a la anotación repetida de algunos tipos de inconsistencia (puede haber inconsistencias de tipo II y tipo VI entre dos acciones, por ejemplo). Para evitar esa duplicación se implementó también una opción para comprobar la existencia de una misma cláusula en el conjunto de cláusulas generadas para el problema de planificación.

Uso de la detección de incompatibilidades para codificación híbrida

Una vez detectadas incompatibilidades entre acciones, es posible generar conjuntos de acciones, incompatibles entre sí, de forma que esas acciones se codifiquen mediante un esquema bitwise. Eso permite reducir el número de variables lógicas empleadas en la compilación SAT, con la ventaja de no comprometer el posible paralelismo entre acciones.

Para obtener cuales son los mejores conjuntos de acciones, se emplea un método basado en grafos. Se construye un grafo en el que cada nodo representa una acción, y las aristas indicarán una relación de incompatibilidad entre dos acciones. Los grupos de acciones que formen un clique son candidatos a ser codificados como bitwise.

3.5. Codificación híbrida de acciones

Al transformar un problema de planificación a una codificación SAT, se pueden usar muchos tipos de codificaciones, pero normalmente se asigna una variable booleana para cada predicado instanciado (grounded), en cada etapa del plan, así como una variable booleana para cada acción (instanciada) posible. Como se ha visto, hay formas alternativas más compactas, como usar varias variables booleanas para codificar una serie de combinaciones de estados posibles. En el caso de emplear esa codificación más compacta, impedimos que pueda realizarse una acción en paralelo con otra, puesto que las variables booleanas codifican una sola de las acciones posibles.

Pueden utilizarse codificaciones híbridas, esto es, algunas acciones pueden representarse como variables booleanas y otras acciones, agruparse para utilizar una codificación en cadenas de bits (bitwise encoding) que es más compacta. Es interesante usar bitwise encoding entre aquellos predicados y acciones que son incompatibles entre sí, dado el modelo del dominio, puesto que al ser incompatibles no pueden darse al mismo tiempo (en la misma etapa). De esta forma se permite paralelismo entre acciones compatibles (el paralelismo entre

acciones incompatibles no puede existir), pero también se aprovecha una notación compacta cuando ese paralelismo no es posible.

Para detectar acciones pueden ser usados en algún grupo de bit encoding, se propone la siguiente técnica basada en grafos:

1. Instanciar predicados, cada uno se representa con un nodo (o vértice) del grafo G (inicialmente vacío).
2. Detectar incompatibilidad entre cada par de predicados. Cada incompatibilidad se representa como una arista o enlace entre nodos del grafo.
3. Identificar una partición del grafo, de forma que se minimice el numero de bits necesarios para su codificación. El numero de bits necesarios viene dado por:

$$bits = \log_2(num\ vertices\ de\ la\ particion\ i)$$

Y el número mínimo que se busca viene dado por:

$$total\ bits = \sum_i ceil(\log_2\ verts_i)$$

3.5.1. Ejemplo

El siguiente ejemplo muestra un problema sencillo de planificación en PDDL en el que se describe la instanciación de las acciones y cómo se construye el grafo de incompatibilidades. En este dominio puede haber varios robots (en el ejemplo se usan dos) que pueden moverse de una habitacion a otra, si existe un camino (path) que las una. Los robots pueden coger cajas y dejarlas, pero solo pueden llevar una en cada momento. Se consideran las restricciones siguientes de forma implícita, por la existencia de conflictos entre acciones:

1. Si un robot coge una caja, ningún otro robot puede cogerla al mismo tiempo.
2. Un robot puede mover una sola vez en cada paso.

A continuación se muestran las definiciones de estas acciones en PDDL. Se emplea STRIPS con la adición de las opciones *typing* y *negative-preconditions*, para que sea una definición más clara y más compacta, además de reducir el número de predicados y acciones en la instanciación.

3. ENTORNO PARA PLANIFICACIÓN AUTOMÁTICA

```
(:action move :parameters ( ?t - robot ?x - room ?y - room )
:precondition ( and ( ROBOT-AT ?t ?x )
                    ( PATH ?x ?y ) )
:effect (and (ROBOT-AT ?t ?y)
             (not (ROBOT-AT ?t ?x)) ) )
(:action load :parameters (?t - robot ?r - room ?b - box)
:precondition (and (BOX-AT ?b ?r )
                  (ROBOT-AT ?t ?r )
                  ( not (ROBOT-LOADED ?t) ) )
:effect (and (ROBOT-LOADED ?t)
             (not (BOX-AT ?b ?r ))
             (ROBOT-CARRIES ?t ?b ) ) )
```

En el fichero de problema, los estados inicial y objetivo se definen así:

```
(:objects bot1 bot2 - robot
          rm1 rm2 - room
          bx1 bx2 - box)
(:init
 (ROBOT-AT bot1 rm1) (ROBOT-AT bot2 rm1)
 (BOX-AT bx1 rm1) (BOX-AT bx2 rm2)
 (PATH rm1 rm2) (PATH rm2 rm1) )
(:goal
 (and (ROBOT-AT bot1 rm2) (ROBOT-AT bot2 rm2 )
      (BOX-AT bx1 rm2) (BOX-AT bx2 rm2) ) )
```

Las acciones instanciadas se muestran en la siguiente tabla (Tabla 7). En el proceso de instanciación se han tenido en cuenta los predicados estáticos y en el ejemplo se ha detectado que el predicado PATH es un predicado estático, y se aplica directamente en la instanciación de acciones. Por ejemplo las acciones `move(bot1,rm1,rm1)`, y `move(bot1,rm2,rm2)` no aparecen en la tabla porque se han demostrado no aplicables en ningún caso (sus precondiciones `PATH(rm1,rm1)` y `PATH (rm2,rm2)` no se cumplen nunca).

1	move_bot1_rm1_rm2	5	load_bot1_rm1_bx1	9	load_bot2_rm1_bx1
2	move_bot1_rm2_rm1	6	load_bot1_rm1_bx2	10	load_bot2_rm1_bx2
3	move_bot2_rm1_rm2	7	load_bot1_rm2_bx1	11	load_bot2_rm2_bx1
4	move_bot2_rm2_rm1	8	load_bot1_rm2_bx2	12	load_bot2_rm2_bx2

Tabla 7. Lista de acciones generadas en el dominio del ejemplo.

Según las reglas de compatibilidad, la acción 1 tiene como uno de los efectos el predicado `¬bot1_rm1`, mientras que la acción 2 tiene `bot1_rm1`. Esto

supone un conflicto, puesto que ambas acciones aplican un efecto sobre el mismo predicado, y de distinto signo. Por tanto estas dos acciones se marcan como incompatibles. Lo mismo sucede con las acciones 3 y 4. Entre las acciones 1 y 5 se detecta otro tipo de incompatibilidad: la precondition `ROBOT-AT_rm1` de una de las acciones es invalidada por la otra acción, al estar entre sus efectos con negación. Sucede algo parecido con las acciones 5 y 6 en este caso con el predicado `ROBOT-LOADED`. Este predicado está presente en el dominio precisamente para evitar que un robot pueda cargar más de una caja al mismo tiempo. Por último, las acciones 5 y 9 también entran en conflicto, debido al predicado `BOX-AT_bx1_rm`.

El grafo de incompatibilidades G_c se muestra en el diagrama de la Fig. 23. En este caso tan sencillo es fácil apreciar visualmente los dos conjuntos de acciones que forman clique. Aplicando el algoritmo BCLQ propuesto se obtienen los dos conjuntos $K_1 = \{1, 2, 5, 6, 7, 8\}$ and $K_2 = \{3, 4, 9, 10, 11, 12\}$. El subconjunto K_1 tiene 6 elementos y por tanto se precisan al menos 3 bits para realizar una codificación bitwise de esos elementos.

Fig. 23. Grafo de incompatibilidades G_c del ejemplo.

3. ENTORNO PARA PLANIFICACIÓN AUTOMÁTICA

En Tabla 8 se indica una posible codificación par el conjunto K_1 usando 3 variables booleanas x_1, x_2, x_3 . Para el conjunto K_2 se necesitan otras 3 variables distintas x_4, x_5, x_6 , como se muestra en la Tabla 9.

#		Acción	x_1	x_2	x_3
0		no action	0	0	0
1	1	move_bot1_rm1_rm2	0	0	1
2	2	move_bot1_rm2_rm1	0	1	0
3	5	load_bot1_rm1_bx1	0	1	1
4	6	load_bot1_rm1_bx2	1	0	0
5	7	load_bot1_rm2_bx1	1	0	1
6	8	load_bot1_rm2_bx2	1	1	0
7		not used	1	1	1

Tabla 8. Codificación bitwise de las acciones del conjunto K1.

#		Acción	x_4	x_5	x_6
0		no action	0	0	0
1	3	move_bot2_rm1_rm2	0	0	1
2	4	move_bot2_rm2_rm1	0	1	0
3	9	load_bot2_rm1_bx1	0	1	1
4	10	load_bot2_rm1_bx2	1	0	0
5	11	load_bot2_rm2_bx1	1	0	1
6	12	load_bot2_rm2_bx2	1	1	0
7		not used	1	1	1

Tabla 9. Codificación bitwise de las acciones del conjunto K2.

En este caso los conjuntos han quedado divididos en función de un parámetro, (el robot), pero puede no ser así.

En esta codificación las acciones se expresarán como la conjunción de las variables usadas en la codificación, de forma directa o negada según el valor

asignado en la codificación. Por ejemplo, la acción 7 `load_bot1_rm2_bx1` se expresaría en la forma:

$$x_1 \wedge \overline{x_2} \wedge x_3$$

Y por ejemplo la acción 9 `load_bot2_rm1_bx1` se expresaría:

$$\overline{x_4} \wedge x_5 \wedge x_6$$

Al usar esta técnica, se obtiene un mejor ahorro en variables lógicas cuanto mayor es el grupo de acciones o predicados que son incompatibles entre sí y por tanto es el número sobre el que se aplica el logaritmo en base 2. Sin embargo debe tenerse en cuenta que siempre se emplean un numero entero de bits, y por tanto da igual si el grupo tiene 18 o 30 acciones, por ejemplo, en ambos casos se precisan 5 bits en la codificación.

La incompatibilidad entre dos predicados se obtiene a partir de la definición de las acciones.

3.5.2. Resultados

El uso de la codificación bitwise reduce significativamente el número de variables lógicas empleadas en el problema SAT convertido. En la siguiente tabla se muestran unos ejemplos de distintos dominios. Las características de estos dominios y problemas se comentan en el Capítulo 5. Nótese que esta codificación no limita en absoluto el uso de acciones en paralelo, al contrario que una codificación bitwise estándar. En la siguiente tabla (Tabla 10) se indica el numero de nodos en el grafo (acciones instanciadas), el número de bits necesarios para la codificación con el método propuesto (bits), el número de cliques encontrados (groups), y el número de elementos en el mayor clique encontrado (max). Por último, se indica la medida de la reducción del número de variables de nuestro método comparado con los bits necesarios con regular encoding.

3. ENTORNO PARA PLANIFICACIÓN AUTOMÁTICA

Problema	Grafo			Subconjuntos			
	<i>n</i>	<i>m</i>	<i>p</i>	grupos	max	bits	reducción
<i>robot_C_Pm01</i>	544	14528	0,037	52	72	174	68.0%
<i>robot_C_Pg02</i>	2508	163536	0,015	190	216	593	76.4%
<i>aipslog01</i>	405	5682	0,071	68	14	182	55.1%
<i>aipslog04</i>	618	8577	0,072	100	11	292	52.7%
<i>blocksworld01</i>	441	57771	0,007	19	86	70	84.1%
<i>blocksworld02</i>	891	203139	0,004	31	132	108	87.9%
<i>hanoi-4</i>	126	5286	0,023	7	37	27	78.6%
<i>hanoi-8</i>	572	79892	0,007	11	101	58	89.9%
<i>slidetile 1</i>	192	5840	0,032	7	40	38	80.2%
<i>slidetile 3</i>	192	5840	0,032	7	40	38	80.2%

Tabla 10. Comparativa de codificación de varios problemas de planificación y reducción del número de bits necesarios empleando la codificación híbrida propuesta.

Los resultados muestran un nivel significativo de reducción del número de variables, lo cual tiene un impacto en el tamaño del problema SAT a resolver. La reducción del número de variables es más importante en aquellos problemas en los que existe gran densidad de incompatibilidad entre acciones. En esos casos las ventajas de poder realizar varias acciones a la vez no son tan importantes, porque las restricciones del problema no lo permite, y darán lugar a planes con unos horizontes de tiempo más largos, por eso en esos problemas reducir el número de variables lógicas empleadas tiene mayor impacto en el solver SAT. Esto significa que la codificación híbrida propuesta funciona mejor en los problemas en tienen más necesidad de ello. El algoritmo BCLQ propuesto es muy rápido, inferior a 1 segundo en estos ejemplos, y no tiene negativo impacto en el proceso de compilación a SAT.

3.6. SAT Solver

En el esquema general se emplea un planificador basado en SAT, por tanto un módulo del sistema es un SAT solver (Fig. 24).

Fig. 24. Diagrama del SAT solver en el contexto del sistema.

En el sistema planteado puede usarse un SAT solver externo, o bien un solver interno, compilado como biblioteca y accesible directamente por el compilador SAT. Como solvers internos se emplearon miniSAT y un SAT solver desarrollado por el grupo de investigación. Este solver se expuso en un congreso en 2008 [58].

En dicho solver se proponen 2 nuevos algoritmos para SAT: BB-SAT y BBP-SAT, basados en el uso de bits en paralelo aprovechando las características de bitstrings y la capacidad de las ALUs de las CPUs actuales.

Una novedad de dichos algoritmos es el uso de *bitstrings sparse*, de forma que los bitstrings pueden compactarse dependiendo de la densidad de unos (1s) existentes en el bitstring, puesto que en muchos casos la densidad de unos es baja. Esto produce un gran ahorro en memoria.

En este SAT solver se almacenan los valores asignados a los literales mediante bitstrings y matrices de bits. Puesto que se almacenan literales directos y negados, también se desarrolló una opción para almacenar los literales de forma que los literales negados quedasen alineados en palabras del sistema. Esa técnica dejaba algunos bits sin uso, consumiendo un poco más de memoria, pero permitía realizar operaciones lógicas entre literales negados y no negados con una mejora de rendimiento, al estar situados con la misma alineación.

3.7. Analizador de soluciones de planificación

En el esquema general del sistema de planificación y simulación se incluye una propuesta de módulos para el análisis de planes obtenidos de los problemas de planificación, y posible generación de problemas alternativos para su evaluación (Fig. 25).

Fig. 25. : Diagrama de los módulos relacionados con el análisis y generación de problemas y planes.

Este analizador obtiene por un lado la descripción del dominio y problema de PDDL. El planificador proporciona un plan válido (o bien no encuentra un plan porque no existe) y el simulador comprueba su validez y características (numero de pasos, uso de acciones, etc.) y de ahí pueden extraerse algunos datos, entre otros:

- Número de pasos.
- Existencia o no de acciones en paralelo.
- Acciones involucradas en la solución, y acciones no usadas.
- Orden de las acciones, y veces que aparecen.
- Objetos involucrados en la solución, y objetos no usados.
- Predicados involucrados en la solución, y predicados no usados.
- Predicados modificados en la solución.

Este tipo de información se puede añadir a la base de conocimiento. Con datos agregados de distintos planes (y de sus problemas respectivos), puede usarse la base de conocimiento para generar un nuevo problema con ese dominio, para probar más posibilidades, testear alternativas. Otra posibilidad es cambiar el

3. ENTORNO PARA PLANIFICACIÓN AUTOMÁTICA

dominio (quitar o poner acciones por ejemplo), para comprobar que sucede en esos escenarios alternativos, y sacar conclusiones con los resultados obtenidos al planificar.

4. ENTORNO DE SIMULACIÓN

4.1. Introducción

Como parte de este proyecto, se diseñó y desarrolló un simulador para planificación, que integrase perfectamente con el planificador y la arquitectura propuesta (Fig. 26).

Al simulador (y sus herramientas adicionales) se le denominó Logic Planning Simulator (LPS). Este simulador permite preparar problemas y casos de simulación, invocar al planificador y obtener un resultado, es decir, un plan que puede entonces verse en una ventana 3D, donde objetos 3D representan objetos del dominio de planificación, y las acciones del plan pueden mostrarse en esa representación 3D mediante movimientos o cambios de estado de los objetos.

Inicialmente se realizó una versión de simulador para ordenadores PC pero fácilmente adaptable a otros entornos (por usar librerías estándar multiplataforma), llamada “desktop”. Posteriormente se propuso y se realizó una versión de simulador “online” que se lanza dentro de un navegador de Internet, aprovechando entonces la capacidad multiplataforma de los navegadores y extendiendo las funcionalidades para facilitar el trabajo colaborativo de personas que estén creando casos de planificación usando PDDL.

Fig. 26. Diagrama de contexto del Simulador.

Ambos tienen funcionalidades similares, aunque en la versión online aporta además conexión con un servidor para planificación, un editor estructurado de PDDL (PDDL Builder) y opciones para colaborar en línea.

Este entorno se presentó en el congreso internacional IADIS en Junio de 2015.

4.2. Simulador de escenarios 3D

4.2.1. Funcionalidades generales

El entorno de simulación acepta por un lado la descripción del dominio y del problema de planificación, expresado en PDDL. Por otra parte necesita un fichero en el que se asocian a los objetos de PDDL, una representación como objetos 3D, indicando la malla (la información de la geometría del objeto) y material (indica el color o textura del objeto). También se debe indicar cómo se van a representar visualmente la realización de cada una de las acciones

definidas en PDDL (por ejemplo: moviendo un objeto de una posición a otra, o cambiando su representación 3D para mostrar un cambio de estado)

El entorno muestra un listado de los objetos presentes en la escena 3D (y por tanto en el caso de planificación) y permite examinar las propiedades de los objetos (su posición, malla, estado, etc.). Muestra asimismo una representación 3D del escenario, con los objetos que existen en ese entorno.

El entorno puede pedir a un planificador que resuelva el problema de planificación, y tras ello, obtener el resultado del planificador. El planificador usado inicialmente es el planificador basado en SAT, que se explica en capítulo 3 de este documento. Posteriormente se adaptó el simulador, para que admita distintos planificadores, según las necesidades o preferencias del usuario, e incluso comparar resultados de distintos planificadores ante el mismo problema.

Este resultado, se interpreta y se muestra como una animación (movimiento de objetos o cambios de color o de geometría de los objetos 3D afectados por las acciones) en la que pueden verse los efectos de cada etapa del plan al realizarlo. Opcionalmente puede realizarse la simulación paso a paso, para permitir al usuario observar detenidamente los efectos de cada acción en cada etapa.

4.2.2. Integración entre Simulador y otros módulos

El simulador sirve como interfaz de usuario para preparar un escenario, resolverlo y visualizar el plan. Para ello, el simulador tiene acceso a leer y editar los ficheros de PDDL, y puede lanzar un planificador, recogiendo el resultado. En el caso de utilizar el planificador SAT propuesto, el simulador tiene también acceso a los símbolos empleados en la codificación. Por otro lado, el simulador realiza una simulación del plan, de forma que comprueba su validez y ese resultado se puede exportar como información para otros módulos.

4.2.3. Simulador - desktop

El simulador en versión desktop (es decir, aplicación para un ordenador de sobremesa, sin usar navegador web como host) se desarrolló en C++ usando las librerías Qt (para interfaz de usuario multiplataforma) y MR-core (librería para el uso en robótica).

Fig. 27. Captura del interfaz del simulador.

En la captura de pantalla (Fig. 27) se puede ver el interfaz de usuario del simulador, en la que destacan varias zonas de la aplicación:

- Una barra de menús, típica de las aplicaciones basadas en ventanas.
- Una gran zona dedicada a la visualización 3D del escenario de la simulación.
- Un listado de objetos presentes en el caso
- Una zona donde se muestran propiedades del objeto que se tenga seleccionado
- Abajo, una zona para mostrar mensajes al usuario, bien acerca de avisos o errores, o bien para mostrar el progreso de la simulación.

Ejemplo de escenario XML:

```
<object class="Elemento" name="" >
  <property name="MODEL3D" type="QString" value="cube10.3ds" />
  <property name="ROOMID" type="QString" value="" />
  <property name="X" type="double" value="0" />
  <property name="Y" type="double" value="0" />
  <property name="Z" type="double" value="0" />
  <property name="ANGLE" type="double" value="0" />
  <property name="ID" type="QString" value="sq_1_1" />
</object>
```

```

    <object class="ActionList" >
    </object>
</object>
<object class="Elemento" name="" >
    <property name="MODEL3D" type="QString" value="" />
    <property name="ROOMID" type="QString" value="" />
    <property name="X" type="double" value="0" />
    <property name="Y" type="double" value="0" />
    <property name="Z" type="double" value="0" />
    <property name="ANGLE" type="double" value="0" />
    <property name="ID" type="QString" value="Global_Funcs" />
    <object class="ActionList" >
        <object class="pyFuncion" name="" >
            <property name="MODEL3D" type="QString" value="" />
            <property name="ROOMID" type="QString" value="" />
            <property name="X" type="double" value="0" />
            <property name="Y" type="double" value="0" />
            <property name="Z" type="double" value="0" />
            <property name="ANGLE" type="double" value="0" />
            <property name="ID" type="QString" value="" />
            <property name="NAME" type="QString" value="move" />
            <property name="PARAMETERS" type="QStringList" value="" />
            <property name="FUNCTIONSRC" type="QString"
value="def runn (who,from_,to_): unset_at(who,from_)
                                set_at(who,to_) " />
            <property name="PRECONDITIONSRC" type="QString"
value="def precondition (who,from_,to_):
return (adj(from_,to_) and not (pit(to_) ) and at(who,from_) )" />
            <object class="ActionList" >
            </object>
        </object>
        <object class="pyFuncion" name="" >
            <property name="MODEL3D" type="QString" value="" />
            <property name="ROOMID" type="QString" value="" />
            <property name="X" type="double" value="0" />
            <property name="Y" type="double" value="0" />
            <property name="Z" type="double" value="0" />
            <property name="ANGLE" type="double" value="0" />
            <property name="ID" type="QString" value="" />
            <property name="NAME" type="QString" value="take" />
            <property name="PARAMETERS" type="QStringList" value="" />
            <property name="FUNCTIONSRC" type="QString"
value="def runn (who,what,where): set_have(who,what)
                                unset_at(what,where) " />
            <property name="PRECONDITIONSRC" type="QString"
value="def precondition (who,what,where):
return (at(who,where) and at(what,where) )" />
            <object class="ActionList" >
            </object>
        </object>
    </object>

```

Como puede verse en el formato anterior, el fichero consiste en una definición de cada uno de los elementos indicando su representación 3D, un identificador que lo asocia a un objeto de PDDL, y cada elemento puede tener una serie de acciones a las que responde, realizando un fragmento de código (o script) en lenguaje Python. Mediante ese script se pueden definir animaciones y los comportamientos sobre los objetos 3D que se deseen, a fin de mostrar el resultado de la aplicación de las etapas del plan.

La mayoría de esos datos son modificables usando el editor incluido en el simulador, facilitando la creación de escenarios.

4.2.4. Simulador - Online

El simulador Online se planteó para dar más difusión al proyecto de simulador para planificación. Aprovecha las características de los navegadores de Internet modernos (Chrome, Firefox, etc.) para funcionar en modo multiplataforma, y facilitar un desarrollo que incluya la conexión con servidores externos al simulador. Se desarrolló usando CSS, HTML5 y JavaScript, para el contenido local al usuario (Fig. 28). En el lado del servidor se emplea en parte LAMP (Linux, Apache, MySQL y PHP, aunque otra parte se desarrolló en C++ (el planificador en sí) para una mayor eficiencia y para reutilizar el trabajo realizado anteriormente. A pesar de estar pensado para funcionar con conexión a internet a un servidor, el simulador no necesita conexión para la mayoría de sus funciones.

Fig. 28. Captura del interfaz del simulador online.

Formato JSON de definición del escenario (sim_extern.js):

```
custom=
{
  package1:{
    position:[10,0,10], rotation: [0, 0, 0], model: "box.obj",
    conditions:{ at_package1_pgh_po:{position:[20,0, 20]}}}},
  package2:{ position:[20,0,10], rotation: [0, 0, 0],model: "box2.obj"},
  package3:{ position:[30,0,10], rotation: [0, 0, 0],},
  airplanel:{
    position:[20,0,30], rotation: [0, 0, 0], model: "pirpir.obj",
  },
  airplane2:{
    position:[30,0,30], rotation: [0, 0, 0], model: "pirpir.obj",
  },
  pgh:{},
  bos:{},
  la:{},
  "pgh-truck":{
    position:[10,0,15], rotation: [0, 0, 0], model: "truck.obj",
  },
  "bos-truck":{
    position:[10,0,20], rotation: [0, 0, 0], model: "truck.obj",
  },
  "la-truck":{
    position:[10,0,30], rotation: [0, 0, 0], model: "truck.obj",
  },
  "pgh-po":{},
  "bos-po":{},
  "la-po":{},
  "pgh-central":{},
  "bos-central":{},
  "la-central":{},
  "pgh-airport":{},
  "bos-airport":{},
  "la-airport":{},
};
```

En el ejemplo anterior pueden localizarse fácilmente los distintos objetos que forman parte del escenario, por su nombre, y sus características (posición, orientación, modelo 3D que lo representa, etc.)

Resultado de la planificación en JSON de un ejemplo (fichero plan.js):

```
plan=[
  "flyairplane(\"airplanel\", \"pgh_airport\", \"la_airport\");\
  loadtruck1(\"package3\", \"pgh_truck\", \"pgh_po\");\
  loadtruck1(\"package1\", \"pgh_truck\", \"pgh_po\");\
  ",
  "flyairplane(\"airplanel\", \"la_airport\", \"pgh_airport\");\
  drivetruck1(\"bos_truck\", \"bos_po\", \"bos_airport\", \"bos\");\
  loadtruck1(\"package2\", \"pgh_truck\", \"pgh_po\");\
  ",
  "flyairplane(\"airplane2\", \"pgh_airport\", \"bos_airport\");\
  flyairplane(\"airplanel\", \"pgh_airport\", \"la_airport\");\
  drivetruck1(\"la_truck\", \"la_po\", \"la_airport\", \"la\");\
  drivetruck1(\"pgh_truck\", \"pgh_po\", \"pgh_airport\", \"pgh\");\
  ",
  "flyairplane(\"airplane2\", \"bos_airport\", \"pgh_airport\");\
  flyairplane(\"airplanel\", \"la_airport\", \"pgh_airport\");\
  drivetruck1(\"la_truck\", \"la_airport\", \"la_central\", \"la\");\
  drivetruck1(\"bos_truck\", \"bos_airport\", \"bos_po\", \"bos\");\
  unloadtruck2(\"package1\", \"pgh_truck\", \"pgh_airport\");\
  unloadtruck1(\"package3\", \"pgh_truck\", \"pgh_airport\");\
  unloadtruck1(\"package2\", \"pgh_truck\", \"pgh_airport\");\
  ",
];
```

4. ENTORNO DE SIMULACIÓN

```
"drivetruck1(\"la_truck\", \"la_central\", \"la_po\", \"la\");\n\n\"drivetruck1(\"bos_truck\", \"bos_po\", \"bos_central\", \"bos\");\nloadairplane(\"package3\", \"airplane1\", \"pgh_airport\");\nloadairplane(\"package2\", \"airplane2\", \"pgh_airport\");\nloadairplane(\"package1\", \"airplane1\", \"pgh_airport\");\n\n\n\"flyairplane(\"airplane2\", \"pgh_airport\", \"la_airport\");\nflyairplane(\"airplane1\", \"pgh_airport\", \"bos_airport\");\n\n\"drivetruck1(\"bos_truck\", \"bos_central\", \"bos_po\", \"bos\");\n\"drivetruck1(\"pgh_truck\", \"pgh_airport\", \"pgh_central\", \"pgh\");\n\n\n\"drivetruck2(\"bos_truck\", \"bos_po\", \"bos_airport\", \"bos\");\n\"drivetruck1(\"la_truck\", \"la_po\", \"la_airport\", \"la\");\n\"drivetruck1(\"pgh_truck\", \"pgh_central\", \"pgh_airport\", \"pgh\");\nunloadairplane(\"package3\", \"airplane1\", \"bos_airport\");\nunloadairplane(\"package2\", \"airplane2\", \"la_airport\");\nunloadairplane(\"package1\", \"airplane1\", \"bos_airport\");\n\n\n\"flyairplane(\"airplane2\", \"la_airport\", \"pgh_airport\");\nflyairplane(\"airplane1\", \"bos_airport\", \"pgh_airport\");\nloadtruck2(\"package3\", \"bos_truck\", \"bos_airport\");\nloadtruck2(\"package2\", \"la_truck\", \"la_airport\");\nloadtruck1(\"package1\", \"bos_truck\", \"bos_airport\");\n\n\n\n\"flyairplane(\"airplane1\", \"pgh_airport\", \"la_airport\");\n\"drivetruck3(\"la_truck\", \"la_airport\", \"la_po\", \"la\");\n\"drivetruck3(\"pgh_truck\", \"pgh_airport\", \"pgh_po\", \"pgh\");\n\"drivetruck1(\"bos_truck\", \"bos_airport\", \"bos_po\", \"bos\");\n\n\n\n\"unloadtruck1(\"package3\", \"bos_truck\", \"bos_po\");\n\"unloadtruck1(\"package2\", \"la_truck\", \"la_po\");\n\"unloadtruck1(\"package1\", \"bos_truck\", \"bos_po\");\n\n\n1;
```

En la estructura de la solución puede observarse que admite paralelismo entre acciones en la misma etapa. Las acciones a realizar en cada etapa se agrupan. Cada etapa se representa con una cadena de texto que contiene las acciones de esa etapa. En JSON se representa como un vector (array) de cadenas de texto, cada entrada del vector corresponde a una etapa del plan.

4.3. Editor Estructurado de PDDL (PDDL Builder)

4.3.1. Introducción

Además de realizar la simulación de planes para casos en PDDL, se planteó la creación de un editor de PDDL dentro del mismo entorno. En el capítulo 3, se comentaron algunos entornos (no muchos) ya existentes para facilitar la escritura de PDDL. El editor propuesto se basa en la estructura de PDDL para mostrar y editar los elementos de PDDL de forma fácil y organizada, evitando algunos errores típicos que se producen al usar editores de texto plano. La novedad de este editor estriba en mostrar y editar los distintos elementos del lenguaje PDDL de forma estructurada, mediante componentes del interfaz de usuario específicos (Fig. 29), facilitando la creación de escenarios PDDL y

evitando algunos de los errores típicos en la escritura de código fuente, PDDL en nuestro caso.

Fig. 29. Interfaz general del editor PDDL Builder.

4.3.2. Funcionalidades del Editor

El editor PDDL Builder permite realizar un gran número de operaciones usando botones, listas desplegables y otros elementos de interfaz gráfica de usuario, reduciendo la necesidad de teclear y facilitando la especificación de problemas en PDDL.

Lista de funciones:

- Importar PDDL, dominio y problema.
- Exportar PDDL, dominio y problema.
- Salvar un dominio y problema en espacio de usuario en la nube.
- Obtener un problema, y su dominio asociado del espacio de usuario en la nube.

- Crear un nuevo problema, indicando un dominio.
- Crear un nuevo dominio.
- Eliminar un problema del espacio de usuario.
- Eliminar un dominio en el espacio de usuario.
- Añadir una nueva acción a un dominio.
- Eliminar una acción existente de un dominio.
- Modificar una acción existente de un dominio.
- Añadir un predicado al dominio.
- Eliminar un predicado del dominio.
- Añadir un tipo de objeto en un dominio.
- Eliminar un tipo de objeto de un dominio.
- Añadir / eliminar objetos de un problema.
- Crear o modificar el estado inicial de un problema.
- Crear o modificar las condiciones de objetivo de un problema.
- Establecer las restricciones (requirements) del dominio.
- Establecer el orden de las acciones y predicados.

El editor permite la creación y modificación de escenarios en PDDL. Se trata de un editor estructurado, no de texto plano como es habitual tanto en PDDL como en la mayoría de los lenguajes de programación. Mantiene una lista de los elementos de PDDL que se han definido, a los que se puede acceder de forma rápida. Puesto que en PDDL hay elementos que pueden usarse o no dependiendo de las distintas opciones de *requirements* que se indiquen (por ejemplo :strips, :typing, etc.), el editor puede configurarse para admitir esas opciones y adaptarse a las que se han establecido para ese dominio (Fig. 30). Por ejemplo, si se ha definido :typing, entonces el usuario podrá crear y modificar tipos en el escenario. Si no, no tendrá esa posibilidad, evitando escribir elementos no válidos.

Fig. 30. Sección de Requirements del PDDL Builder.

El editor permite importar ficheros PDDL de dominio y de problema. Al importar estos ficheros, convierte internamente la descripción PDDL a un árbol de elementos PDDL, mostrándola en el árbol de la izquierda (explorador del

4. ENTORNO DE SIMULACIÓN

escenario), y las acciones y predicados etc. , se mostrarán en la zona de edición de forma estructurada para permitir la edición de esos elementos.

Es posible que al importar ficheros PDDL, el editor encuentre errores o inconsistencias. En ese caso el editor indica los errores encontrados, y en muchos casos puede obtener un escenario (aunque incompleto) para permitir editar ese escenario y solucionar los errores.

El escenario sobre el que se está trabajando puede exportarse a ficheros PDDL, generando un fichero de dominio y otro fichero de problema.

En el editor se diferencia el fichero de dominio y el fichero de problema, siguiendo la filosofía de PDDL. En una zona a la izquierda se muestra una lista de elementos del dominio, y de elementos del problema que se está editando.

El editor permite añadir predicados, acciones, objetos y otros elementos de forma fácil, mediante una opción en el menú. Las opciones Add Action, Add Predicate, Add Type, permiten crear un elemento del tipo indicado.

La opción del menú Add Predicate permite definir un predicado en el dominio. Se mostrará una ventana con varios campos para poder definir el nombre y los parámetros del predicado, indicando su tipo si procede (Fig. 31). El editor mostrará una lista desplegable cuando puede ayudar al usuario a introducir algún nombre, con una lista de alternativas posibles.

The screenshot shows the 'Predicates' section of the PDDL Builder interface. It contains four predicate entries, each with a name, a '+' button to add it, and an 'x' button to remove it. Below the name are input fields for arguments, with a dropdown menu showing suggestions.

Predicate Name	Buttons	Arguments
ROBOT-AT	+, x	?r - robot, ?r - room
ROBOT-LOADED	+, x	
PATH	+, x	?r1 - room, ?r2 - room
BOX-AT	+, x	?b - box, ?r - room

Fig. 31. Detalle del Interfaz de usuario de PDDL Builder, en la edición de predicados.

Al usar Add Action, se añade una acción al dominio, mostrando una ventana con los campos que componen la acción, y el usuario puede cambiar el nombre, los parámetros, las precondiciones y los efectos (**¡Error! No se encuentra el origen de la referencia.**).

The screenshot shows a software interface titled 'Actions' with a '+ Add Action' button. Below the button, two action templates are displayed in a list-like structure. The first action is 'MOVE', with a parameter template *?t - robot ?r1 - room ?r2 - room*. Its preconditions are 'ROBOT-AT ?r1' and 'PATH ?r1 ?r2'. Its effects are 'ROBOT-AT ?r2' and 'NOT ROBOT-AT ?r1'. The second action is 'LOAD', with a parameter template *?t - robot ?r - room ?b - box*. Its preconditions are 'ROBOT-AT ?r', 'BOX-AT ?b ?r', and 'NOT ROBOT-LOADED ?t'. Its effects are 'ROBOT-LOADED ?t', 'NOT BOX-AT ?b ?r', and 'ROBOT-CARRIES ?t ?b'.

Action	Parameters	Preconditions	Effects
MOVE	<i>?t - robot ?r1 - room ?r2 - room</i>	ROBOT-AT ?r1 PATH ?r1 ?r2	ROBOT-AT ?r2 NOT ROBOT-AT ?r1
LOAD	<i>?t - robot ?r - room ?b - box</i>	ROBOT-AT ?r BOX-AT ?b ?r NOT ROBOT-LOADED ?t	ROBOT-LOADED ?t NOT BOX-AT ?b ?r ROBOT-CARRIES ?t ?b

Fig. 32. Sección de acciones en PDDL Builder.

Pulsando con el ratón se puede seleccionar uno de esas partes de la acción, y el usuario puede escribir en esa parte. Al escribir los parámetros, el usuario puede establecer el tipo de cada parámetro usando una lista desplegable que muestra los tipos de objetos ya definidos en el dominio. Al escribir las precondiciones, el usuario introducirá nombres de predicados y los parámetros de cada uno. El editor ayuda al usuario mostrando debajo una lista desplegable de predicados (cuando está escribiendo el predicado), o una lista de los parámetros definidos ya en esa acción, cuando está ya escribiendo parámetros de un predicado. Esto permite al usuario escribir más rápidamente la acción, al mostrarle el editor las alternativas válidas más frecuentes (Fig. 33).

:strips :types
+

Move	?posx - <i>position</i> ?posy - <i>position</i>	
pre:	<div style="border: 1px solid #000; padding: 5px; display: inline-block;"> robo </div>	
(and)	<div style="border: 1px solid #000; padding: 5px; display: inline-block;"> robot-at robot-loaded empty path </div>	
eff:	-empty (?posy)	robot-at(?posy) empty(?posx)

Move	?posx - <i>position</i> ?posy - <i>position</i>	
pre:	robot-at (?posx)	
(and)	empty (?posy) path(?posx ?posy)	
eff:	-robot-at (?posx) -empty (?posy)	robot-at(?posy) empty(?posx)

Fig. 33. Diseño del IU de PDDL Builder, edición de precondiciones.

Durante la escritura de una acción, alguno de los campos puede aparecer en rojo, indicando algún tipo de error. Los errores más comunes son:

- El nombre de la acción está repetido.
- Se está usando un tipo de objeto que no está definido.
- Se está usando un nombre de predicado que no se ha definido.
- Se está usando un nombre de objeto o parámetro que no se ha definido.
- Se ha escrito un predicado incompleto o con un número de parámetros incorrectos.

Al definir una acción, es posible escribir un predicado que aún no existe. Es decir, crear una nueva acción y rellenar sus campos, incluido un nombre de predicado que no está declarado. Al terminar de escribir la acción el nombre de predicado sin definir se mostrará en color rojo. Posteriormente se puede declarar el predicado con sus parámetros. El editor lo detectará y si la acción es correcta entonces, eliminará el resalte en color rojo del predicado. Esta forma respeta la forma de trabajo de un usuario típico al crear escenarios PDDL.

Al usar la opción Add Type, se creará un nuevo tipo de objeto, indicando el nombre y el tipo base. Esta opción sólo está disponible si se ha definido :typing en los requirements.

Estas facilidades son parecidas a las que ofrecen los editores modernos de los lenguajes de programación (auto completar, mostrar información de parámetros de las funciones que se están escribiendo, coloreado por sintaxis, etc.) pero de forma estructurada, de forma que además se evitan ciertos errores de incoherencia o de sintaxis al escribir el problema en PDDL.

En el diagrama (Fig. 34) se muestra la ventana del árbol del proyecto. Un proyecto consta de un fichero de dominio y otro fichero de problema. PDDL admite teóricamente más de un fichero para el dominio y para el problema pero raramente se ve ese caso. (Podría ponerse 1 dominio y más de 1 problema para el mismo dominio...)

En el árbol se muestra un esquema del escenario, es decir se identifica el PDDL y divide en sus partes: Para el dominio aparecen los Tipos, si los hay, constantes, predicados y acciones (y funciones si hubiera). Para el problema se divide en Objetos, estado inicial (predicados que lo compone) y objetivo(s).

Los tipos se muestran en el árbol del proyecto, a la izquierda. Todos los objetos en PDDL tienen un padre llamado 'object', y de ese tipo derivan el resto de tipos definidos por el usuario. Es posible cambiar el nombre de un tipo, o cambiar el tipo padre del que deriva.

Fig. 34. Sección del árbol de proyecto de PDDL Builder.

Al cambiar el nombre de un tipo, el editor actualiza también los tipos usados en la definición de predicados y acciones. Si se cambia la ubicación en la jerarquía de tipos de alguno, el usuario debe comprobar que los elementos PDDL que usan ese tipo sigan siendo válidos.

4.4. Espacio de usuario en la nube

Siguiendo la tendencia a ofrecer a los usuarios servicios “en la nube”, se consideró la posibilidad de ofrecer un espacio a los usuarios que quieran emplear los editores de PDDL y el Simulador, y se decidieron una serie de servicios que podrían integrar una oferta interesante a aquellos usuarios interesados en desarrollar problemas de planificación en PDDL.

Los servicios más importantes son:

- Acceso, protegido por contraseña, al espacio de usuario.
- Almacenamiento de ficheros de dominio y problemas en la nube.
- Acceso a un repositorio publico de problemas PDDL.
- Posibilidad de publicar ficheros de dominio y problemas en el repositorio, conservando la autoría original de los mismos.
- Acceso directo desde los ficheros almacenados para utilizarlos con el editor.
- Posibilidad de lanzar un planificador (de entre varios) en el servidor, para buscar soluciones a uno o varios problemas por orden del usuario.
- Envío de mensajes entre usuarios, y buzón de mensajes recibidos. Gestión de mensajes (enviar, leer, borrar, agrupar en carpetas).
- Lista de usuarios “amigos” o contactos.
- Foro de mensajes públicos.

Ahí puede crear sus PDDLs, verlos y editarlos, invocar a planificadores online, obtener una solución y simular.

El usuario puede acceder mediante un nombre de usuario y contraseña, y puede almacenar ficheros PDDL de dominio o de problemas (ligados a un dominio), puede editarlos directamente en el navegador usando uno de los editores, y obtener un análisis previo de corrección.

Puede acceder a un repositorio público de problemas, y el usuario puede poner como públicos sus problemas PDDL y dominios.

Los editores pueden tener un analizador de PDDL para detectar inconsistencias, o cosas inútiles/no usadas en el problema, o generar pistas (clausulas) para facilitar la tarea al SAT Solver.

Con estos servicios se completa la oferta a los usuarios, pudiendo entonces desde un mismo interface basado en navegador web, disponer de editor, planificador (en servidor), simulador y repositorio propio y comunitario de problemas PDDL.

5. ESCENARIOS PARA LA SIMULACIÓN Y PRUEBAS

5.1. Descripción de escenarios

Dado que este trabajo se ha realizado en el marco de planificación automática aplicada a robots, se han creado unos problemas basados en dominios relacionados con planificación de alto nivel para robots.

Previamente, para la realización de las pruebas de desarrollo de este proyecto se emplearon algunos problemas en PDDL ya existentes que han sido usado por otros investigadores en planificación automática y/o han formado parte del a colección de problemas en las competiciones de planificadores realizadas junto con los congresos ICAPS en diversos años. Puesto que estos son problemas ya conocidos y con soluciones disponibles, también han servido como comprobación de la corrección del planificador y del simulador. Como escenarios originales, se crearon varias colecciones de problemas, ambientados en el uso de robots, con dominios denominados A, B, C y E.

5.1.1. Escenarios ya existentes

Los escenarios utilizados se basan en STRIPS, son los siguientes:

- Logística: Consiste en diferentes vehículos que deben transportar cajas de unos lugares a otros. Admite paralelismo.

- Barman: Se trata de problemas de combinación de productos y útiles, para preparar recetas. Es parecido a los problemas de montaje.
- Puzzles: Las torres de Hanoi o el 8-puzzle son ejemplos sencillos, consisten en encontrar una secuencia para completar el puzzle. Generalmente no permiten paralelismo, o lo permiten de forma muy limitada.

5.1.2. Problemas de logística aips

Estos problemas se basan en distintos vehículos que pueden transportar objetos, y se desea obtener un plan para realizarlo. Está especificado en STRIPS. Permiten paralelismo. El dominio de es:

```
(define (domain logistics-strips)
  (:requirements :strips)
  (:predicates (truck ?t) (location ?l) (airport ?a)
               (airplane ?p) (city ?c) (obj ?o)
               (at ?x ?y) (in ?x ?y) (in-city ?x ?y))

  (:action LoadTruck1
    :parameters (?o ?t ?l)
    :precondition (and (obj ?o) (truck ?t) (location ?l)
                      (at ?t ?l) (at ?o ?l))
    :effect (and (in ?o ?t) (not (at ?o ?l))))

  (:action LoadTruck2
    :parameters (?o ?t ?a)
    :precondition (and (obj ?o) (truck ?t) (airport ?a)
                      (at ?t ?a) (at ?o ?a))
    :effect (and (in ?o ?t) (not (at ?o ?a))))

  (:action LoadAirplane
    :parameters (?o ?p ?a)
    :precondition (and (obj ?o) (airplane ?p) (airport ?a)
                      (at ?o ?a) (at ?p ?a))
    :effect (and (in ?o ?p) (not (at ?o ?a))))

  (:action UnloadTruck1
    :parameters (?o ?t ?l)
    :precondition (and (obj ?o) (truck ?t) (location ?l)
                      (at ?t ?l) (in ?o ?t))
    :effect (and (at ?o ?l) (not (in ?o ?t))))

  (:action UnloadTruck2
    :parameters (?o ?t ?a)
    :precondition (and (obj ?o) (truck ?t) (airport ?a)
                      (at ?t ?a) (in ?o ?t))
    :effect (and (at ?o ?a) (not (in ?o ?t))))

  (:action UnloadAirplane
```

5. ESCENARIOS PARA LA SIMULACIÓN Y PRUEBAS

```
:parameters (?o ?p ?a)
:precondition (and (obj ?o) (airplane ?p)      (airport ?a)
                  (in ?o ?p) (at ?p ?a))
:effect (and (at ?o ?a) (not (in ?o ?p))))

(:action DriveTruck1
 :parameters (?t ?l1 ?l2 ?c)
 :precondition (and (truck ?t) (location ?l1) (location ?l2)
 (city ?c)
                  (at ?t ?l1) (in-city ?l1 ?c) (in-city ?l2 ?c))
 :effect (and (at ?t ?l2) (not (at ?t ?l1))))

(:action DriveTruck2
 :parameters (?t ?l ?a ?c)
 :precondition (and (truck ?t) (location ?l) (airport ?a)
 (city ?c)
                  (at ?t ?l) (in-city ?l ?c) (in-city ?a ?c))
 :effect (and (at ?t ?a) (not (at ?t ?l))))

(:action DriveTruck3
 :parameters (?t ?a ?l ?c)
 :precondition (and (truck ?t) (airport ?a) (location ?l)
 (city ?c)
                  (at ?t ?a) (in-city ?a ?c) (in-city ?l ?c))
 :effect (and (at ?t ?l) (not (at ?t ?a))))

(:action FlyAirplane
 :parameters (?p ?a1 ?a2)
 :precondition (and (airplane ?p) (airport ?a1) (airport ?a2)
                  (at ?p ?a1))
 :effect (and (at ?p ?a2) (not (at ?p ?a1))))
)
```

La solución de unos de los problemas es:

Domain: logistics-strips, Problem: log001

```
( 0 ) loadtruck1.package1.pgh-truck.pgh-po@0
( 0 ) loadtruck1.package2.pgh-truck.pgh-po@0
( 0 ) loadtruck1.package3.pgh-truck.pgh-po@0
( 0 ) drivetruck1.bos-truck.bos-po.bos-central.bos@0
( 1 ) drivetruck1.pgh-truck.pgh-po.pgh-airport.pgh@1
( 1 ) drivetruck1.bos-truck.bos-central.bos-airport.bos@1
( 1 ) drivetruck1.la-truck.la-po.la-airport.la@1
( 1 ) flyairplane.airplane1.pgh-airport.bos-airport@1
( 2 ) unloadtruck1.package1.pgh-truck.pgh-airport@2
( 2 ) unloadtruck1.package2.pgh-truck.pgh-airport@2
( 2 ) unloadtruck2.package3.pgh-truck.pgh-airport@2
( 2 ) flyairplane.airplane1.bos-airport.pgh-airport@2
```

```
( 3 ) loadairplane.package1.airplane1.pgh-airport@3
( 3 ) loadairplane.package2.airplane2.pgh-airport@3
( 3 ) loadairplane.package3.airplane1.pgh-airport@3
( 3 ) drivetruck1.bos-truck.bos-airport.bos-po.bos@3
( 3 ) drivetruck1.la-truck.la-airport.la-central.la@3
( 4 ) drivetruck1.pgh-truck.pgh-airport.pgh-po.pgh@4
( 4 ) drivetruck1.la-truck.la-central.la-airport.la@4
( 4 ) drivetruck2.bos-truck.bos-po.bos-airport.bos@4
( 4 ) flyairplane.airplane1.pgh-airport.bos-airport@4
( 4 ) flyairplane.airplane2.pgh-airport.la-airport@4
( 5 ) unloadairplane.package1.airplane1.bos-airport@5
( 5 ) unloadairplane.package2.airplane2.la-airport@5
( 5 ) unloadairplane.package3.airplane1.bos-airport@5
( 6 ) loadtruck1.package1.bos-truck.bos-airport@6
( 6 ) loadtruck2.package2.la-truck.la-airport@6
( 6 ) loadtruck2.package3.bos-truck.bos-airport@6
( 6 ) flyairplane.airplane2.la-airport.pgh-airport@6
( 7 ) drivetruck1.bos-truck.bos-airport.bos-po.bos@7
( 7 ) drivetruck1.la-truck.la-airport.la-po.la@7
( 7 ) flyairplane.airplane1.bos-airport.la-airport@7
( 8 ) unloadtruck1.package1.bos-truck.bos-po@8
( 8 ) unloadtruck1.package2.la-truck.la-po@8
( 8 ) unloadtruck1.package3.bos-truck.bos-po@8
```

9 predicates, 10 actions.

Grounded predicates: 1320

Grounded actions: 405

Time: 50.578s

5.1.3. Escenario A

El primer escenario original consiste en un robot, que debe llevar cajas identificadas de unos lugares (habitaciones) a otros (Fig. 35) Es un problema parecido al de logística, que es un problema empleado en las competiciones de ICAPS, pero en este caso está restringido a un solo vehículo de transporte (el robot).

Fig. 35. Ilustración para ejemplo de escenario A.

Acciones:

- Mover (de una habitación origen a otra de destino). Debe existir un camino directo entre ambas habitaciones.
- Coger (caja). El robot debe estar en la misma habitación que la caja para poder cogerla. Además, el robot no puede llevar otra caja ya cargada.
- Dejar (caja). El robot deposita la caja que está acarreando en la habitación en la que se encuentra.

Predicados:

- Robot-en(habitación), identifica la posición del robot, debe ser única.
- Caja-en(caja, habitación), establece la ubicación de una determinada caja en una habitación. La posición de una caja debe ser única. Se permite un número indeterminado de cajas en una habitación.
- Robot-cargado-con(caja): Indica la caja que está acarreando el robot.
- Camino(habitación, habitación), indica la existencia de un camino entre 2 habitaciones. Se considera que el camino es unidireccional, y por tanto lo normal es que si existe un camino entre la habitación r1 y la r2, también lo exista entre r2 y r1, pero deben definirse ambos predicados.

Dado este dominio, se definieron varios problemas, de diferente complejidad, por el número de habitaciones, caminos y cajas, y también por la diferencia entre el estado inicial y el objetivo, lo cual resulta en un mayor número de pasos necesarios para resolver el problema.

Definición en PDDL del dominio A:

```
; (c) Cristobal Tapia 2015
(define (domain robot_domain_A)
  (:requirements :typing :negative-preconditions)
  (:types room box) ; no case sensitive
  (:predicates
    (ROBOT-AT ?r - room) ; robot is in room
    (ROBOT-LOADED) ; BOOLEAN
    (BOX-AT ?b - box ?r - room) ; box can only be on 1 place
    (PATH ?x ?y - room)
    (ROBOT-CARRIES ?b - box)
  )

  (:action move :parameters (?x - room ?y - room)
    :precondition (and (ROBOT-AT ?x)
      (PATH ?x ?y))
    :effect (and (ROBOT-AT ?y)
      (not (ROBOT-AT ?x))
    )
  )

  (:action load :parameters (?r - room ?b - box)
    :precondition (and (BOX-AT ?b ?r)
      (ROBOT-AT ?r)
      (not (ROBOT-LOADED))
    )
    :effect (and (ROBOT-LOADED)
      (not (BOX-AT ?b ?r))
      (ROBOT-CARRIES ?b)
    )
  )

  (:action unload :parameters (?r - room ?b - box)
    :precondition (and (ROBOT-AT ?r)
      (ROBOT-LOADED)
      (ROBOT-CARRIES ?b)
    )
    :effect (and (not (ROBOT-LOADED))
      (BOX-AT ?b ?r)
      (not (ROBOT-CARRIES ?b))
      (not (ROBOT-LOADED))
    )
  )
)
```

Ejemplo de problema sencillo para el dominio A:

```
; (c) Cristobal Tapia 2015
(define (problem robot_problem_A_P01)
  (:domain robot_domain_A)
  (:objects   rm1 rm2 rm3 rm4 - room
              bx1 bx2 bx3 bx4 - box)

  (:init
    (ROBOT-AT rm1)
    (BOX-AT bx1 rm1) (BOX-AT bx2 rm2)
    (BOX-AT bx3 rm3) (BOX-AT bx4 rm4)
    (PATH rm1 rm2) (PATH rm1 rm3) (PATH rm1 rm4) ; 1 --- 2
    (PATH rm2 rm1) (PATH rm2 rm3) (PATH rm2 rm4) ; | \  / |
    (PATH rm3 rm1) (PATH rm3 rm2) (PATH rm3 rm4) ; | x  |
    (PATH rm4 rm1) (PATH rm4 rm2) (PATH rm4 rm3) ; | /  \ |
    ; 3 --- 4
  )
  (:goal
    (and (ROBOT-AT rm3) (not (ROBOT-AT rm1) )
         (BOX-AT bx2 rm4)
    )
  )
)
```

Solución:

```
( 0 ) move.rm1.rm2@0
( 1 ) load.rm2.bx2@1
( 2 ) move.rm2.rm4@2
( 3 ) unload.rm4.bx2@3
( 4 ) move.rm4.rm3@4
```

5 steps, 5 actions.

5 predicados, 3 acciones,

Grounded predicates: 41

Grounded actions: 44

Ejemplo de problema sencillo para el dominio A (problema 3):

```
; (c) Cristobal Tapia 2015
(define (problem robot_problem_A_P03)
  (:domain robot_domain_A)
  (:objects   rm1 rm2 - room
              bx1 bx2 bx3 bx4 - box)

  (:init
    (ROBOT-AT rm1)
    (BOX-AT bx1 rm1) (BOX-AT bx2 rm2)
    (BOX-AT bx3 rm1) (BOX-AT bx4 rm1)
    (PATH rm1 rm2)           ; 1 --- 2
    (PATH rm2 rm1)
  )
  (:goal
    (and (ROBOT-AT rm2) (not (ROBOT-AT rm1) )
         (BOX-AT bx1 rm2) (BOX-AT bx2 rm2)
         (BOX-AT bx3 rm2) (BOX-AT bx4 rm2)
    )
  )
)
```

Solución:

```
( 0 ) load.rm1.bx4@0
( 1 ) move.rm1.rm2@1
( 2 ) unload.rm2.bx4@2
( 3 ) move.rm2.rm1@3
( 4 ) load.rm1.bx3@4
( 5 ) move.rm1.rm2@5
( 6 ) unload.rm2.bx3@6
( 7 ) move.rm2.rm1@7
( 8 ) load.rm1.bx1@8
( 9 ) move.rm1.rm2@9
(10 ) unload.rm2.bx1@10
```

11 steps, 11 actions

5 predicates, 3 actions.

Grounded predicates: 19

Grounded actions: 18

Ejemplo de problema medio para el dominio A:

```

; (c) Cristobal Tapia 2016

(define (problem robot_problem_A_Pm01)
  (:domain robot_domain_A)
  (:objects   rm1 rm2 rm3 rm4 rm5 rm6 rm7 rm8 - room
              bx1 bx2 bx3 bx4 bx5 bx6 bx7 bx8 - box)
  (:init
    (ROBOT-AT rm1)
    (BOX-AT bx1 rm1) (BOX-AT bx2 rm2)
    (BOX-AT bx3 rm3) (BOX-AT bx4 rm4)
    (BOX-AT bx5 rm5) (BOX-AT bx6 rm6)
    (BOX-AT bx7 rm7) (BOX-AT bx8 rm8)
    ; Paths:
    ; 1--2--3--4
    ; |      |
    ; 5--6--7--8
    (PATH rm1 rm2) (PATH rm1 rm5 )
    (PATH rm2 rm1) (PATH rm2 rm3 )
    (PATH rm3 rm2) (PATH rm3 rm4 ) (PATH rm3 rm7)
    (PATH rm4 rm3)
    (PATH rm5 rm1) (PATH rm5 rm6)
    (PATH rm6 rm5) (PATH rm6 rm7)
    (PATH rm7 rm6) (PATH rm7 rm8) (PATH rm7 rm3)
    (PATH rm8 rm7)
  )
  ;collect all boxes and put them in rm8
  (:goal
    (and (ROBOT-AT rm1)
          (BOX-AT bx1 rm8) (BOX-AT bx2 rm8)
          (BOX-AT bx3 rm8) (BOX-AT bx4 rm8)
          (BOX-AT bx5 rm8) (BOX-AT bx6 rm8)
          (BOX-AT bx7 rm8) (BOX-AT bx8 rm8)
        )
  )
)

```

Solución :

(0) load(rm1 bx1)	(24) move(rm4 rm3)
(1) move(rm1 rm2)	(25) move(rm3 rm7)
(2) move(rm2 rm3)	(26) move(rm7 rm8)
(3) move(rm3 rm7)	(27) unload(rm8 bx4)
(4) move(rm7 rm8)	(28) move(rm8 rm7)

(5) unload(rm8 bx1)	(29) move(rm7 rm6)
(6) move(rm8 rm7)	(30) move(rm3 rm5)
(7) move(rm7 rm3)	(31) load(rm5 bx5)
(8) move(rm3 rm2)	(32) move(rm5 rm6)
(9) load(rm2 bx2)	(33) move(rm6 rm7)
(10) move(rm2 rm3)	(34) move(rm7 rm8)
(11) move(rm3 rm7)	(35) unload(rm8 bx5)
(12) move(rm7 rm8)	(36) move(rm8 rm7)
(13) unload(rm8 bx2)	(37) move(rm7 rm6)
(14) move(rm8 rm7)	(38) load(rm6 bx6)
(15) move(rm7 rm3)	(39) move(rm6 rm7)
(16) load(rm3 bx3)	(40) move(rm7 rm8)
(17) move(rm3 rm7)	(41) unload(rm8 bx6)
(18) move(rm7 rm8)	(42) move(rm8 rm7)
(19) unload(rm8 bx3)	(43) load(rm7 bx7)
(20) move(rm8 rm7)	(44) move(rm7 rm8)
(21) move(rm7 rm3)	(45) unload(rm8 bx7)
(22) move(rm3 rm4)	
(23) load(rm4 bx4)	46 steps, 46 actions

5.1.4. Escenario B

Este escenario se basa en el anterior, consisten en un robot cuya misión principal será trasladar cajas. Añadimos unos elementos nuevos, las puertas, que pueden estar en los caminos que unen unas habitaciones con otras. Las puertas pueden estar cerradas o abiertas y se crean una llaves que permiten abrir las puertas cerradas (equivale a quitar un obstáculo). Las llaves son objetos que pueden estar en una habitación. El robot puede tener llaves, y cogerlas de la habitación, y usarlas para abrir la puerta que corresponda.

En este escenario se ha considerado que los caminos son bidireccionales, y basta con definir un camino (path) entre una habitación y otra para que el robot pueda moverse en un sentido u otro. Para ello se han creado 2 acciones para mover, en vez de una, para contemplar ambos sentidos de movimiento.

Definición en PDDL del dominio B:

```
(define (domain robot_domain_B_b)
  (:requirements :typing)           ;domain robot without or-op
  in preconditions
  (:types room box key)           ; no case sensitive
  (:predicates
    (ROBOT-AT ?r - room)           ; robot is in room
    (ROBOT-LOADED)                 ; BOOLEAN
    (BOX-AT ?b - box ?r - room)    ; can only be on 1 place
    (PATH ?x ?y - room)           ; a path is bidirec.
    (ROBOT-CARRIES ?b - box)       ; the robot carries this box
    (CLOSED-DOOR ?x ?y - room)
    ; key k opens the path between x and y
    (KEY ?k - key ?x ?y - room )
    (ROBOT-HAS-KEY ?k - key )     ; robot has a key k
    (KEY-AT-ROOM ?k - key ?r - room) ; key is on room r
  )
  (:action move-OD :parameters (?orig - room ?dest - room)
    :precondition ( and ( ROBOT-AT ?orig )
                        ( PATH ?orig ?dest )
                        ( not ( CLOSED-DOOR ?orig ?dest ) )
                        ( not ( CLOSED-DOOR ?dest ?orig ) )
    )
    :effect (and ( ROBOT-AT ?dest )
                 (not ( ROBOT-AT ?orig ) ) )
  )
  (:action move-DO :parameters (?orig - room ?dest - room)
    :precondition ( and ( ROBOT-AT ?orig )
                        ( PATH ?dest ?orig )
                        ( not ( CLOSED-DOOR ?orig ?dest ) )
                        ( not ( CLOSED-DOOR ?dest ?orig ) )
    )
    :effect (and ( ROBOT-AT ?dest )
                 (not ( ROBOT-AT ?orig ) ) )
  )
  (:action open-S :parameters ( ?side ?other - room ?k - key)
    :precondition ( and ( ROBOT-AT ?side )
                        ( CLOSED-DOOR ?side ?other)
                        ( KEY ?k ?side ?other)
                        ( ROBOT-HAS-KEY ?k )
    )
    :effect (and ( not (CLOSED-DOOR ?side ?other)) )
  )
  (:action open-O :parameters ( ?side ?other - room ?k - key)
    :precondition ( and ( ROBOT-AT ?other )
                        ( CLOSED-DOOR ?side ?other)
                        ( KEY ?k ?side ?other)
                        ( ROBOT-HAS-KEY ?k )
    )
    :effect (and ( not (CLOSED-DOOR ?side ?other)) )
  )
  (:action take-key :parameters ( ?r - room ?k - key )
    :precondition (and ( ROBOT-AT ?r)
```

```

        ( KEY-AT-ROOM ?k ?r)
      )
      :effect ( and ( ROBOT-HAS-KEY ?k )
                    ( not (KEY-AT-ROOM ?k ?r )) )
    )
    (:action load :parameters (?r - room ?b - box)
      :precondition (and (BOX-AT ?b ?r )
                         (ROBOT-AT ?r )
                         ( not (ROBOT-LOADED) ) )
      :effect (and (ROBOT-LOADED)
                  (not (BOX-AT ?b ?r ))
                  (ROBOT-CARRIES ?b ) )
    )
    (:action unload :parameters (?r - room ?b - box)
      :precondition (and (ROBOT-AT ?r )
                         (ROBOT-LOADED)
                         (ROBOT-CARRIES ?b ) )
      :effect (and (not (ROBOT-LOADED))
                  (BOX-AT ?b ?r )
                  (not (ROBOT-CARRIES ?b ))
                  (not (ROBOT-LOADED)) )
    )
  )
)

```

Un problema ejemplo de ese dominio:

```

; (c) Cristobal Tapia 2016
(define (problem robot_problem_B_P01)
  (:domain robot_domain_B_b)
  (:objects   rm1 rm2 rm3 - room
              bx1 bx2 bx3 - box
              ke1 ke2 - key)

  (:init
    (ROBOT-AT rm1)
    (BOX-AT bx1 rm1) (BOX-AT bx2 rm2) (BOX-AT bx3 rm3)
    ; Paths:
    ; 1-|-2-|-3

    (PATH rm1 rm2) (PATH rm2 rm3 )
    (CLOSED-DOOR rm3 rm2) (CLOSED-DOOR rm2 rm1 )
    (KEY ke1 rm2 rm1) (KEY ke2 rm3 rm2)
    (KEY-AT-ROOM ke1 rm1) (KEY-AT-ROOM ke2 rm2)
  )
  ;collect box 3 and put in rm3
  (:goal
    (and (ROBOT-AT rm3)
         (BOX-AT bx1 rm3)
    )
  )
)

```

Solución:

```
( 0 ) take-key.rm1.ke1@0
( 1 ) open-o.rm2.rm1.ke1@1
( 1 ) load.rm1.bx1@1
( 2 ) move-od.rm1.rm2@2
( 3 ) take-key.rm2.ke2@3
( 4 ) open-o.rm3.rm2.ke2@4
( 5 ) move-od.rm2.rm3@5
( 6 ) unload.rm3.bx1@6
```

7 steps, 7 actions

9 predicates, 7 actions.

Grounded predicates: 60

Grounded actions: 32

5.1.5. Escenario C

Este escenario es algo más complicado que el anterior. En él puede haber más de un robot, cada uno puede transportar una caja. Existen igualmente llaves y puertas pero en este caso se utilizó el criterio de usar un predicado para indicar puertas abiertas (en vez de puertas cerradas). Cada robot puede moverse independientemente a habitaciones y coger cajas y/o llaves.

Este escenario se diseñó para permitir paralelismo de acciones, pero hay algunas restricciones, por ejemplo dos robots no pueden coger la misma llave o la misma caja, puesto que eso daría lugar a inconsistencias.

En este escenario un robot no puede soltar una llave que ha cogido previamente (no hay acción para ello), por tanto será el robot que posea la llave el responsable de abrir la puerta correspondiente si lo necesita, o si lo necesitan otros robots.

En principio se permite el paso de varios robots que entran o salen de una misma habitación, suponiendo que hay alguna forma de resolver esos posibles conflictos a bajo nivel.

En este escenario no sólo tenemos más elementos y por tanto una situación más compleja, además varias de las acciones resultan modificadas respecto al escenario anterior al incluir un nuevo parámetro (el robot, de entre varios posibles robots activos). Esto provocará una mayor complejidad en las acciones y se generarán muchas más posibles acciones al realizar el grounding o al tratar de resolver el problema.

```

; (c) Cristobal Tapia 2016
; domain multi-robot, open/close doors, carry boxes
(define (domain robot_domain_C)
  ;domain robot without or-op in preconditions
  (:requirements :typing
    :negative-preconditions )
  (:types room box door key robot) ; no case sensitive
  (:predicates
    (ROBOT-AT ?t - robot ?r - room) ; robot t is in room
    (ROBOT-LOADED ?t - robot) ; robot t is loaded
    (BOX-AT ?b - box ?r - room)
    ; a path between a room and another room, is bidirec.
    (DOOR-BETWEEN ?d - door ?x ?y - room)
    ; robot t is carrying this box
    (ROBOT-CARRIES ?t - robot ?b - box)
    (OPEN-DOOR ?d - door)
    (KEY ?k - key ?d - door )
    (ROBOT-HAS-KEY ?t - robot ?k -key)
    (KEY-AT-ROOM ?k - key ?r - room)
  )
  (:action move-OD :parameters (?t - robot ?orig - room
    ?dest - room ?d - door)
    :precondition ( and ( ROBOT-AT ?t ?orig )
      ( DOOR-BETWEEN ?d ?orig ?dest )
      ( OPEN-DOOR ?d )
    )
    :effect (and ( ROBOT-AT ?t ?dest )
      (not ( ROBOT-AT ?t ?orig ))
    )
  )
  (:action move-DO :parameters (?t - robot ?orig - room
    ?dest - room ?d - door)
    :precondition ( and ( ROBOT-AT ?t ?orig )
      ( DOOR-BETWEEN ?d ?dest ?orig )
      ( OPEN-DOOR ?d )
    )
    :effect (and ( ROBOT-AT ?t ?dest )
      (not ( ROBOT-AT ?t ?orig ))
    )
  )
  (:action open-1 :parameters ( ?t - robot ?d - door
    ?k - key ?r1 ?r2 -room)
    :precondition ( and ( DOOR-BETWEEN ?d ?r1 ?r2 )
      ( ROBOT-AT ?t ?r1 )
      ( not (OPEN-DOOR ?d))
      ( KEY ?k ?d)
      ( ROBOT-HAS-KEY ?t ?k)
    )
    :effect (and ( OPEN-DOOR ?d)
  )
  (:action open-2 :parameters ( ?t - robot ?d - door
    ?k - key ?r1 ?r2 -room)

```

```

:precondition ( and ( DOOR-BETWEEN ?d ?r1 ?r2 )
                    ( ROBOT-AT ?t ?r2 )
                    ( not (OPEN-DOOR ?d))
                    ( KEY ?k ?d)
                    ( ROBOT-HAS-KEY ?t ?k)
                  )
:effect (and ( OPEN-DOOR ?d)
            )
)
(:action close-1 :parameters ( ?t - robot ?d - door
                               ?r1 ?r2 -room)
  :precondition ( and ( DOOR-BETWEEN ?d ?r1 ?r2 )
                      ( ROBOT-AT ?t ?r1 )
                      ( OPEN-DOOR ?d )
                    )
  :effect (and ( not (OPEN-DOOR ?d) )
              )
)
(:action close-2 :parameters ( ?t - robot ?d - door
                               ?r1 ?r2 -room)
  :precondition ( and ( DOOR-BETWEEN ?d ?r1 ?r2 )
                      ( ROBOT-AT ?t ?r2 )
                      ( OPEN-DOOR ?d )
                    )
  :effect (and ( not (OPEN-DOOR ?d) )
              )
)
(:action take-key :parameters ( ?t - robot ?r - room
                                ?k - key )
  :precondition (and ( ROBOT-AT ?t ?r)
                    ( KEY-AT-ROOM ?k ?r)
                  )
  :effect ( and ( ROBOT-HAS-KEY ?t ?k )
              ( not (KEY-AT-ROOM ?k ?r ) )
            )
)
(:action load :parameters (?t - robot ?r - room ?b - box)
  :precondition (and (BOX-AT ?b ?r )
                    (ROBOT-AT ?t ?r )
                    ( not (ROBOT-LOADED ?t ) )
                  )
  :effect (and (ROBOT-LOADED ?t)
              (not (BOX-AT ?b ?r ))
              (ROBOT-CARRIES ?t ?b )
            )
)
(:action unload :parameters ( ?t - robot ?r - room ?b - box)
  :precondition (and (ROBOT-AT ?t ?r )
                    (ROBOT-LOADED ?t )
                    (ROBOT-CARRIES ?t ?b )
                  )
)

```

5. ESCENARIOS PARA LA SIMULACIÓN Y PRUEBAS

```
      :effect (and (not (ROBOT-LOADED ?t))
                  (BOX-AT ?b ?r )
                  (not (ROBOT-CARRIES ?t ?b ))
                )
    )
  )
```

El siguiente listado muestra un problema sencillo en este dominio. Se trata de un escenario con 2 robots, con 3 habitaciones y puertas cerradas entre ellas. El objetivo sólo pide transportar una caja, pero para ello es necesario coger las llaves y abrir las puertas.

```
; (c) Cristobal Tapia 2016
(define (problem robot_problem_C_Ps01)
  (:domain robot_domain_C)
  (:objects
    bot1 bot2 - robot
    rm1 rm2 rm3 - room
    bx1 bx2 bx3 - box
    do1 do2 - door
    ke1 ke2 - key
  )
  (:init
    (ROBOT-AT bot1 rm1) (ROBOT-AT bot2 rm1)
    (BOX-AT bx1 rm1) (BOX-AT bx2 rm2) (BOX-AT bx3 rm3)
    (DOOR-BETWEEN do1 rm1 rm2) (DOOR-BETWEEN do2 rm2 rm3 )
    ; Paths:
    ; 1-|-2-|-3
    (KEY ke1 do1) (KEY ke2 do2)
    (KEY-AT-ROOM ke1 rm1) (KEY-AT-ROOM ke2 rm2)
  )
  ;collect one box and take to rm3. Close all doors.
  (:goal
    (and (ROBOT-AT bot1 rm3)
          (ROBOT-AT bot2 rm3)
          (BOX-AT bx1 rm3)
          (not (ROBOT-LOADED bot1) )
          (not (ROBOT-LOADED bot2) )
          (not (OPEN-DOOR do1) )
          (not (OPEN-DOOR do2) )
        )
  )
)
```

Si se permite paralelismo, una solución de este ejemplo es:

```
( 0 ) take-key.bot1.rm1.ke1@0
( 1 ) open-1.bot1.do1.ke1.rm1.rm2@1
```

```
( 1 ) load.bot1.rm1.bx1@1
( 2 ) move-od.bot1.rm1.rm2.do1@2
( 2 ) move-od.bot2.rm1.rm2.do1@2
( 3 ) close-2.bot1.do1.rm1.rm2@3
( 3 ) take-key.bot2.rm2.ke2@3
( 4 ) open-1.bot2.do2.ke2.rm2.rm3@4
( 5 ) move-od.bot1.rm2.rm3.do2@5
( 5 ) move-od.bot2.rm2.rm3.do2@5
( 6 ) close-2.bot1.do2.rm2.rm3@6
( 6 ) unload.bot1.rm3.bx1@6
```

7 steps, 12 actions

9 predicates, 9 actions.

Grounded predicates: 57

Grounded actions: 72

Este otro ejemplo sitúa a 4 robots, que deben llevar varias cajas. En este caso existe una puerta que no puede abrirse por no disponer de llave.

; (c) Cristobal Tapia 2016

```
(define (problem robot_problem_C_Pm01)
  (:domain robot_domain_C)
  (:objects
    bot1 bot2 bot3 bot4 - robot
    rm1 rm2 rm3 rm4 rm5 rm6 - room
    bx1 bx2 bx3 bx4 bx5 bx6 - box
    do1 do2 do3 do4 do5 do6 - door
    ke1 ke2 ke3 ke4 ke5 - key
  )
  (:init
    (ROBOT-AT bot1 rm1) (ROBOT-AT bot2 rm1)
    (ROBOT-AT bot3 rm1) (ROBOT-AT bot4 rm1)
    (BOX-AT bx1 rm1) (BOX-AT bx2 rm2) (BOX-AT bx3 rm3)
    (BOX-AT bx4 rm1) (BOX-AT bx5 rm2) (BOX-AT bx6 rm3)
    (DOOR-BETWEEN do1 rm1 rm2) (DOOR-BETWEEN do2 rm2 rm3 )
    (DOOR-BETWEEN do3 rm3 rm6) (DOOR-BETWEEN do4 rm5 rm6 )
    (DOOR-BETWEEN do5 rm4 rm5) (DOOR-BETWEEN do6 rm1 rm4 )
    ; Paths:
    ; 1-|-2-|-3
    ; + + ; door between 1-4, but no key.
    ; 4-|-5-|-6
    (KEY ke1 do1) (KEY ke2 do2) (KEY ke3 do3)
    (KEY ke4 do4) (KEY ke5 do5)
    (KEY-AT-ROOM ke1 rm1) (KEY-AT-ROOM ke2 rm2)
    (KEY-AT-ROOM ke3 rm3) (KEY-AT-ROOM ke4 rm3)
    (KEY-AT-ROOM ke5 rm3)
```

```
)  
;collect one box ant take to rm3. Close all doors.  
(:goal  
  (and (ROBOT-AT bot1 rm3)  
        (ROBOT-AT bot2 rm3)  
        (BOX-AT bx1 rm3)  
        (not (ROBOT-LOADED bot1) )  
        (not (ROBOT-LOADED bot2) )  
        (not (OPEN-DOOR do1) )  
        (not (OPEN-DOOR do2) )  
      )  
)  
)
```

Una solución para este problema es:

```
( 0 ) take-key.bot1.rm1.ke1@0  
( 1 ) open-1.bot1.do1.ke1.rm1.rm2@1  
( 2 ) move-od.bot2.rm1.rm2.do1@2  
( 2 ) move-od.bot4.rm1.rm2.do1@2  
( 3 ) take-key.bot2.rm2.ke2@3  
( 3 ) load.bot1.rm1.bx1@3  
( 4 ) move-od.bot1.rm1.rm2.do1@4  
( 4 ) open-1.bot2.do2.ke2.rm2.rm3@4  
( 5 ) move-od.bot1.rm2.rm3.do2@5  
( 5 ) move-od.bot2.rm2.rm3.do2@5  
( 5 ) move-od.bot4.rm2.rm3.do2@5  
( 5 ) close-1.bot3.do1.rm1.rm2@5  
( 6 ) close-2.bot4.do2.rm2.rm3@6  
( 6 ) take-key.bot1.rm3.ke3@6  
( 6 ) take-key.bot1.rm3.ke4@6  
( 6 ) take-key.bot1.rm3.ke5@6  
( 6 ) unload.bot1.rm3.bx1@6
```

7 steps, 17 actions

9 predicates, 9 actions.

Grounded predicates: 390

Grounded actions: 544

5.1.5. Escenario con modificadores emocionales

Como parte del trabajo de planificación con robots, se planteó una idea de planificación teniendo en cuenta aspectos emocionales en los robots, una línea de investigación muy interesante actualmente [2] y que se está empleando en nuestro grupo de investigación [1].

Este ejemplo es parecido a los anteriores, en los que robots llevan cajas entre habitaciones. En este caso se añaden unas características emocionales a los robots, que pueden estar felices (HAPPY), asustados (SCARED), o enfadados (ANGRY). Un robot pasa a estar asustado si entra en una habitación oscura (las habitaciones pueden estar oscuras o no). Un robot pasa a estar enfadado si coge una caja pesada (hay un predicado para indicar si una caja es pesada o ligera). La forma de pasar a estar felices y eliminar los estados de miedo o enfado es ir a una habitación que tenga música y escuchar la música (es una acción especial). Aunque es un ejemplo muy sencillo, permite mostrar algunas posibilidades en el ámbito de planificación incluyendo emociones.

```
; (c) Cristobal Tapia 2016
; domain multi-robot, carry boxes,
; with emotional predicates
; room can be dark, -> robot enters and gets scared
;

(define (domain robot_domain_E)
  ;domain robot without or-op in preconditions
  (:requirements :typing
                 :negative-preconditions )
  (:types room box door light robot) ; no case sensitive
  (:predicates
    (ROBOT-AT ?t - robot ?r - room) ; robot t is in room
    (ROBOT-LOADED ?t - robot) ; robot t is loaded
    (BOX-AT ?b - box ?r - room) ; a box on a room
    (HEAVY-BOX ?b - box) ; this box is heavy
    (PATH-BETWEEN ?x ?y - room) ; no bidirectional
    (ROBOT-CARRIES ?t - robot ?b - box)
    (ANGRY ?t - robot) ; Emotional states
    (HAPPY ?t - robot)
    (SCARED ?t - robot)
    (DARK-ROOM ?r - room) ; the room is dark - scary
    (MUSIC-ROOM ?r -room) ; music sounds this room
  )
  (:action move-to-light :parameters (?t - robot ?orig - room
                                     ?dest - room)
    :precondition ( and ( ROBOT-AT ?t ?orig )
                        ( not (DARK-ROOM ?dest))
                        ( PATH ?orig ?dest )
                      )
    :effect (and ( ROBOT-AT ?t ?dest )
```

```
(not ( ROBOT-AT ?t ?orig ))
)
)
(:action move-to-dark :parameters (?t - robot ?orig - room
                                   ?dest - room)
  :precondition ( and ( ROBOT-AT ?t ?orig )
                      ( PATH ?orig ?dest)
                      ( DARK-ROOM ?dest)
                      ( not (SCARED ?t ))
                    )
  :effect (and ( ROBOT-AT ?t ?dest )
              (not ( ROBOT-AT ?t ?orig ))
              (SCARED ?t)
              ( not (HAPPY ?t))
            )
)
)
(:action listen-music :parameters ( ?t - robot ?r - room)
  :precondition ( and ( MUSIC-ROOM ?r )
                      ( ROBOT-AT ?t ?r)
                    )
  :effect (and ( not (ANGRY ?t) )
              ( not (SCARED ?t) )
              ( HAPPY ?t )
            )
)
)
(:action load :parameters (?t - robot ?r - room ?b - box)
  :precondition (and (BOX-AT ?b ?r )
                    (ROBOT-AT ?t ?r )
                    ( not (ROBOT-LOADED ?t ) )
                    ( HAPPY ?t )
                    ( not (ANGRY ?t))
                    ( not (SCARED ?t))
                    ( not (HEAVY-BOX ?b))
                  )
  :effect (and (ROBOT-LOADED ?t)
              (not (BOX-AT ?b ?r ))
              (ROBOT-CARRIES ?t ?b )
            )
)
)
(:action load-heavy :parameters (?t - robot ?r - room
                                 ?b - box)
  :precondition (and (BOX-AT ?b ?r )
                    (ROBOT-AT ?t ?r )
                    ( not (ROBOT-LOADED ?t ) )
                    ( HAPPY ?t )
                    ( not (ANGRY ?t))
                    ( not (SCARED ?t))
                    (HEAVY-BOX ?b)
                  )
  :effect (and (ROBOT-LOADED ?t)
              (not (BOX-AT ?b ?r ))
              (ROBOT-CARRIES ?t ?b )
              (not (HAPPY ?t))
            )
)
```

```

                (ANGRY ?t)
            )
        )
        (:action unload :parameters ( ?t - robot ?r - room ?b - box)
          :precondition (and (ROBOT-AT ?t ?r )
                             (ROBOT-LOADED ?t )
                             (ROBOT-CARRIES ?t ?b )
                           )
          :effect (and (not (ROBOT-LOADED ?t))
                      (BOX-AT ?b ?r )
                      (not (ROBOT-CARRIES ?t ?b ))
                    )
        )
    )
)

```

Problema sencillo sobre ese dominio:

```

; (c) Cristobal Tapia 2016

(define (problem robot_problem_E_Ps01)
  (:domain robot_domain_E)
  (:objects
    bot1 - robot
    rm1 rm2 rm3 - room
    bx1 bx2 bx3 - box
  )
  (:init
    (ROBOT-AT bot1 rm1) (HAPPY bot1)
    (BOX-AT bx1 rm1) (BOX-AT bx2 rm2) (BOX-AT bx3 rm3)
    (PATH rm1 rm2) (PATH rm2 rm1 )
    (PATH rm2 rm3) (PATH rm3 rm2 )
    ; Paths:
    ; 1-|-2-|-3
    (HEAVY-BOX bx1)
    (HEAVY-BOX bx2)
    (DARK-ROOM rm3)
    (MUSIC-ROOM rm1)
  )
  ;move boxes
  ;lift heavy box bx1 , bx2, robot will need music
  ;bx2->rm3 forces robot to enter a dark room
  (:goal
    (and (ROBOT-AT bot1 rm2)
          (BOX-AT bx1 rm2)
          (BOX-AT bx2 rm3)
        )
  )
)

```

Una solución para este problema es:

```
( 0 ) load-heavy.bot1.rm1.bx1@0
( 1 ) listen-music.bot1.rm1@1
( 2 ) move-to-light.bot1.rm1.rm2@2
( 3 ) unload.bot1.rm2.bx1@3
( 4 ) load-heavy.bot1.rm2.bx2@4
( 5 ) move-to-dark.bot1.rm2.rm3@5
( 6 ) unload.bot1.rm3.bx2@6
( 7 ) move-to-light.bot1.rm3.rm2@7
```

8 steps, 8 actions

11 predicates, 6 actions.

Grounded predicates: 37

Grounded actions: 23

6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1. Conclusiones

Se ha propuesto una arquitectura modular (o framework) para un agente inteligente centrada en planificación, que incluye elementos de alto nivel, planificador, simulador y visualizador en 3D de escenarios de planificación. Además se ha diseñado y realizado un editor que facilita la programación en PDDL, mediante ayudas como autocompletar, detección de errores típicos etc.

Dentro de ese sistema se han desarrollado gran parte de los módulos, como el analizador del lenguaje PDDL, el traductor a problemas SAT, el SAT solver, el simulador y el interfaz de usuario.

Se ha construido un planificador basado en SAT, desarrollando la conversión de un problema de planificación a SAT.

Esta conversión emplea las técnicas existentes en otros conversores pero además se aporta un nuevo tipo de codificación de acciones basado en la detección de incompatibilidad entre ellas. El nuevo tipo de codificación, de tipo híbrido, es más compacto permitiendo la concurrencia entre acciones.

El entorno proporciona una solución completa para escribir problemas en PDDL, resolverlos y comprobar los resultados bien mediante simulación y visualización en 3D bien mediante análisis de resultados.

6.2. Trabajos futuros

En este trabajo quedan muchas líneas abiertas que pueden ser objeto de mejoras y de investigaciones posteriores.

Una línea interesante es el estudio de distintas variantes de codificación híbrida, a partir de la propuesta en este trabajo. El empleo de variaciones junto con otras mejorar de los planificadores existentes puede ofrecer aún mejores resultados en el rendimiento.

Otra línea de trabajo es la mejora y ampliación del entorno online de simulación, en el que podrían mejorarse tanto las opciones del editor PDDL Builder como el *backend*, es decir, el uso de servicios *en la nube*, para mejorar la comunicación y colaboración de la comunidad investigadora en este campo. También puede ofrecerse una mayor gama de planificadores y servicios relativos a su uso. Otro aspecto es el interfaz 3D de visualización, que podría mejorarse para facilitar la generación de escenarios 3D para usuarios no expertos y flexibilizar su uso.

Finalmente, dentro del apartado del planificador SAT, cabe la posibilidad de mejorar el SAT *solver* del que disponemos, incorporando técnicas recientes para mejorar el rendimiento, tales como aprendizaje de cláusulas, literales observados, mantener información entre instancias adaptado a los problemas de planificación, etc.

7. REFERENCIAS

1. Álvarez M, Galán R, Matía F, Rodríguez-Losada D, Jiménez A (2010) An emotional model for a guide robot. *IEEE Trans Syst Man, Cybern Part A Systems Humans* 40:982–992. doi: 10.1109/TSMCA.2010.2046734
2. Asensio J, Jiménez M, Fernández S, Borrajo D (2008) A Social and Emotional Model for Obtaining Believable Emergent Behaviors. In: Dochev D, Pistore M, Traverso P (eds) *Artif. Intell. Methodol. Syst. Appl.* 13th Int. Conf. AIMS A 2008, Varna, Bulg. Sept. 4-6, 2008. Proc. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 395–399
3. Audemard G, Simon L (2009) Predicting Learnt Clauses Quality in Modern SAT Solvers. In: *IJCAI*. pp 399–404
4. Bäckström C, Nebel B (1995) Complexity results for SAS+ planning. *Comput Intell* 11:625–655.
5. Blum AL, Furst ML (1997) Fast planning through planning graph analysis. *Artif Intell* 90:281–300.
6. Blum AL, Langford JC (2000) Probabilistic Planning in the Graphplan Framework. In: Biundo S, Fox M (eds) *Recent Adv. AI Plan.* 5th Eur. Conf. Planning, ECP’99, Durham, UK, Sept. 8-10, 1999. Proc. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 319–332
7. Bonet B, Geffner H (2001) Planning as heuristic search. *Artif Intell*

7. REFERENCIAS

- 129:5–33.
8. Bylander T (1994) The computational complexity of propositional STRIPS planning. *Artif Intell* 69:165–204. doi: 10.1016/0004-3702(94)90081-7
 9. Cenamor I, De La Rosa T, Fernández F (2016) The IBaCoP planning system: instance-based configured portfolios. *J Artif Intell Res* 56:657–691.
 10. Cook SA (1971) The complexity of theorem-proving procedures. In: *Proc. third Annu. ACM Symp. Theory Comput.* ACM, pp 151–158
 11. Davis M, Logemann G, Loveland D (1962) A Machine Program for Theorem-proving. *Commun ACM* 5:394–397. doi: 10.1145/368273.368557
 12. Duch W, Oentaryo R, Pasquier M (2008) Cognitive Architectures: Where do we go from here? *Proc 2008 Conf Artif Gen Intell 2008 Proc First AGI Conf* 171:122–136.
 13. Edelkamp S, Hoffmann J (2004) PDDL2. 2: The language for the classical part of the 4th international planning competition. 4th Int. Plan. Compet. (IPC’04), ICAPS’04
 14. Eén N, Sörensson N (2004) An extensible SAT-solver. *Theory Appl. Satisf. Test.*
 15. Ernst MDM, Millstein TTD, Weld DDS (1997) Automatic SAT-compilation of planning problems. In: *IJCAI*. pp 1169–1176
 16. Fikes RE, Nilsson NJ (1971) Strips: A new approach to the application of theorem proving to problem solving. *Artif Intell* 2:189–208. doi: 10.1016/0004-3702(71)90010-5
 17. Fox M, Long D (2003) PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *J Artif Intell Res(JAIR)* 20:61–124.
 18. Garvey A, Raja A, Horling B, Decker K, Vincent R, Zhang S, Wagner T, Lesser V (1999) The Taems White Paper. University of Massachusetts, Multi-Agent Systems Lab
 19. Gerevini A, Long D (2005) Plan constraints and preferences in PDDL3. *Lang. Fifth Int. Plan. Compet. Tech. Rep. Tech. Report*, Dep. Electron. Autom. Univ. Brescia, Italy 75:
 20. Gerevini A, Long D (2006) Preferences and soft constraints in PDDL3. In: *ICAPS Work. Plan. with Prefer. soft constraints*. pp 46–53
 21. Ghallab M, Nau D, Traverso P (2004) *Automated Planning: Theory and Practice*. Elsevier
-

7. REFERENCIAS

22. Hao J-K, Dorne R (1994) A new population-based method for satisfiability problems. In: ECAI. PITMAN, pp 135–139
23. Helmert M Fast Downward homepage. <http://www.fast-downward.org/>.
24. Helmert M (2006) The fast downward planning system. *J Artif Intell Res* 26:191–246. doi: 10.1613/jair.1705
25. Helmert M (2008) Changes in PDDL 3.1. Unpubl. Summ. from IPC-2008 website
26. Helmert M (2009) Concise finite-domain representations for PDDL planning tasks. *Artif Intell* 173:503–535. doi: 10.1016/j.artint.2008.10.013
27. Helmert M (2009) Concise finite-domain representations for PDDL planning tasks. *Artif Intell* 173:503–535.
28. Hoffmann J, Nebel B (2001) The FF planning system: Fast plan generation through heuristic search. *J Artif Intell Res* 14:263–312. doi: 10.1613/jair.855
29. Huang R (2011) Efficient Automated Planning with New Formulations. WASHINGTON UNIVERSITY IN ST. LOUIS School of Engineering and Applied Science
30. Huang R, Chen Y, Zhang W (2010) A Novel Transition Based Encoding Scheme for Planning as Satisfiability. *Artif Intell* 89–94.
31. Huang R, Chen Y, Zhang W (2012) SAS+ Planning as Satisfiability. *J Artif Intell Res* 43:293–328. doi: 10.1613/jair.3442
32. Kautz H, Selman B (1992) Planning as Satisfiability (SAT - Plan). *Proc. 10th Eur. Conf. Artif. Intel.* 9:
33. Kautz H, Selman B (1996) Pushing the envelope: Planning, propositional logic, and stochastic search. In: *Proc. Natl. Conf. Artif. Intell.* pp 1194–1201
34. Kautz H, Selman B (1998) BLACKBOX: A new approach to the application of theorem proving to problem solving. In: *AIPS98 Work. Plan. as Comb. Search.* pp 58–60
35. Kautz H, Selman B (1999) Unifying SAT-based and graph-based planning. In: *IJCAI.* pp 318–325
36. Kovacs DL (2011) BNF definition of PDDL 3.1. Unpubl. Manusc. from IPC-2011 website
37. Langley P, Laird JE, Rogers S (2006) Cognitive architectures: Research issues and challenges.
38. McDermott D, Ghallab M, Howe A, Knoblock C, Ram A, Veloso M,

7. REFERENCIAS

- Weld D, Wilkins D (1998) PDDL-the planning domain definition language.
39. McDermott D V (1996) A Heuristic Estimator for Means-Ends Analysis in Planning. In: AIPS. pp 142–149
 40. Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: Engineering an Efficient SAT Solver. In: Proc. 38th Annu. Des. Autom. Conf. ACM, New York, NY, USA, pp 530–535
 41. Muise C (2016) Planning.Domains. 26th Int. Conf. Autom. Plan. Sched. - Demonstr.
 42. Newell A, Shaw J, Simon H (1959) Report on a general problem solving program. IFIP Congr.
 43. Newell A, Simon H (1961) GPS, a program that simulates human thought.
 44. Pednault EPD (1987) Formulating multi-agent, dynamic-world problems in the classical planning framework. In: Reason. About Actions Plans. pp 47–82
 45. Pednault EPD (1987) Formulating multiagent, dynamic-world problems in the classical planning framework. Reason about actions plans 47–82.
 46. Pfahringer B (2011) Conjunctive normal form. In: Encycl. Mach. Learn. Springer, pp 209–210
 47. Plch T, Chomut M, Brom C, Barták R (2012) Inspect, edit and debug PDDL documents: Simply and efficiently with PDDL studio. Syst Demonstr Exhib ICAPS 15–18.
 48. Quintero E, García Á, Borrajo D, Fernández F (2010) Control architecture for autonomous mobile robots with automated planning techniques.
 49. Rintanen J Madagascar - SAT planning. <https://users.ics.aalto.fi/rintanen/satplan.html>.
 50. Rintanen J (2009) Planning and SAT. Handb Satisf 185:483–504.
 51. Rintanen J (2012) Engineering efficient planners with SAT. In: Proc. 20th Eur. Conf. Artif. Intell. IOS Press, pp 684–689
 52. Rintanen J (2012) Engineering efficient planners with SAT. Front Artif Intell Appl 242:684–689. doi: 10.3233/978-1-61499-098-7-684
 53. Rintanen J (2014) Madagascar: Scalable planning with sat. Proc. 8th Int. Plan. Compet.
 54. Robinson N, Gretton C, Pham D-N, Sattar A (2008) A compact and efficient SAT encoding for planning. ICAPS 2008 - Proc. 18th Int. Conf.
-

7. REFERENCIAS

- Autom. Plan. Sched.
55. Rodriguez-Losada D, Matia F, Galan R, Hernando M, Montero JM, Lucas JM (2008) Urbano, an interactive mobile tour-guide robot. *Adv. Serv. Robot.*
 56. Russell S, Norvig P (2010) *Artificial Intelligence: A Modern Approach*, 3rd Editio. Pearson, New Jersey
 57. San Segundo P, Tapia C (2014) Relaxed approximate coloring in exact maximum clique search. *Comput Oper Res* 44:185–192.
 58. San Segundo P, Tapia C, Puente J, Rodríguez-Losada D (2008) A new exact bit-parallel algorithm for SAT. In: *Proc. - Int. Conf. Tools with Artif. Intell. ICTAI*. pp 59–65
 59. Selman B, Kautz HA, Cohen B (1993) Local search strategies for satisfiability testing. *Cliques, Color Satisf* 26:521–532.
 60. Sideris A, Dimopoulos Y (2010) Constraint Propagation in Propositional Planning. *Int Conf Autom Plan Sched* 153–160.
 61. Strobel V, Kirsch A (2014) Planning in the Wild: Modeling Tools for PDDL BT - KI 2014: Advances in Artificial Intelligence: 37th Annual German Conference on AI, Stuttgart, Germany, September 22–26, 2014. *Proceedings*. In: Lutz C, Thielscher M (eds). Springer International Publishing, Cham, pp 273–284
 62. Tapia C, San Segundo P, Artieda J (2015) a Pddl-Based Simulation System. In: *Int. Conf. Intell. Syst. Agents 2015*. pp 81–88
 63. Trudeau RJ (2013) *Introduction to graph theory*. Courier Corporation
 64. Usero Vélchez JL TFG - PLAGUE. <http://hdl.handle.net/10630/11652>.
 65. Vallati M (2012) A Guide to Portfolio-Based Planning BT - Multi-disciplinary Trends in Artificial Intelligence: 6th International Workshop, MIWAI 2012, Ho Chi Minh City, Vietnam, December 26–28, 2012. *Proceedings*. In: Sombattheera C, Loi NK, Wankar R, Quan T (eds). Springer Berlin Heidelberg, Berlin, Heidelberg, pp 57–68
 66. Vize Y, Weissenbacher G, Malik S (2015) Boolean satisfiability solvers and their applications in model checking. *Proc IEEE* 103:2021–2035.
 67. Wehrle M, Rintanen J (2007) Planning as Satisfiability with Relaxed $\exists$ -StepPlans. *AI 2007 Adv Artif Intell* 244–253.
 68. Weld D (1994) An introduction to least commitment planning. *AI Mag.*
 69. West DB (2001) *Introduction to graph theory*. Prentice hall Upper Saddle River
 70. Wooldridge M (2009) *An Introduction to MultiAgent Systems*
-

7. REFERENCIAS

- [Paperback]. Chichester, Engl. doi: 10.1108/eb050656
71. Diccionario de la Real Academia de la Lengua Española. <http://dle.rae.es/?id=LqtyoaQ>.
 72. WalkSAT homepage.
 73. CUDA web page. <http://www.nvidia.es/object/cuda-parallel-computing-es.html>.
 74. BitScan Library Source Code. <https://github.com/psanse/bitscan>.
 75. JSON web site. <http://www.json.org/>.