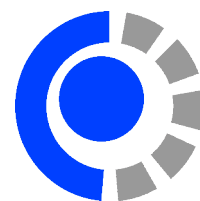




UNIVERSIDAD NACIONAL DEL COMAHUE  
FACULTAD DE INFORMÁTICA



TESIS DE LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

## Planificación Continua mediante PDDL

Germán Alejandro Braun

Director: Mg. Gerardo Parra

NEUQUÉN

ARGENTINA

2012

## PREFACIO

Esta tesis es presentada como parte de los requisitos finales para optar al grado académico de *Licenciado/a en Ciencias de la Computación*, otorgado por la Universidad Nacional del Comahue, y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otras. La misma es el resultado de la investigación llevada a cabo en el Departamento de Teoría de la Computación, de la Facultad de Informática, en el período comprendido entre Septiembre del 2011 y Agosto del 2012, bajo la dirección del Mg. Gerardo Parra.

Germán Braun  
FACULTAD DE INFORMÁTICA  
UNIVERSIDAD NACIONAL DEL COMAHUE  
*Neuquén, 6 de Septiembre de 2012.*



UNIVERSIDAD NACIONAL DEL COMAHUE

Facultad de Informática

La presente tesis ha sido aprobada el día ....., mereciendo la calificación de .....

## DEDICATORIAS

*A mi mamá y a mi papá, a quienes debo mi educación.*

## AGRADECIMIENTOS

*A Caro, por su paciencia y ayuda mientras escribía.*

*A Fede, mi hermano, que tiene mucho que ver con esto también.*

*A mis profesores Gerardo P., Laura C., Claudio V. que me aconsejaron, corrigieron y ayudaron desinteresadamente para este trabajo y para mi carrera en general. A Mario M., por “prestarme” su framework.*

*A mis amigos de la vida, a los de la facu y resto de mi familia.*

## RESUMEN

La temática que se investiga en este trabajo es la planificación continua mediante especificaciones en un lenguaje de definición de dominios de planificación denominado PDDL. El objetivo es implementar un traductor de un subconjunto de este lenguaje a fin de que el Framework de Planificación Continua, también presentado aquí, pueda aprovechar las características de PDDL. Este enfoque es relevante ya que plantea la posibilidad de combinar la expresividad de un lenguaje estándar, como PDDL, con un planificador continuo capaz de resolver problemas en ambientes reales. Es esperable alcanzar un mayor nivel de abstracción para tratar dominios más complejos y, además, realizar comparaciones empíricas de performance con otras soluciones para un mismo problema.

El lenguaje del Planificador Continuo, y uno de los lenguajes fundacionales de representación de problemas de planificación, es STRIPS. Este formalismo permite modelar acciones simples como listas de precondiciones y efectos. Además, cada lista es una conjunción de proposiciones. Por su parte, PDDL es un lenguaje sensiblemente más expresivo que STRIPS. Este lenguaje provee características adicionales que definen varios niveles de expresividad permitiendo enriquecer la definición de dominios y problemas de planificación.

En base a lo expuesto, se plantea la necesidad de tratar con lenguajes más complejos con el objetivo de modelar acciones aplicables en ambientes reales. No obstante, esto resulta en un mayor costo computacional de los algoritmos al momento de resolver problemas de planificación y, por lo tanto, es importante encontrar un balance aceptable entre expresividad y complejidad.

Este trabajo presenta un traductor, desarrollado en Ciao Prolog, de un subconjunto PDDL que soporta algunas de sus características. El subconjunto conforma el lenguaje fuente del traductor y su salida consiste en una representación similar a STRIPS que permite adaptar este traductor a diferentes planificadores, entre ellos, al Planificador Continuo.

La investigación también presenta otros dos resultados teóricos. La redefinición de la arquitectura modular del Framework de Planificación Continua para soportar PDDL y una nueva variante del lenguaje STRIPS para modelar acciones con cuantificación universal en sus precondiciones. Sobre este último resultado, también introducimos un esquema de compilación que permite traducir las acciones, definidas en esta variante, a STRIPS estándar. Además, esta traducción conserva los resultados de planificación que se obtienen con ambas especificaciones. Este esquema, junto con otros esquemas de compilación enunciados, son usados para el desarrollo del traductor propuesto.

La experimentación sobre la implementación presentada se realiza sobre distintas instancias del dominio del “Mundo de Bloques” (un problema clásico de la literatura de Inteligencia Artificial). Cada instancia de este problema es modelada en PDDL empleando las distintas características del subconjunto considerado.

## ABSTRACT

The researched topic in this work is related to the continuous planning using specifications written in the Planning Domain Definition Language named PDDL. The objective is to implement a translator of a PDDL language subset in order to allow that the Continuous Planning Framework (also, it is introduced here) supports PDDL features. This approach is relevant because it supposes combining the expressiveness of PDDL standard language with a continuous planner that tries to solve problems in real environments. We hope to reach a high-level abstraction in order to define more complex domains and to compare the performance with other solutions for the same problem.

The continuous planner language is STRIPS. This formalism allows modeling simple actions using precondition and effect lists. Also, each list is a conjunction of propositions. Particularly, PDDL is more expressive than STRIPS because it provides other features defining expressiveness levels and allowing to enrich the definition of domains and problems of planning.

Based on the above, it is necessary to try with more expressive languages in order to model actions in real environments. Nevertheless, this implies to work with more complex algorithms too. Then, a trade-off between complexity and expressiveness is important.

This thesis presents a translator implemented in Ciao Prolog of a PDDL subset with additional features. This subset defines the source language of translator and its output is a STRIPS-like notation. This output allows adapting the translator to different planning algorithms as the Continuous Planner.

The research also presents other two theoretical results. First, the architecture of Planning Continuous Framework is redefined. After, a new STRIPS variant with universal preconditions is presented. In the last result, actions defined in this variant could be translated to STRIPS preserving results for both specifications. This result together with other compilation schemes are used in the translator implementation.

The experimentation on the implementation has been done on different instances of the “Blocks World” problem (a classic problem of Artificial Intelligence bibliography). Each instance of this problem is modeled in PDDL by using different features.

# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Trabajos Relacionados . . . . .	2
1.2. Estructura de la Tesis . . . . .	2
<b>2. El Lenguaje PDDL</b>	<b>3</b>
2.1. Planificación . . . . .	3
2.1.1. Introducción a la Planificación . . . . .	3
2.1.2. STRIPS . . . . .	4
2.2. PDDL . . . . .	5
2.2.1. Características del Lenguaje . . . . .	5
2.2.2. Ancestros de PDDL . . . . .	7
2.2.3. Evolución de la especificación de PDDL . . . . .	8
2.3. Definición de Problemas de Planificación en PDDL . . . . .	13
2.3.1. Definición de Dominios . . . . .	13
2.3.2. Definición de Problemas . . . . .	14
2.4. Requerimientos PDDL adicionales . . . . .	15
<b>3. Control de Agentes Basados en Planificación Continua</b>	<b>19</b>
3.1. Descripción del Problema . . . . .	19
3.2. Características Principales del Framework . . . . .	20
3.2.1. Algoritmo de Planificación Continua . . . . .	21
3.3. Caso de estudio . . . . .	21
3.4. Integrando al Framework el Traductor PDDL desarrollado . . . . .	22
<b>4. Implementaciones Existentes</b>	<b>25</b>
4.1. Analizador en SWI-Prolog . . . . .	25
4.2. Gramática ANTLR para PDDL . . . . .	27
4.3. Librería PDDL4J . . . . .	31
<b>5. Traducción de requerimientos PDDL</b>	<b>35</b>
5.1. Marco Teórico . . . . .	35
5.1.1. Esquemas de Compilación . . . . .	36
5.1.2. Compilabilidad . . . . .	37
5.2. PDDL <sub>STRIPS</sub> . . . . .	38
5.3. PDDL <sub>L</sub> . . . . .	40
5.4. PDDL <sub>C</sub> . . . . .	42
5.5. PDDL <sub>D</sub> . . . . .	44
5.6. PDDL <sub>u</sub> . . . . .	46

<b>6. Implementación Propuesta</b>	<b>49</b>
6.1. Descripción General de la Solución . . . . .	49
6.1.1. El Lenguaje Fuente . . . . .	49
6.1.2. El Lenguaje Destino . . . . .	50
6.1.3. Módulos de la Arquitectura . . . . .	51
6.2. Ciao Prolog . . . . .	53
6.2.1. Expansiones Sintácticas . . . . .	53
6.2.2. Definite Clause Grammar (DCG) . . . . .	54
6.3. Implementación del Traductor . . . . .	55
6.4. Demostración del Traductor . . . . .	58
<b>7. Conclusiones</b>	<b>63</b>
7.1. Resultados y Contribuciones . . . . .	63
7.2. Trabajo Futuro . . . . .	64



# Índice de figuras

3.1. Arquitectura Modular del Framework de Planificación Continua . . . . .	20
3.2. El subsistema de creencias en la Arquitectura Modular del Framework. . . . .	22
3.3. Arquitectura del Framework y Traductor PDDL . . . . .	23
5.1. Capas de Traducción . . . . .	38
5.2. Expansión Total para Efectos Condicionales . . . . .	42
5.3. Traducción de Precondiciones Disyuntivas . . . . .	44
6.1. Módulo Traductor . . . . .	50
6.2. Arquitectura Modular del Traductor . . . . .	52
6.3. Estado Inicial del “Mundo de Bloques” . . . . .	59
6.4. Estado Final del “Mundo de Bloques” . . . . .	59
6.5. Especificación PDDL en Ciao Prolog . . . . .	59
6.6. Planificador POP en Ciao Prolog . . . . .	60
6.7. Consulta para el planificador POP . . . . .	60
6.8. Resultado obtenido de la planificación . . . . .	61



# Capítulo 1

## Introducción

Esta tesis aborda dos tópicos complementarios implicados en la resolución de problemas utilizando agentes inteligentes: la planificación continua y los lenguajes de representación de problemas.

Russell y Norving [48] definen a la planificación como la tarea de obtener una secuencia de acciones que pueden ser aplicadas a un conjunto discreto de estados para lograr una meta. Puntualmente, la planificación continua está orientada a resolver problemas en ambientes dinámicos y, por lo tanto, se encarga de atacar problemas en ambientes reales.

La complejidad inherente a este tipo de planificación y a los ambientes reales, implica trabajar con otros lenguajes de representación de problemas. Un lenguaje de representación debe permitir la definición de estados, acciones y metas y, además, posibilitar que los algoritmos de planificación puedan obtener ventajas de la estructura lógica de los problemas. Esta necesidad de un lenguaje con un poder expresivo mucho mayor que los existentes hasta el momento, impulsó el desarrollo de un novedoso lenguaje estándar llamado PDDL (*Planning Domain Definition Language*) [33]. PDDL es un lenguaje centrado en acciones y está inspirado en la formulación de problemas de planificación en STRIPS. La importancia de PDDL radica en el nivel de abstracción y la expresividad para modelar dominios y problemas más complejos. Además, trabajar con un lenguaje estándar permite realizar comparaciones de performance entre diferentes algoritmos planificadores.

La principal motivación de esta investigación es dotar al Planificador Continuo, presentado por Moya en [36], de un módulo traductor del lenguaje PDDL con el objetivo de lograr que el sistema de creencias de un agente soporte percepciones y acciones especificadas en dicho lenguaje. Esto nos posibilitará combinar la expresividad de PDDL con el Planificador Continuo y modelar dominios de planificación con un nivel de abstracción necesario en el trato con ambientes reales y dinámicos. La presentación preliminar de esta investigación fue publicada en [6].

Como uno de los resultados principales de esta tesis, se presenta un módulo traductor para un subconjunto del lenguaje PDDL cuyo lenguaje destino es similar a STRIPS. La traducción propuesta involucra un análisis exhaustivo de PDDL y la definición de esquemas de compilación que permitan obtener la representación STRIPS equivalente a las especificaciones PDDL de entrada. Además, se abordan diferentes variantes STRIPS, presentadas en la literatura de Inteligencia Artificial (IA), que surgen de la adición de características más expresivas al STRIPS estándar. En este contexto teórico y como un resultado complementario de esta investigación, se formaliza una nueva variante que involucra el uso de cuantificación universal.

El traductor es implementado como una expansión sintáctica de Ciao Prolog. Esta solución ofrece una notación genérica Prolog que permite adaptar el módulo a diferentes planificadores cuyo lenguaje de representación sea similar a STRIPS. Para ilustrar los resultados obtenidos, se muestra el uso del traductor en el dominio del “Mundo de Bloques” y cómo un planificador, cuyo lenguaje de representación es STRIPS, resuelve un problema especificado inicialmente en PDDL.

## 1.1. Trabajos Relacionados

Algunos resultados preliminares han formado parte de las siguientes publicaciones:

- Braun, G., Moya, M. y Parra, G. Sistemas Multiagentes en Ambientes Dinámicos: Planificación Continua mediante PDDL. En *XIII Workshop de Investigación en Ciencias de la Computación*, Rosario, Santa Fé. Universidad Nacional de Rosario. 2011.
- Braun, G. y Parra, G. Multiagent Systems in Dynamic Environments: Continuous Planning using PDDL. Enviado a las *41 Jornadas Argentinas de Informática*, La Plata, Buenos Aires. Universidad Nacional de La Plata. 2012.
- Moya, M. y Vaucheret, C. Agentes Deliberativos Basados en Planificación Continua. En *X Workshop Agentes y Sistemas Inteligentes (WASI)*, S.S. de Jujuy. Universidad Nacional de Jujuy. Facultad de Ingeniería. 2009.
- Moya, M. y Vaucheret, C. Planificador Continuo como Controlador de Agentes Robots. En *X Workshop de Investigadores en Ciencias de la Computación*, General Pico, La Pampa, Argentina. Universidad Nacional de la Pampa. 2008.

## 1.2. Estructura de la Tesis

En el capítulo 2 se presenta un análisis del lenguaje PDDL, cómo ha sido su evolución y cuáles son sus características más relevantes. El capítulo finaliza con una descripción de los requerimientos del lenguaje considerados para ser implementados en el traductor propuesto.

En el capítulo 3 se introduce el Framework de Planificación Continua, se analizan sus características principales y su implementación. Además, se concluye con la especificación de una nueva arquitectura para el framework.

Las implementaciones existentes, hasta el momento de presentación de este trabajo, se detallan en el capítulo 4. Para cada una de las tres implementaciones abordadas, se consideran las características PDDL que soportan, el ambiente de programación en el que están desarrolladas, la especificación del lenguaje de salida y las licencias correspondientes. En todos los casos, se remarcan las diferencias con nuestra propuesta.

En el capítulo 5 se define el marco teórico subyacente, se estudian variantes de los lenguajes STRIPS y PDDL y se especifican esquemas de compilación que luego serán implementados en el traductor.

Posteriormente, en el capítulo 6, se detalla la arquitectura propuesta para el traductor de PDDL, se analizan los módulos principales de la implementación y se concluye con un ejemplo de aplicación en Ciao Prolog.

Por último, en el capítulo 7, se presentan las conclusiones de esta tesis, sus resultados y contribuciones y se proponen algunas ideas para trabajos futuros.

## Capítulo 2

# El Lenguaje PDDL

El **lenguaje de definición de dominios de planificación (PDDL)** es un lenguaje desarrollado originalmente por el comité organizador de la competencia AIPS-98<sup>1</sup> para la definición de dominios y problemas de planificación. La primera definición del lenguaje fue propuesta por Drew McDermott<sup>2</sup> en 1998.

El lenguaje PDDL es soportado por muchos planificadores y, al igual que STRIPS [15], se basa en la suposición de mundo cerrado<sup>3</sup>. Esto permite que la transformación de estados pueda ser calculada agregando o eliminando literales de la descripción del estado de partida. Está factorizado en un conjunto de características, llamadas *requerimientos*, que permiten a los planificadores determinar si son aptos para trabajar sobre el dominio actual. En la actualidad, se está dedicando mucha investigación y desarrollo sobre su especificación a fin de alcanzar nuevos objetivos tales como la comparación empírica de la performance de los planificadores y el desarrollo de un conjunto estándar de problemas en notaciones comparables.

El propósito del presente capítulo es hacer un breve repaso de los lenguajes de planificación, tomando como referencia al lenguaje STRIPS y analizar PDDL desde su versión inicial, publicada en 1998, hasta la actualización más reciente. Además, mostrar cómo los problemas y los dominios de planificación son expresados en el lenguaje. En la sección 2.1 se estudia la formulación de problemas de planificación y la importancia de contar con un adecuado lenguaje para expresar estos problemas. A continuación, en la sección 2.2, se analiza en profundidad el lenguaje PDDL, sus características, sus predecesores y una cronología de cómo fue evolucionando su especificación. Por último, en la sección 2.3, se describe cómo modelar problemas y dominios en PDDL y se ilustran algunos ejemplos que serán usados a lo largo de este trabajo.

## 2.1. Planificación

### 2.1.1. Introducción a la Planificación

Russell y Norving [48] definen a la **planificación** como la tarea de obtener una secuencia de acciones para lograr una meta y, para hacer esto posible, la planificación necesita de un lenguaje de representación de problemas y de un algoritmo planificador.

Un lenguaje de representación debe permitir la definición de estados, acciones y metas y, además, posibilitar que los algoritmos de planificación puedan obtener ventajas de la estructura lógica de los problemas. La clave es encontrar un lenguaje que sea lo suficientemente expresivo

---

<sup>1</sup> *The Fourth International Conference on Artificial Intelligence Planning Systems 1998*. <http://www.cs.cmu.edu/~aips98/>. Actualmente, AIPS es ICAPS (*International Conference on Automated Planning & Scheduling (ICAPS)*) <http://www.icaps-conference.org/index.php/Main/HomePage>. Disponibles en Septiembre de 2012.

<sup>2</sup> Drew McDermott. <http://cs-www.cs.yale.edu/homes/dvm/>. Disponible en Septiembre de 2012.

<sup>3</sup> *Closed World Assumption*. Asume que el conocimiento es completo. Esto significa que las proposiciones no incluidas en el estado inicial de un problema, son consideradas falsas [45].

para describir una gran cantidad de problemas pero, a su vez, lo suficientemente restrictivo para permitir que los algoritmos operen eficientemente.

Una formulación simple de un problema de planificación consta de los siguientes aspectos:

1. una descripción del *estado inicial* del mundo en algún lenguaje,
2. una descripción de las *metas* del agente, en algún lenguaje, y
3. una descripción de las posibles acciones que pueden ser ejecutadas. Esta última descripción es llamada *teoría de dominio*.

Como ejemplos de lenguajes de representación de problemas podemos mencionar al lenguaje **STRIPS**, que será detallado luego, y a **PDDL**, lenguaje abordado en este trabajo.

Por otro lado, los algoritmos de planificación, conocidos simplemente como **planificadores**, son algoritmos que solucionan problemas (expresados en un determinado lenguaje) produciendo **planes**, es decir, secuencias de acciones. La salida de un planificador consiste en un conjunto de acciones que, cuando son ejecutadas en un mundo que satisface la descripción del estado inicial, permiten alcanzar la meta.

Desde un enfoque teórico, la planificación puede dividirse en dos tipos claramente identificados: la planificación “clásica” y la planificación en ambientes reales o “no clásica”. En la primera, los ambientes en que el agente opera tienen características determinísticas, finitas, estáticas, discretas y completamente observables. Los algoritmos de planificación clásica son, por ejemplo, el Planificador de Order Parcial (POP) [30] y los algoritmos de Búsqueda basados en Gráfos como Graphplan [4], entre otros. Por su parte, en planificación “no clásica”, el ambiente tiene características contrarias a las anteriores, ya que se presenta estocástico y parcialmente observable. Dentro de la planificación no clásica se incluyen a los algoritmos que tratan con restricciones de tiempo y recursos, Redes de Tareas Jerárquicas o HTN<sup>4</sup> [28], entre otros. Además, en [48], Russell y Norving, postulan que otra de las alternativas para trabajar en ambientes reales es la **planificación continua**. Este tipo de planificación será abordada en el capítulo 4.

Luego de haber definido los principales conceptos subyacentes a la planificación, vamos a enfocar nuestro estudio en los lenguajes de representación de problemas. Comenzaremos por STRIPS y luego analizaremos PDDL, lenguaje de particular interés para el presente trabajo.

### 2.1.2. STRIPS

Uno de los primeros lenguajes de representación y uno de los más referenciados en la literatura de Inteligencia Artificial, es STRIPS<sup>5</sup> [15]. La representación STRIPS describe el estado inicial del mundo mediante un conjunto completo de literales básicos (*ground*) y a las metas como una conjunción proposicional. La teoría de dominio, es decir, la descripción formal de las acciones disponibles para el agente, completa la descripción del problema de planificación.

En la representación STRIPS cada acción es descrita por dos fórmulas: la fórmula de precondición y la de poscondición. Ambas están constituidas por una conjunción de literales y definen una función de transición de un mundo a otro. Una acción puede ser ejecutada en cualquier mundo  $w$  que satisfaga la fórmula de precondición. El resultado de ejecutar una acción en un mundo  $w$  es especificado tomando la descripción de  $w$ , adicionando cada literal de la poscondición de la acción y eliminando literales contradictorios.

En general, un esquema de acción en la representación STRIPS consta de tres listas:

- **Lista de Precondiciones:** es una conjunción de literales que deben ser verdaderos en un estado previo para que la acción pueda ser ejecutada.

<sup>4</sup>En Inglés, *Hierarchical Task Networks*.

<sup>5</sup>*Stanford Research Institute Problem Solver*.

- **Lista de Borrados:** es un conjunto de átomos que dejan de ser verdaderos luego de la ejecución de la acción.
- **Lista de Agregados:** es conjunto de átomos que se vuelven verdaderos en el estado posterior a la ejecución de la acción.

STRIPS está basado en la idea de que algunas relaciones en el mundo no son afectadas por la ejecución de una acción. Estas restricciones (entre varias otras como tiempo atómico, no existencia de eventos exógenos, efectos determinísticos en acciones, etc.) permiten trabajar con algoritmos de planificación más simples y eficientes, pero dificultan la descripción de problemas más complejos o de problemas reales.

A continuación, mostraremos un ejemplo de la representación STRIPS para el conocido problema del “Mundo de Bloques”<sup>6</sup>.

**Ejemplo 2.1.1.** *La acción `pickup(X,Y)` permite tomar el bloque `X` desde el bloque `Y` (cuando `X` y el agente están libres) y, la acción `stack(X,Y)` permite apilar el bloque `X` sobre el `Y` (cuando el agente tiene `X` e `Y` está libre). El problema STRIPS especifica un posible estado inicial.*

```
% pickup(X,Y)
Precondiciones: clear(X), armempty
Borrados: clear(X), armempty
Agregados: holding(X)

% stack(X,Y)
Precondiciones: clear(Y), holding(X)
Borrados: clear(Y), holding(X)
Agregados: armempty, clear(X), on(X,Y)

% Problema STRIPS
holds(ontable(c),init)
holds(ontable(b),init)
holds(on(a,c),init)
holds(clear(a),init)
holds(clear(b),init)
holds(armempty,init)
```

Nombraremos a este formalismo **STRIPS estándar**. Dicho nombre nos permitirá distinguir este lenguaje de las demás variantes STRIPS, al momento de clasificarlas, según sus respectivos niveles de expresividad.

## 2.2. PDDL

### 2.2.1. Características del Lenguaje

La necesidad de un lenguaje con un poder expresivo mucho mayor que los existentes hasta el momento, impulsó, a fines de los 90's, el desarrollo del lenguaje de representación PDDL

---

<sup>6</sup>Los problemas del Mundo de Bloques son problemas típicos en Inteligencia Artificial. Es un mundo ficticio en el que sólo existen bloques que pueden encontrarse sobre una mesa o apilados. Un plan para resolverlo debe reordenar los bloques para conseguir una meta. Se permite mover un bloque (siempre que éste no tenga ninguno encima) a la mesa o encima de otro bloque.

(*Planning Domain Definition Language*) [33]. En la actualidad, PDDL es considerado el *estándar de facto*<sup>7</sup> de los lenguajes de representación.

PDDL es un lenguaje *action-centred* (centrado en acciones) y está inspirado en la formulación de los ya nombrados problemas de planificación en STRIPS. Su núcleo es una simple estandarización de la sintaxis para expresar la semántica familiar de acciones, usando pre y pos condiciones para describir la aplicabilidad y los efectos de las acciones. Su sintaxis es similar a la del lenguaje Lisp<sup>8</sup> [51], por lo tanto, la estructura de una definición es una lista de expresiones parentizadas. PDDL intenta expresar la *física* del dominio, es decir, cuáles son los predicados, qué acciones son posibles, cuál es la estructura de las acciones compuestas y cuáles son los efectos de las acciones.

El lenguaje separa la definición del dominio (con acciones parametrizadas) de la definición de los problemas (con objetos específicos, metas y condiciones iniciales). Un problema de planificación es creado asociando una descripción de un dominio y una descripción de un problema, permitiendo que una misma descripción de dominio pueda ser asociada a diferentes problemas. Esto permite expresar distintos problemas de planificación sobre un mismo dominio.

A pesar de que el núcleo de PDDL es el formalismo STRIPS, PDDL se extiende más allá de este simple lenguaje. Este poder expresivo extendido incluye la capacidad de definir estructuras de tipos, tipos de parámetros, restricciones, acciones con precondiciones negativas y efectos condicionales y el uso de cuantificación en pre y pos condiciones.

El siguiente es un ejemplo básico PDDL del “Mundo de Bloques”, equivalente a la especificación STRIPS presentada en el ejemplo 2.1.1.

### Ejemplo 2.2.1.

```
% Dominio PDDL
(define (domain bkw)
  (:requirements :strips)
  (:predicates (clear ?x)
               (ontable ?x)
               (armempty)
               (holding ?x)
               (on ?x ?y))

  (:action pickup
    :parameters (?ob)
    :precondition (and (clear ?ob) (armempty))
    :effect (and (holding ?ob) (not (clear ?ob)) (not (armempty))))

  (:action stack
    :parameters (?ob ?underob)
    :precondition (and (clear ?underob) (holding ?ob))
    :effect (and (clear ?ob) (on ?ob ?underob) (armempty)
                (not (clear ?underob)) (not (holding ?ob))))
)

% Problema PDDL
(define (problem pb1)
  (:domain bkw)
  (:objects a b c)
```

<sup>7</sup>Un estándar de facto es una norma que se caracteriza por no haber sido consensuada ni legitimada por un organismo de estandarización. Se trata de una norma generalmente aceptada y ampliamente utilizada por iniciativa propia de un gran número de interesados.

<sup>8</sup>*LISt Processing*.



```
(:goal (on a b))
(:init (ontable c) (ontable b)
      (on a c) (clear a) (clear b) (armempty))
)
```

PDDL incluye, a través de la etiqueta **requirements**, una representación sintáctica sobre el nivel de expresividad requerido en la descripción de un dominio particular. Esta definición de requerimientos permite identificar el poder expresivo de estos problemas y qué planificadores son capaces de manejar estas características del lenguaje.

### 2.2.2. Ancestros de PDDL

PDDL es descendiente de los siguientes formalismos:

- **ADL** [44]. El lenguaje ADL<sup>9</sup> fue propuesto por Pednault<sup>10</sup> como un formalismo de planificación con el poder expresivo del Calculo Situacional [31] y los beneficios computacionales del lenguaje STRIPS. ADL soporta *equality* como un predicado *built-in* y las precondiciones pueden contener negación, disyunción y cuantificación. Además, permite efectos condicionales y los objetos pueden tener tipos asociados.
- **SIPE-2** [59]. El sistema de planificación SIPE-2<sup>11</sup> provee un formalismo para descripción de acciones y utiliza el conocimiento contenido en el formalismo, junto con sus heurísticas de combinatoria de problemas, para generar planes y alcanzar las metas en diversos problemas de planificación. SIPE-2 es un sistema de planificación de orden parcial, jerárquico e independiente del dominio.
- **Prodigy4.01** [9]. Prodigy es una arquitectura para resolver problemas independiente del dominio usada en planificación, aprendizaje y adquisición de conocimiento. El poder de Prodigy es, en parte, debido a la flexibilidad y expresividad de su lenguaje de representación. Los dominios de planificación consisten de tipos para las entidades, un conjunto de operadores, reglas de inferencia y reglas para el control de búsqueda. El lenguaje de representación es una extensión de STRIPS.
- **UMCP** [29]. UMCP<sup>12</sup> es una formalización del procedimiento de planificación HTN<sup>13</sup> [28]. Hereda la sintaxis y semántica del formalismo HTN con los conceptos de *Goal tasks*, *Primitive tasks*, *Compound tasks*. Las *Goal tasks* son equivalentes a las metas en STRIPS; las *Primitive tasks* son tareas que pueden lograrse ejecutando la acción correspondiente; y las *Compound tasks* denotan cambios deseados que involucran varias *goal tasks* and *primitive tasks*. Todas estas tareas son conectadas mediante *task networks*, también llamadas “redes procedurales”.
- **UNPOP** [32]. El planificador UNPOP<sup>14</sup> no está restringido sólo a representaciones STRIPS, sino que también, soporta lenguajes más expresivos como ADL. Fue desarrollado por Drew McDermott.

---

<sup>9</sup> *Action Description Language*.

<sup>10</sup> Edwin P. Pednault. <http://researcher.watson.ibm.com/researcher/view.php?person=us-pednault>. Disponible en Septiembre de 2012.

<sup>11</sup> *System for Interactive Planning and Execution*.

<sup>12</sup> *Universal Method Composition Planner*.

<sup>13</sup> *Hierarchical Task-Network*.

<sup>14</sup> *un-Partial Order Planner*.

- **UCPOP** [3]. El lenguaje de representación de UCPop<sup>15</sup> es un subconjunto de KRSL<sup>16</sup> [2] que se corresponde estrechamente a ADL con respecto al contenido expresivo y al lenguaje de representación de Prodigy. En particular, el lenguaje de representación de UCPop permite definir acciones usando efectos condicionales y cuantificaciones anidadas. También, permite la declaración de axiomas de dominio, restricciones y hechos.

### 2.2.3. Evolución de la especificación de PDDL

La Competencia Internacional de Planificación (IPC)<sup>17</sup> es una importante referencia para la investigación de tópicos relacionados a la planificación y la adopción de PDDL como un lenguaje común. La especificación de problemas es uno de los resultados más importantes de este tipo de competiciones.

La evolución de PDDL en el tiempo tiene algunos hitos importantes, principalmente en sucesivas competiciones, con la definición de nuevas extensiones del lenguaje. Las versiones reconocidas son: **PDDL 1.2** (IPC 1998), **PDDL 2.1** (IPC 2002), **PDDL 2.2** (IPC 2004), **PDDL 3.0** (IPC 2006) y **PDDL 3.1** (IPC 2008). También, existen algunas otras extensiones como **PDDL+** [17] y **Opt** [34], pero están fuera del alcance del presente trabajo.

A continuación, y en forma cronológica, vamos a describir las características más importantes de cada versión del lenguaje.

#### PDDL 1.2

La versión original de PDDL fue la 1.2 del año 1998 [35]. Esta versión soporta acciones básicas de estilo STRIPS (precondiciones y efectos), efectos condicionales<sup>18</sup> y cuantificación universal sobre universos dinámicos. Además, permite especificar restricciones de seguridad<sup>19</sup> y jerarquía de subacciones y submetas. Por último, incluye la posibilidad de reutilizar dominios en varios planificadores que manejen diferentes niveles de expresividad.

A continuación, damos una breve descripción de las principales características de PDDL 1.2.

- **Efectos condicionales.** Los efectos condicionales permiten definir acciones cuyos efectos son dependientes del contexto. Para esto, PDDL define una sentencia **when** que tiene dos argumentos, antecedente y consecuente. Este requerimiento PDDL será ampliado en la sección 2.4.
- **Cuantificación universal sobre universos dinámicos.** Esta cuantificación permite que los efectos de las acciones puedan crear y eliminar nuevos objetos. Es posible modelar la creación de objetos teniendo el efecto (**book dict**) y también la destrucción de objetos con el efecto (**not (book dict)**). Sin embargo, para definir el universo de un tipo dado (por ejemplo, **book**), es necesario considerar todos los **books** que son creados por las acciones, posiblemente ejecutadas, antes de la acción actual.
- **Restricciones de Seguridad.** Las restricciones de seguridad son metas *background* que deben ser verdaderas al finalizar el plan. Cualquier violación sobre alguna restricción debe ser salvada al momento de finalización del plan. En PDDL, se declara como: **safety-constraints**. A continuación, presentamos un ejemplo.

**Ejemplo 2.2.2.** *El ejemplo muestra una restricción para un robot que evita eliminar archivos que no tienen copia de seguridad:*

<sup>15</sup> *Universal, Conditional Partial-Order Planner.*

<sup>16</sup> *Knowledge Representation Specification Language.*

<sup>17</sup> *International Planning Competition.* La IPC se desarrolla en el contexto de la ICAPS y tiene, como metas, estudiar el estado del arte de los sistemas de planificación automáticos, la evaluación de diferentes enfoques a la planificación automatizadas y promover la aceptación y aplicación de estas tecnologías.

<sup>18</sup> En Inglés, *conditional effects*.

<sup>19</sup> En Inglés, *safety constraints*.

```
(:safety
  (forall (?f)
    (or (file ?f)
        (written-to-tape ?f)))
  )
```

## PDDL 2.1

La versión original recién fue modificada hacia el año 2002. PDDL 2.1 [16] adiciona dos importantes características: acciones durativas<sup>20</sup> y funciones objetivo<sup>21</sup>.

- **Acciones durativas.** Las acciones durativas son aquellas que requieren una determinada cantidad de tiempo para ejecutarse. En PDDL 2.1 se definieron dos formas de acciones durativas: discretizadas y continuas. En la especificación del lenguaje, Maria Fox<sup>22</sup> y Derek Long<sup>23</sup>, definen la diferencia entre ambas acciones. Las acciones discretizadas son aquellas que toman una cantidad fija de tiempo y un ejemplo claro de esto es un agente viajando entre dos ciudades. Sin embargo, este tipo de acciones puede condicionar la performance del agente teniendo en cuenta que podría completar la acción en menor tiempo que el definido. Por otro lado, las acciones continuas no parecen tener este problema ya que permiten al agente detener el proceso en cualquier punto consistente con las restricciones definidas sobre la duración de la acción.

Una acción durativa es definida de la siguiente manera:

**Ejemplo 2.2.3.** *La siguiente acción define la carga de un transporte sin restricciones de capacidad.*

```
(:durative-action load-truck
  :parameters (?t - truck) (?l - location)
              (?o - cargo) (?c - crane)
  :duration (= ?duration 5)
  :condition (and (at start (at ?t ?l))
                  (at start (at ?o ?l))
                  (at start (empty ?c))
                  (over all (at ?t ?l))
                  (at end (holding ?c ?o))
  :effect (and (at end (in ?o ?t))
               (at start (holding ?c ?o))
               (at start (not (at ?o ?l)))
               (at end (not (holding ?c ?o))))
  )
```

- **Funciones objetivo.** La inclusión de métricas en PDDL es consecuencia de la adopción de una extensión numérica estable (esto es, permitir funciones de tipo  $Object^n \rightarrow \mathbb{R}$  para una colección finita de objetos y funciones finitas de aridad  $n$ ). El nuevo campo opcional se define como `:metric` y es incluido en la especificación de problemas PDDL. Las métricas permiten especificar cómo un plan será evaluado y permiten a los modeladores explorar los

<sup>20</sup>En Inglés, *durative actions*.

<sup>21</sup>En Inglés, *objective functions*.

<sup>22</sup>Maria Fox. <https://personal.cis.strath.ac.uk/~maria/>. Disponible en Septiembre de 2012.

<sup>23</sup>Derek Long. <https://personal.cis.strath.ac.uk/~derek/>. Disponible en Septiembre de 2012.

efectos de diferentes métricas en la construcción de soluciones a problemas con un mismo dominio.

El siguiente ejemplo muestra la definición de una métrica:

**Ejemplo 2.2.4.** *La métrica define una función que minimiza el consumo de combustible total.*

```
(:metric minimize (total-fuel-used))
```

En contraposición y, luego del ejemplo anterior, la inclusión de métricas en problemas de planificación es compleja y requiere un manejo cuidadoso.

## PDDL 2.2

La tercer actualización de PDDL fue publicada en el año 2004. La versión 2.2 [14] incluye, como nuevas características, a los predicados derivados<sup>24</sup> y a los literales acotados en tiempo<sup>25</sup>.

- **Predicados derivados.** Los predicados derivados son aquellos que no son afectados por ninguna de las acciones disponibles para el planificador. Los valores de verdad de estos predicados son derivados a partir de un conjunto de reglas de la forma  $P \Rightarrow Q$ . Los dominios PDDL que incluyen predicados derivados deben declarar el requerimiento **durative-predicates**. A continuación, damos un ejemplo de cómo PDDL define estos predicados.

**Ejemplo 2.2.5.** *Dados los bloques  $x$  e  $y$ , el predicado **above** es verdadero si  $x$  está transitivamente sobre  $y$ .*

```
(:derived (above ?x ?y)
  (or on(?x ?y)
    exists (?z) (and (on ?x ?z)
                      (above ?z ?y))))
)
```

Estos predicados combinan varios aspectos claves. Ellos proveen una manera concisa y conveniente para expresar la clausura transitiva de una relación. Los predicados pueden ser compilados a un formalismo menos expresivo introduciendo acciones y hechos equivalentes. Por último, no causan un *overhead* significativo en algunos planificadores, como por ejemplo, *Forward Search* [47].

- **Literales acotados en tiempo.** Son una manera muy simple de expresar un evento restringido en el tiempo. Los hechos se volverán *true* o *false* en puntos del tiempo que son conocidos, previamente por el planificador, sin considerar las acciones que son eligidas para ejecutar. Esta extensión de PDDL es relevante ya que eventos determinísticos son muy comunes en el mundo real (por ejemplo, tiempo en el que una persona trabaja). Un dominio PDDL que utiliza estos literales debe incluir el requerimiento **timed-initial-literals**. El siguiente ejemplo indica cómo incluir este requerimiento en un dominio PDDL.

**Ejemplo 2.2.6.** *La acción **go-shopping** requiere que el **shop** esté abierto como precondición. El **shop** abre al público a las 9 horas y cierra sus puertas a las 20 horas.*

<sup>24</sup>En Inglés, *Derived predicates*.

<sup>25</sup>En Inglés, *Timed initial literals*.

```
(:init
  (at 9 (shop-open))
  (at 20 (not (shop-open)))
)
```

### PDDL 3.0

La generación de planes de *buena calidad* es uno de los intereses centrales en aquellos dominios de planificación que modelan problemas del mundo real. Esta extensión de PDDL propone una mejor caracterización para obtener planes de calidad permitiendo la definición de restricciones sobre la estructura de los planes. La versión del lenguaje es la 3.0 [21], [22] y fue actualizada en 2006. Incluye restricciones<sup>26</sup> y preferencias<sup>27</sup>.

- **Restricciones.** PDDL define restricciones sobre un conjunto válido de planes expresadas en lógica temporal. Las restricciones establecen condiciones que deben ser cumplidas por la secuencia de estados visitados durante la ejecución del plan. Las condiciones son expresadas con operadores modales, como por ejemplo, **always**, **sometime**, **at-most-once** y **at-end**. Las restricciones son incluidas en problemas y dominios PDDL usando la declaración **:constraints**<sup>28</sup>. A continuación, mostramos un ejemplo con la definición de una restricción.

**Ejemplo 2.2.7.** *El siguiente ejemplo restringe la cantidad de bloques que pueden estar apilados sobre otro. En este caso, no más de un bloque.*

```
(:constraints
  (and (always (forall (?b1 ?b2 - block)
    (implies (and (fragile ?b1) (on ?b2 ?b1))
      (clear ?b2))))))
)
```

- **Preferencias.** Las preferencias son también denominadas restricciones *soft*. Son restricciones que no necesitan ser satisfechas por un plan pero decrementan la calidad del mismo si no se satisfacen. La sintaxis de las preferencias tiene dos partes bien identificadas. Primero, la identificación de la restricción y segundo, la descripción de cómo la satisfacción, o la falta de ella, afecta la calidad del plan.

Las preferencias pueden introducirse en dominios y problemas declarando la cláusula **:preferences**. Las preferencias sobre las restricciones de estados son expresadas luego de la definición de **(:constraints ...)** y las preferencias sobre la meta son expresadas en **(:goal ...)**.

Los próximos ejemplos muestran definiciones de preferencias en dominios y problemas.

**Ejemplo 2.2.8.** *La restricción incluye la preferencia por apilar bloques del mismo color.*

```
(:constraints
  (and (preference
    (always (forall (?b1 ?b2 - block ?c1 ?c2 - color)
      (implies (and (on ?b1 ?b2)
        (color ?b1 ?c1))
```

<sup>26</sup>En Inglés, *Constraints*.

<sup>27</sup>En Inglés, *Preferences*.

<sup>28</sup>**constraints** pueden ser vistas como **safety-constraints**.

```

(color ?b2 ?c2))
(= ?c1 ?c2))))))
)

```

**Ejemplo 2.2.9.** *La siguiente meta implica que el `package1` esté en `london` con la restricción de que el transporte esté vacío.*

```

(:goal (and (at package1 london)
             (preference (clean truck1)))
)

```

### PDDL 3.1

Finalmente, la última versión de PDDL que estudiaremos es la versión 3.1 del año 2008 [27]. Entre las principales extensiones incluye *fluents* de objetos<sup>29</sup> y costos en acciones<sup>30</sup>.

- **Fluents objeto.** Los *fluents* de objetos son análogos a los *fluents* numéricos<sup>31</sup>, sin embargo, el rango de estos *fluents* también puede incluir objetos del dominio. Este nuevo requerimiento es definido en PDDL como `:object-fluents` y, cuando es usado, tanto la definición de dominios como los problemas pueden hacer uso de estos *fluents*.

**Ejemplo 2.2.10.** *El ejemplo muestra la definición de un fluent (`on-block ?x - block`) - `block` cuyo rango es un objeto tipo `block` que puede ser retornado por la función `on-block ?x`. El argumento de la función también debe ser un objeto. En el segundo fluent, la función también retorna el bloque que está in-hand.*

```

(:functions (in-hand) - block
            (on-block ?x - block) - block)
)

```

- **Costos en acciones.** Cuando el requerimiento es incluido en una especificación PDDL, mediante `:action-costs`, el uso de *fluents* numéricos es habilitado. Estos costos en acciones enfatizan aún más la importancia de la calidad del plan.

**Ejemplo 2.2.11.** *Para incluir costos en acciones es necesario definir un fluent que permita registrar el costo, especificar que el fluent es modificado por la ejecución de una acción, inicializar el fluent y definir una métrica. A continuación, mostramos un ejemplo acotado con un dominio y un problema PDDL con estas definiciones.*

```

(define (domain travel)
...
  (:functions (distance ?from ?to)
              (total-cost))
  (:action go

```

<sup>29</sup>En Inglés, *object fluents*.

<sup>30</sup>En Inglés, *action-costs*.

<sup>31</sup>Un *fluent* es una variable/predicado de estado cuyo valor es numérico y que permiten modelar recursos tales como tiempo, energía, distancia, etc. en dominios PDDL. El rango de estos *fluents* son números enteros o reales. Por ejemplo, el *fluent* numérico (`total-cost`) mantiene el costo total de una acción. Entonces, para que el valor de la acción sea computado, su efecto deberá definir el incremento de este costo de la siguiente manera: `:effect (increase (total-cost))`.

```

:parameters (?from ?to)
:precondition (and (in ?from) (road ?from ?to))
:effect (and (not (in ?from)) (in ?to)
            (increase (total-cost) (distance ?from ?to))))
)

(define (problem travelp)
...
(:init ...
  (= (distance A B) 10)
  (= (distance A C) 35)
...
)
...
)

```

## 2.3. Definición de Problemas de Planificación en PDDL

Una definición PDDL consiste de dos partes: un dominio y un problema. Ambos pueden o no estar definidos en archivos separados pero, esto también, puede ser una condición del planificador que se esté utilizando.

A continuación, se detallan la estructuras correspondientes a dominios y problemas PDDL.

### 2.3.1. Definición de Dominios

La definición de un dominio contiene predicados, operadores (llamados *acciones* en PDDL) y requerimientos. Los dominios básicos PDDL tienen la siguiente sintaxis:

```

(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:equality] [:typing] [:adl])
  (:typing ...)
  (:constants ...)
  (:predicates (PREDICATE_1_NAME ?A1 ?A2 ... ?AN)
               (PREDICATE_2_NAME ?A1 ?A2 ... ?AN)
               ...)
  (:action ACTION_1_NAME
    [:parameters (?P1 ?P2 ... ?PN)]
    [:precondition PRECOND_FORMULA]
    [:effect EFFECT_FORMULA]
  )
  (:action ACTION_2_NAME
    ...)
  ...)

```

donde cada etiqueta tiene la siguiente descripción:

- **Requerimientos:** el campo `:requirements` incluye una lista de los requerimientos usados en la definición del dominio. Si ningún requerimiento es especificado, el poder expresivo del problema de planificación es equivalente al del lenguaje STRIPS.

- **Tipos:** el campo `:typing` provee una declaración de tipos. Es una lista de nombres donde cada uno de ellos representa un tipo diferente.

**Ejemplo 2.3.1.** *La sentencia `(:types agent wumpus gold square)` define cuatro tipos de objetos `agent`, `wumpus`, `gold` y `square`.*

- **Constantes:** el campo `:constants` puede declarar una lista de nombres con tipos o bien una de nombres sin tipos. Una lista de constantes con tipos es simplemente una lista de nombres los cuales pueden incluir tipos precedidos por un signo `'-'` (menos). Este campo es opcional.

**Ejemplo 2.3.2.** *La sentencia `(:constants b1 b2 - block)` declara dos constantes: `b1` y `b2` de tipo `block`.*

- **Predicados:** el campo requerido `:predicates` consiste de una lista de declaraciones de predicados. La lista indica qué predicados serán usados en el dominio y cuáles son los argumentos y la aridad de cada uno. En caso de que se hayan declarado tipos, la lista de variables es la misma que la declarada para la definición de constantes con tipos. El predicado de igualdad `'='` es un predicado predefinido que puede tomar dos argumentos de cualquier tipo.

**Ejemplo 2.3.3.** *La sentencia `(:predicates (clear ?x) (on-table ?x))` declara los predicados `clear(x)` y `on-table(x)` ambos de aridad 1.*

- **Acciones:** la declaración de una acción involucra tres campos `:parameters` (parámetros), `:precondition` (precondición) y `:effect` (efecto). El campo parámetros es una lista de variables, es decir, los argumentos sobre los cuales operan los predicados. Los parámetros pueden también incluir tipos. Los campos precondición y efecto son conjunciones de literales. En ambos casos, todas las variables de estos componentes deben aparecer en la lista de parámetros.

**Ejemplo 2.3.4.** *La acción `stack` tiene los parámetros `ob` y `underob`. La precondición y efectos son conjunciones con variables que están en la lista de parámetros. Los predicados son aquellos definidos en la sección `predicates` del dominio PDDL.*

```
(:action stack
  :parameters (?ob ?underob)
  :precondition (and (clear ?underob) (holding ?ob))
  :effect (and (clear ?ob) (on ?ob ?underob) (armempty)
              (not (clear ?underob)) (not (holding ?ob))))
)
```

Una instancia de una acción es aplicable a un estado  $S$ , si y sólo si, su precondición es verdadera en  $S$ . El nuevo estado  $S'$  es generado adicionando todos los átomos positivos que aparecen en la lista de efectos y eliminando desde  $S$  todas las fórmulas atómicas negativas que aparecen en la lista de efectos. Las restantes fórmulas atómicas *ground* verdaderas son también agregadas a  $S$ .

### 2.3.2. Definición de Problemas

Un problema es lo que un planificador trata de resolver y es definido con respecto a un dominio. Básicamente, un problema especifica una situación inicial y una meta a ser alcanzada. Los problemas PDDL tienen la siguiente sintaxis:



```
(define (problem PROBLEM_NAME)
  (:domain DOMAIN_NAME)
  (:objects OBJ1 OBJ2 ... OBJ_N)
  (:init ATOM1 ATOM2 ... ATOM_N)
  (:goal CONDITION_FORMULA)
)
```

donde cada etiqueta tiene la siguiente descripción:

- **Dominio:** el campo `:domain` declara el dominio asociado al problema actual.

**Ejemplo 2.3.5.** La sentencia `(:domain typed-blocksworld)` indica que el problema está siendo definido para el dominio llamado `typed-blocksworld`.

- **Estado Inicial:** el campo `:init` incluye una lista con todos los átomos *ground* que son verdaderos en el estado inicial.

**Ejemplo 2.3.6.** La sentencia `(:init (clear A) (on A B) (on B C) (on-table C) (empty H))` define que los átomos `clear(A)`, `on(A,B)`, `on(B,C)`, `on-table(C)` y `empty(H)` son verdaderos.

- **Objetos:** el campo `:objects` lista los objetos que existen en el problema y puede ser un super conjunto de aquellos objetos que aparecen en la definición del estado inicial. En el caso que la definición incluya tipos, el campo, es una lista de objetos con tipos inmediatamente precedidos por el signo ‘-’ (menos). Este campo es obligatorio.

**Ejemplo 2.3.7.** La siguiente definición de objetos `(:objects H - hand A B C - block)` declara que el objeto `H` es de tipo `hand` y que los objetos `A`, `B`, `C` son de tipo `block`.

- **Meta:** la meta, `:goal`, de la definición de un problema, es una conjunción de átomos *ground*.

**Ejemplo 2.3.8.** La meta para el problema actual es `(:goal (and (on C B) (on B A)))`

A diferencia de las precondiciones en las acciones del dominio, tanto el estado inicial como el final, tienen que contener fórmulas *ground*. Esto significa que los argumentos de todos estos predicados deben estar instanciados a objetos o nombres de constantes.

La solución a un problema es una serie de acciones tal que:

1. La secuencia de acciones, incluidas en el plan, es aplicable a partir de la situación inicial, y
2. La meta es verdadera en la situación que resulta de la ejecución de dicha secuencia de acciones.

## 2.4. Requerimientos PDDL adicionales

Como mencionamos en la sección 2.2, el núcleo de PDDL es el formalismo STRIPS. Este formalismo es soportado por la mayoría de los planificadores existentes, como por ejemplo, POP [30], entre otros. Sin embargo, PDDL se extiende más allá de este simple lenguaje STRIPS. El incremento en la complejidad del lenguaje hace que muchos de los planificadores estudiados se vuelvan obsoletos para tratar con características que incluyen, por ejemplo, lógicas temporales.

Ante este problema, existen dos alternativas principales de solución. La primera implica la adaptación del algoritmo de planificación para manejar extensiones de PDDL. Este enfoque es, normalmente, más eficiente pero su implementación puede ser compleja.

Por otro lado, la segunda propuesta es *compilar* los requerimientos adicionales generando una salida equivalente en un lenguaje más simple, en nuestro caso, STRIPS. Este último enfoque es el adoptado para nuestro trabajo. De esta manera, logramos combinar la expresividad de PDDL con un planificador de tipo continuo que analizaremos con mayor detalle en el próximo capítulo.

Debido a que PDDL es un lenguaje muy general y muchos planificadores soportan sólo un subconjunto de él, los dominios declaran requerimientos. Los requerimientos definen la semántica subyacente en los problemas de planificación en los que son declarados y permiten clasificar a los planificadores según los requerimientos que pueden procesar estos algoritmos. El requerimiento por defecto para cualquier especificación PDDL es **strips**.

En esta sección vamos a detallar cada uno de los requerimientos PDDL incluidos en nuestro trabajo. Como parte de este trabajo de tesis, cada uno de estos requerimientos es *compilado* generando una salida equivalente en lenguaje STRIPS.

Una lista más exhaustiva de todos los requerimientos que incluye el lenguaje puede encontrarse en [27].

## Strips

Como mencionamos previamente, STRIPS es el lenguaje básico de representación y es el núcleo de PDDL. Los detalles de STRIPS fueron discutidos en la sección 2.1.2. La declaración de este requerimiento en un dominio PDDL se realiza mediante **:strips**.

## Igualdad

Este requerimiento indica que la descripción del modelo soporta el operador '=' como un predicado *built-in*. La declaración de este requerimiento en un dominio PDDL se realiza mediante **:equality**.

La siguiente es una acción sobre el problema del “Mundo de Bloques” que utiliza este predicado para la comparación de parámetros.

**Ejemplo 2.4.1.** *En este caso, la acción **stack** permite apilar un bloque sólo si el destino es la mesa. **table** es una constante.*

```
(:action stack
  :parameters (?X ?Y)
  :precondition (and (clear table) (= ?Y table))
  :effect (and (on ?X table) (not (clear table)))
)
```

## Efectos condicionales

Los efectos condicionales son utilizados para describir acciones cuyos efectos son dependientes del contexto. Sintácticamente, se incluye una cláusula especial **when** en los efectos de las acciones que recibe dos argumentos: *antecedente* y *consecuente*. La acción tendrá los efectos del consecuente si el antecedente se cumple inmediatamente antes de la ejecución de la acción. El antecedente es llamado precondición secundaria y, por lo tanto, se refiere al estado del mundo antes que la acción sea ejecutada, mientras que el consecuente, se refiere al estado del mundo luego de la ejecución de dicha acción. Los efectos condicionales son usados en PDDL declarando el requerimiento **conditional-effects**.

A continuación, ilustramos el uso los efectos condicionales con un ejemplo.

**Ejemplo 2.4.2.** *Redefinimos la acción **stack** del “Mundo de Bloques” para incluir efectos condicionales. En esta nueva acción, el bloque **?X** será apilado sobre **?Z** siempre que **?X** esté previamente apilado sobre **?Y**.*

```
(:action stack
  :parameters (?X ?Y ?Z)
  :precondition (and (clear ?X) (clear ?Z) (on ?X ?Y))
  :effects (and (on ?X ?Z) (clear ?Y) (not (on ?X ?Y))
               (when (block ?Z) (not (clear ?Z))))
)
```

*Analicemos el efecto condicional definido para la acción **stack**. La cláusula (**when** (**block** ?Z) (**not** (**clear** ?Z))) define la precondition secundaria (**block** ?Z), por lo tanto, (**not** (**clear** ?Z)) será agregado al conjunto de efectos de la acción si ?Z es un bloque.*

### Precondiciones disyuntivas

Este requerimiento implica la inclusión de la cláusula especial ‘**or**’ en la definición de las acciones en PDDL. Las precondiciones disyuntivas permiten expresar que una acción puede ser ejecutada si sólo algunas de las precondiciones son verdaderas.

En el siguiente ejemplo vemos cómo se modela en PDDL usando disyunción en las precondiciones de las acciones.

**Ejemplo 2.4.3.** *Esta acción modela una variante de la acción **stack** del “Mundo de Bloques” presentada previamente. La precondition incluye el nuevo requerimiento **or** estableciendo que un objeto ?X puede ser apilado sobre ?Z si ?Z es la mesa (**istable**) u otro bloque, sin ningún otro bloque apilado.*

```
(:action stack
  :parameters (?X ?Y ?Z)
  :precondition (and (or (istable ?Z) (clear ?Z))
                   (clear ?X) (on ?X ?Y))
  :effect (and (on ?X ?Z) (clear ?Y)
              (not (clear ?Y)) (not (on ?X ?Y)))
)
```

### Precondiciones universales

La definición de este requerimiento implica la inclusión de una nueva cláusula especial, llamada **forall**, en las especificaciones de las acciones PDDL. La cuantificación universal permite modelar acciones que serán ejecutadas si todos los objetos, considerados en el mundo que se está modelando, cumplen con la precondition correspondiente.

En el siguiente ejemplo vemos como las precondiciones las precondiciones universales son empleadas en una especificación PDDL:

**Ejemplo 2.4.4.** *Esta es una variante de la acción **stack** del “Mundo de Bloques”. La acción permitirá apilar un bloque sobre otro si todos los bloques, especificados en la definición del problema, están posicionados sobre la mesa. Notar que los bloques sobre la mesa pueden estar, a su vez, también apilados sobre otros.*

*Antes de definir la acción **stack** vamos a declarar el problema PDDL con los respectivos objetos.*

```
(define (problem pb1)
  (:domain bkwup)
  (:objects a b c)
  (:goal (on a b))
  (:init (ontable c) (ontable b) (ontable a)
         (on a c) (clear a) (clear b) (armempty))
)
```

*Luego, la definición de la acción **stack** es la siguiente:*

```
(:action stack
:parameters (?ob ?underob)
:precondition (and (forall (?block) (ontable ?block))
                  (clear ?underob) (holding ?ob))
:effect (and (clear ?ob) (on ?ob ?underob) (armempty)
            (not (clear ?underob)) (not (holding ?ob))))
```

En resumen, hicimos un breve repaso de los formalismos de planificación comenzando con el lenguaje STRIPS. Luego, estudiamos las principales características de PDDL, su evolución como estándar y algunos de sus ancestros. Por último, analizamos con más detalle los requerimientos PDDL que serán compilados a STRIPS como parte de nuestro trabajo de tesis.

En el próximo capítulo presentaremos el Framework de Planificación Continua, repasaremos sus principales características y analizaremos la integración del traductor PDDL a su arquitectura modular.

## Capítulo 3

# Control de Agentes Basados en Planificación Continua

La problemática abordada por Moya en [36], [37], [38] y [39] es la planificación en ambientes dinámicos y no determinísticos. El objetivo principal es la implementación de un agente de propósito general que pueda operar en dominios del mundo real lidiando con múltiples eventos y con otros agentes.

El trabajo presenta un framework para la implementación de agentes deliberativos cuyo razonamiento está basado en planificación continua. La implementación está íntegramente desarrollada en Ciao Prolog [7]. Lo novedoso de este tipo de planificadores es que están aptos para ambientes reales permitiendo al agente persistir indefinidamente en su entorno, es decir, que no se detiene al alcanzar una meta sino que la ejecución continúa mediante la formulación de nuevas metas. De esta manera, el agente continúa planificando y actuando. Algunos experimentos con esta implementación fueron realizados bajo el dominio del Fútbol de Robots, utilizando Rakiduum. Rakiduum es un equipo de fútbol de robots con licencia GNU<sup>1</sup> que ha participado en numerosas ediciones del Campeonato Argentino de Fútbol con Robots (CAFR) con resultados más que satisfactorios. Algunos de ellos fueron publicados en [25], [26] y [53].

En el presente capítulo se estudia con más profundidad esta implementación. En la sección 3.1 se hace un breve repaso del problema planteado por Moya. En la sección 3.2 se detallan las características del Framework y el Planificador Continuo que implementa. Por último, en la sección 3.4, se explica la integración del traductor PDDL con el presente framework basado en planificación continua.

### 3.1. Descripción del Problema

La **planificación continua** está orientada a resolver problemas en ambientes dinámicos, por lo tanto, se encarga de atacar problemas del mundo real. Este método de planificación es denominado “clásico”, es decir, que no tiene separadas las etapas de planificación y ejecución sino, que se encuentra planificando de manera continua en el tiempo. Hasta el momento de la publicación de este framework, la planificación continua solamente había sido presentada en forma conceptual en [47].

La temática que aborda el trabajo de Moya es el desarrollo de una arquitectura para agentes que soporte tanto *control reactivo* como *deliberativo* y, en consecuencia, permitir que el agente pueda actuar de manera competente y efectiva en un ambiente real. En [23], Hanks y Firby, sugieren tratar de alcanzar un sutil equilibrio entre estas dos estrategias: *deliberación* y *reacción*. La primera estrategia implica tomar todas las decisiones factibles en forma tan anticipada en el tiempo como sea posible. Mientras que la segunda, consiste en demorar las decisiones que se

---

<sup>1</sup> General Public License.

tomen actuando, únicamente, en el último momento posible.

Un agente que pueda pensar a futuro será capaz de considerar más opciones y, por lo tanto, estará más informado para decidir qué acción tomar. Sin embargo, debido a que la información sobre el futuro puede ser poco confiable y, en muchas situaciones del mundo real, difícil o incluso imposible de obtener, puede ser razonable también actuar a último momento.

En resumen, esta implementación aborda los problemas planteados previamente dotando a un agente inteligente con capacidades deliberativas y reactivas, brindando así, la posibilidad de elegir cuál sería la mejor forma de actuar frente a un problema determinado.

### 3.2. Características Principales del Framework

Tal como comentamos arriba, el diseño del agente tiene dos modos de operación: reactivo y deliberativo. Por simplificación, el trabajo profundiza sobre el modo deliberativo y la posibilidad de escoger en qué modo actuar.

La implementación provee un subsistema de control que interactúa directamente con los sensores y efectores de un agente. El sistema recibe las percepciones y devuelve acciones que pueden ser provistas tanto por el modo reactivo como por el modo deliberativo. Desde el punto de vista modular, la organización sigue los principios del modelo BDI (Belief, Desire, Intention) [46].

Una de las características principales, en la implementación, es el uso de hilos<sup>2</sup>. El framework mantiene dos hilos principales de ejecución, uno para el controlador, que toma nuevas percepciones y actualiza el estado del agente, y otro para el planificador, que intentará encontrar el plan que satisfaga las metas y percepciones actuales. Estos hilos son definidos en Ciao Prolog usando *data predicates*. Los *data predicates* están compuestos exclusivamente por hechos que pueden ser observados, agregados o eliminados dinámicamente.

La arquitectura modular del framework se muestra a continuación.

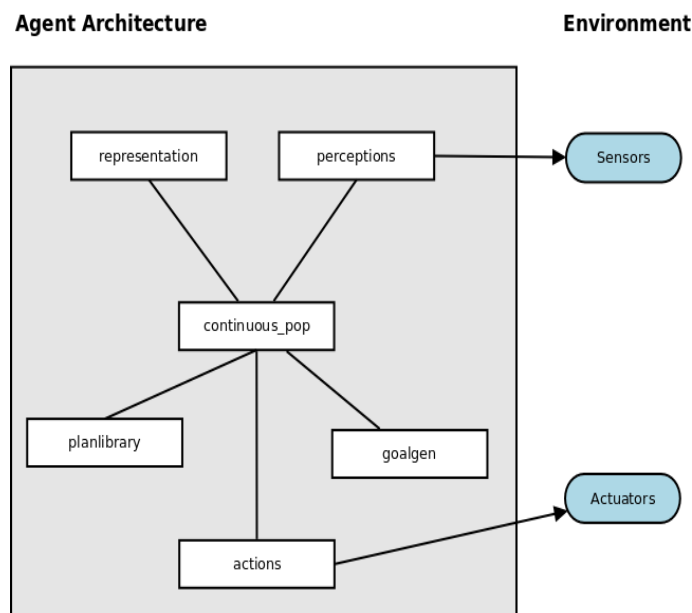


Figura 3.1: Arquitectura Modular del Framework de Planificación Continua

<sup>2</sup>En Inglés, *threads*.

### 3.2.1. Algoritmo de Planificación Continua

El algoritmo de planificación continua, presentado en [36], es un bucle que obtiene percepciones, actualiza el estado actual e intenta generar un plan.

Una extensión importante de la implementación es la capacidad de elegir un plan predefinido de una librería de planes. Por lo tanto, cuando el planificador es invocado, recibe un plan inicial de la librería de planes. En caso que no se encuentre un plan acorde, se comienza con uno vacío.

El algoritmo tiene tres estados bien definidos:

- En primer lugar, reformula las metas luego de procesar los deseos del agente. Entonces, se remueven enlaces no soportados y así se evitan acciones con precondiciones falsas debido a cambios en los deseos del agente. A continuación, el algoritmo elimina acciones redundantes que ya no son necesarias en el plan. Esta eliminación de acciones conlleva un reemplazo de enlaces causales<sup>3</sup>.
- El siguiente paso es una conducta heredada del Planificador de Orden Parcial (POP) [30]. Tal conducta involucra resolver precondiciones abiertas. Para esto, se selecciona una precondición e intenta resolverse agregando una nueva acción o utilizando una ya existente.
- El último paso del algoritmo es verificar si se ha alcanzado el conjunto actual de metas, es decir, no existen precondiciones abiertas y todos los enlaces causales van del estado *Start* al estado *Finish*<sup>4</sup>.

## 3.3. Caso de estudio

Como mencionamos previamente, algunas experiencias con esta implementación fueron realizadas bajo el dominio del Fútbol de Robots. A continuación, se detallan las razones por las que este dominio fue usado para la evaluación del framework.

- Es un dominio muy desafiante y ampliamente estudiado por el Departamento de Teoría de la Computación [11].
- Ofrece la posibilidad de aplicar un amplio rango de tecnologías, como por ejemplo, video, comunicaciones y robótica, entre otras.
- Existen en la actualidad, distintas competencias de robots con especificaciones y reglamentos diferentes. Entre ellas, podemos destacar a Robocup<sup>5</sup>, FIRA<sup>6</sup> y CAFR<sup>7</sup>.
- Presenta dos características importantes: es un ambiente continuo y no determinista.

La utilización de este dominio es, entonces, propicio para la aplicación del framework. El caso de estudio también utiliza al software Rakiduum.

Con el objetivo de adaptar el framework y Rakiduum, es necesario identificar y definir los deseos del agente, identificar acciones y metas y especificarlas en un lenguaje de representación de problemas de planificación.

<sup>3</sup>Un enlace causal es de la forma  $A \rightarrow^p B$  donde  $A$  y  $B$  son acciones y  $p$  es una proposición que es precondición de la acción  $B$ . La ejecución de  $A$  hace verdadero a  $p$  para  $B$  [47].

<sup>4</sup>Las acciones *Start* y *Finish* pertenecen al plan inicial de la formulación de problemas de planificación para el planificador POP. Tales acciones tienen la siguiente restricción de orden  $Start < Finish$ . *Start* no tiene precondiciones y sus efectos son todos los literales en el estado inicial del problema. *Finish* no tiene efectos y sus precondiciones son los literales de la meta del problema [47].

<sup>5</sup>Robocup. <http://www.robocup.org/>. Disponible en Septiembre de 2012.

<sup>6</sup>Federation of International Robot-soccer Association. <http://www.fira.net/>. Disponible en Septiembre de 2012.

<sup>7</sup>Campeonato Argentino de Fútbol de Robots.

En primer lugar, y de acuerdo a la arquitectura descrita en 3.1, el módulo **goalgen** generará un conjunto nuevo de metas, consistentes entre sí, a partir de las creencias actuales del agente. Estas creencias incluyen las percepciones (**perceptions**) y la representación de acciones (**representation**). El módulo **goalgen** también contiene la estrategia del equipo y la de cada jugador según su rol. El módulo **actions** implementa cada una de las acciones y las traduce a velocidades de los motores del agente mediante las primitivas de navegación provistas por Rakiduum. Por último, el lenguaje para la representación de acciones y percepciones es STRIPS.

Luego de esta breve descripción del framework presentado por Moya, vamos a introducir, en la sección siguiente, algunas cuestiones relacionadas al rol de nuestro traductor en esta arquitectura.

### 3.4. Integrando al Framework el Traductor PDDL desarrollado

Como describimos en la sección previa, el formalismo que el framework usa para especificar percepciones y acciones es STRIPS. Por otro lado, también vimos, en el capítulo 2, que STRIPS está limitado a la representación simple de acciones como una conjunción de literales. Esta última restricción torna dificultoso el modelado de acciones complejas, por lo tanto, surge la necesidad de tener un lenguaje de representación más general, con un nivel de abstracción mayor al de STRIPS y orientado al modelado de dominios de aplicaciones reales. Entonces, PDDL, aparece como un potencial lenguaje de definición de dominios para integrar al framework implementado por Moya. También, es importante notar que PDDL fue adoptado como un estándar con el objetivo de comparar la performance de planificadores y compartir diferentes problemas para un mismo dominio. De esta manera, el framework podría ser cotejado con otras soluciones a un mismo problema.

En base a lo expuesto, podemos definir una nueva arquitectura en la que integraremos nuestro traductor al Framework de Planificación Continua. Luego, daremos un ejemplo de cómo las acciones de un agente pueden ser representadas en PDDL.

Previamente, vamos a presentar un gráfico que mapea el subsistema de creencias, integrado por los módulos de percepciones y representación, sobre el que nos proponemos trabajar.

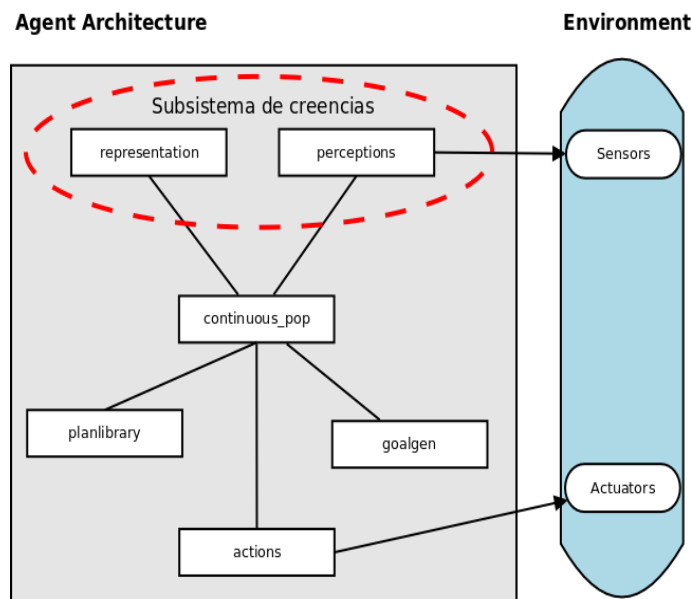


Figura 3.2: El subsistema de creencias en la Arquitectura Modular del Framework.

Vemos que existen dos módulos que dependen de una especificación STRIPS, el módulo **representation**, que contiene las acciones que el agente puede ejecutar y el módulo **perceptions**, que recibe percepciones desde los sensores y las traduce a una representación STRIPS. Por lo tan-



to, debido a que ambos módulos constituyen el subsistema de creencias del agente, tendremos un subsistema puramente especificado en PDDL.

En la figura 3.3, podemos ver que los módulos **perceptions** y **representation** dependen del traductor PDDL. Las percepciones recibidas desde los sensores y la representación de las acciones del agente son, entonces, traducidas a STRIPS. La línea punteada en el gráfico indica que ambos módulos del subsistema de creencias dependen de nuestro traductor.

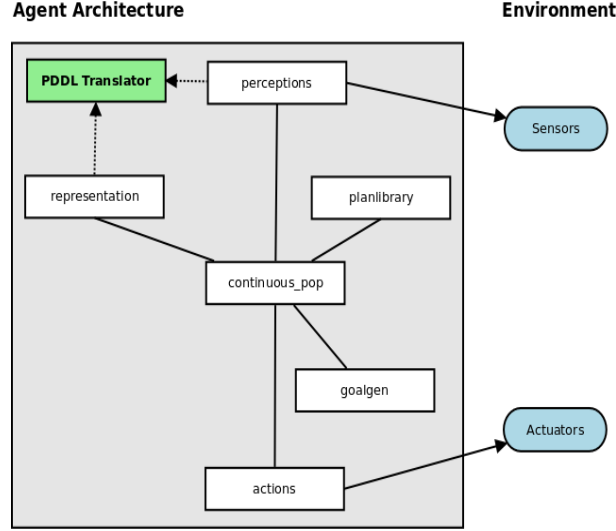


Figura 3.3: Arquitectura del Framework y Traductor PDDL

A continuación, mostramos un ejemplo de cómo especificamos acciones del agente en PDDL. Primero, mostramos la especificación de la acción en STRIPS y luego, la equivalente en PDDL. Por simplicidad, empleamos el mismo ejemplo utilizado por Moya en [36].

**Ejemplo 3.4.1.** *En la acción  $\text{move}(\text{Ag}, \text{Pos1}, \text{Pos2})$ , el agente  $\text{Ag}$ , se desplaza de la posición  $\text{pos1}$  a la posición  $\text{pos2}$ .*

```
preconditions(move(Ag,Pos1,Pos2), [player(Ag),waiting_at(Ag,Pos1),
valid_move(Pos1,Pos2)]).
achieves(move(Ag,Pos1,Pos2),waiting_at(Ag,Pos2)).
deletes(move(Ag,Pos1,Pos2),waiting_at(Ag,Pos1)).
```

**Ejemplo 3.4.2.** *La acción equivalente en PDDL es la siguiente:*

```
(:action move
  (:parameters (?Ag ?Pos1 ?Pos2)
  (:precondition (and (player ?Ag) (waiting_at ?Ag ?Pos1)
    (valid_move ?Pos1 ?Pos2)))
  (:effect (and (waiting_at ?Ag ?Pos2)
    (not (waiting_at ?Ag ?Pos1))))
)
```

El módulo **perceptions** recibe el estado actual que incluye, para el caso del Fútbol de Robots, las coordenadas X e Y de los jugadores y de la pelota. El siguiente ejemplo especifica el estado actual usando PDDL.

**Ejemplo 3.4.3.** *Suponemos que el dominio de Fútbol de Robots es llamado `soccer_domain` y los jugadores son nombrados como `Ag` seguido de un número que lo identifica unívocamente en el problema. La meta es que el jugador `Ag1` se desplace hacia la pelota `ball`.*

```
(define (problem soccer_problem)
  (:domain soccer_domain)
  (:objects Ag1 Ag2 Ag3 ball)
  (:goal (carrying Ag1 ball))
  (:init (at Ag1 cell1)
         (at Ag2 cell2)
         (at Ag3 cell3)
         (at ball cell4)
         (clear cell5)
         (clear cell6)
         (clear cell7))
)
```

Por lo tanto, proyectamos un planificador continuo cuyo lenguaje de representación es PDDL.

Hasta aquí, hemos estudiado el lenguaje PDDL, la implementación del Framework de Planificación Continua desarrollado por Moya y vimos como el traductor PDDL propuesto puede ser integrado al Framework. En el capítulo siguiente, analizaremos alternativas existentes y, a partir del capítulo 6, comenzaremos a delinear nuestra implementación.

## Capítulo 4

# Implementaciones Existentes

En el trascurso de este trabajo hemos investigado y probado implementaciones existentes que tienen la capacidad de interpretar PDDL como lenguaje de entrada y generar una especificación equivalente en un lenguaje destino o, simplemente, procesar sus estructuras. Además de considerar los lenguajes fuente y destino, al momento de abordar estas implementaciones, también consideramos algunas otras características importantes como lenguaje de implementación, licencias, subconjunto de PDDL que es capaz de interpretar y nivel de madurez de la implementación. Por último, indicamos las principales diferencias con nuestra propuesta.

En este capítulo se detallan las implementaciones investigadas indicando los puntos resaltados en el parrafo anterior. En la sección 4.1 se describe la alternativa planteada sobre SWI-Prolog [58] y, en la sección 4.2, se estudia la grámatica especificada en ANTLR [42]. Por último, en la sección 4.3, se analiza una librería JAVA adaptada para Graphplan [4].

### 4.1. Analizador en SWI-Prolog

SWI-Prolog es una implementación del lenguaje de programación Prolog basada en un subconjunto de la WAM (*Warren Abstract Machine*) [56]. Tiene un importante conjunto de características entre las cuales se destacan las librerías para programación lógica restringida, multihilos, testing unitario, interfaces y herramientas para la Web Semántica como RDF<sup>1</sup> [55] y RDFS<sup>2</sup> [54]. SWI-Prolog está bajo continuo desarrollo desde 1987 y su principal autor es Jan Wielemaker<sup>3</sup>. El nombre SWI deriva del grupo *Sociaal-Wetenschappelijke Informatica* ("*Social Science Informatics*") perteneciente a la Universidad de Amsterdam.

SWI-Prolog está distribuido bajo las licencias LGPL<sup>4</sup> [18], para librerías del kernel y aquellas no desarrolladas en código Prolog y, bajo licencia GPL<sup>5</sup> [19], para librerías desarrolladas en código Prolog.

El analizador PDDL en SWI-Prolog [50] fue desarrollado por Róbert Sasák<sup>6</sup> utilizando la librería Prolog DCG<sup>7</sup>. La implementación está incompleta ya que abarca sólo un subconjunto de PDDL 3.0 y fue probada por Sasák sobre los planificadores *Forward* and *Backward Search* [48].

El analizador incluye tres módulos principales. El primero, llamado `readFile.pl`, implementa un "tokenizador". Dicho módulo toma como entrada un archivo de texto con la especificación PDDL y retorna un conjunto de tokens. El módulo `parseProblem.pl` recibe como entrada la definición de un problema PDDL y retorna términos Prolog. Por último, `parseDomain.pl`, recibe

---

<sup>1</sup> *Resources Description Framework*.

<sup>2</sup> *Resources Description Framework Schema*.

<sup>3</sup> Jan Wielemaker. <http://staff.science.uva.nl/~wielemak/>. Disponible en Septiembre de 2012.

<sup>4</sup> *Lesser GNU General Public License*.

<sup>5</sup> *GNU General Public License*.

<sup>6</sup> Róbert Sasák. <http://www.sasak.sk/>. Disponible en Septiembre de 2012.

<sup>7</sup> *Definite Clause Grammar*. En el capítulo 6 introduciremos DCG.

un dominio PDDL y retorna términos Prolog equivalentes. Es importante destacar que la salida de este analizador es una representación tipo Prolog y no STRIPS.

A continuación, mostramos un ejemplo de una conversión simple ejecutada por el parser y luego la salida del analizador para una instancia del “Mundo de Bloques”.

**Ejemplo 4.1.1.** *El predicado PDDL ( $on\ ?x\ ?y$ ) es convertido al siguiente predicado Prolog  $on(?x, ?y)$ .*

**Ejemplo 4.1.2.** *La salida equivalente a la definición del “Mundo de Bloques” es la siguiente:*

```
?-parseDomain('blocks_world.pddl', 0).
    0 = domain(blocks,                                %name
               [strips, equality],                     %requirements
               [table],                                %constants
               [on(?x, ?y),                             %predicates
                clear(?x),
                holding(?x),
                ontable(?x),
                armempty],
               [action('pickup',                        %actions
                       [?x],
                       [clear(?x), armempty],
                       [holding(?x)],
                       [on(?x ?y), armempty]]],
               [action('stack',
                       [?x, ?y],
                       [holding(?x), clear(?y)],
                       [on(?x ?y), clear (?x), armempty],
                       [clear(?y), holding(?x)]]
               )

?-parseProblem('problem.pddl', 0).
    0 = problem('blocks-4-0',                          %name
               blocks,%domain name
               [(a,b)],                                  %objects
               [clear(a),                                %init
                clear(b),
                ontable(b),
                ontable(c),
                armempty,
                on(a,c)],
               [on(a,b)],                                %goal
               )
```

Los términos Prolog retornados por el analizador son predicados con una estructura general con un argumento por cada sección PDDL en el dominio y problema de entrada.

Por último, como mencionamos previamente, el analizador PDDL aquí presentado sólo abarca un subconjunto de la especificación de PDDL 3.0. Los requerimientos no incluidos son: restricciones, precondiciones negativas y disyuntivas, acciones con restricciones de duración, predicados derivados, preferencias, precondiciones universales, precondiciones existenciales y efectos condicionales.

En base a lo expuesto, concluimos que las diferencias con nuestra propuesta radican en el tipo de salida que presenta el traductor implementado por Sasák y en el ambiente de programación en el que está integrado. Como hemos mencionado en esta sección, esta implementación está desarrollada en un ambiente Prolog diferente al de Ciao. Además, su lenguaje destino no es STRIPS (no distingue entre listas de precondiciones, agregados y borrados) y no manipula precondiciones disyuntivas, universales ni efectos condicionales.

## 4.2. Gramática ANTLR para PDDL

ANTLR<sup>8</sup> es una herramienta que provee un framework para la construcción de analizadores. Puede generar analizadores léxicos, sintácticos y semánticos en varios lenguajes *target*, como JAVA, C, C++ y Python, a partir de una especificación en su propio lenguaje. Básicamente, el lenguaje de ANTLR está formado por una serie de reglas EBNF<sup>9</sup> [24] y un conjunto de reglas auxiliares. ANTLR toma una gramática como entrada y genera como salida el código generado en un lenguaje destino, como algunos de los mencionados arriba. El lenguaje *target* deseado es especificado en la misma gramática. La herramienta está disponible desde 1990, fue desarrollada por Terence Parr<sup>10</sup> y es distribuida bajo licencia de software BSD<sup>11</sup> [43].

ANTLR genera analizadores  $LL(k)$  que soportan predicados sintácticos y semánticos permitiendo especificar muchas gramáticas libres y sensibles al contexto [12]. Se denomina  $LL(k)$  a la clase de gramáticas para las cuales es posible construir analizadores sintácticos descendentes predictivos. Estos analizadores realizan un recorrido de izquierda a derecha, con  $k$  símbolos de anticipación<sup>12</sup> y obtienen una derivación a izquierda de las cadenas de entrada.

Además, ANTLR provee soporte para la construcción de árboles gramaticales y para la recuperación y reporte de errores.

Los tres tipos básicos de herramientas procesadoras de lenguaje que ANTLR genera son:

1. **Analizadores Léxicos (*lexers*)**. Un analizador léxico lee una cadena de caracteres, y a partir de patrones específicos, detecta y retorna una cadena de *tokens*. También puede procesar espacios en blanco y comentarios,
2. **Analizadores Sintácticos (*parsers*)**. Un analizador sintáctico lee una cadena de tokens e identifica frases usando reglas gramaticales. Generalmente, también ejecuta algunas acciones semánticas para cada frase identificada, y
3. ***Tree-parsers***. Es un analizador que aplica una estructura gramatical para guiar la traducción de una entrada. Son útiles durante la traducción agregando atributos, reglas y acciones a la estructura del árbol. Estos analizadores generan como salida un Árbol Sintáctico Abstracto (AST)<sup>13</sup>. Un AST es una forma de código intermedio que representa una estructura sintáctica y jerárquica del programa fuente.

ANTLR permite, entonces, definir las reglas que el *lexer* debe usar para *tokenizar* caracteres y las reglas que un *parser* debe usar para interpretar a los *tokens*. A continuación, explicaremos cómo una gramática ANTLR es definida y ejemplificaremos usando la gramática ANTLR PDDL.

Los archivos ANTLR tienen una extensión “\*.g”. Pueden contener la definición gramatical de uno o varios analizadores y, por cada analizador descripto, se generará una clase en el lenguaje destino elegido.

<sup>8</sup> ANother Tool for Language Recognition.

<sup>9</sup> Extended Backus Naur Form.

<sup>10</sup> Terence Parr. <http://www.cs.usfca.edu/~parrr/>. Disponible en Septiembre de 2012.

<sup>11</sup> Berkeley Software Distribution.

<sup>12</sup> En Inglés, *lookahead*.

<sup>13</sup> En Inglés, *Abstract Syntax Tree*.

ANTLR PDDL fue desarrollada por Zeyn Saigol<sup>14</sup> en el año 1998. La gramática toma un subconjunto de PDDL 3.0 y genera un analizador léxico y un analizador sintáctico en código nativo JAVA. La salida de la gramática, luego de cargarla en el entorno de desarrollo ANTLRWorks<sup>15</sup> [5], está conformada por tres archivos: un archivo de tokens, y dos “\*.java”, una clase *Lexer.java* y una clase *Parser.java*.

A continuación, analizaremos la estructura de la gramática ANTLR desarrollada para el lenguaje PDDL. Empleamos algunos ejemplos extraídos de la definición original que puede ser encontrada en [49]. ANTLR PDDL tiene la siguiente estructura:

- **Cabecera.** La cabecera, en una gramática ANTLR, contiene código fuente que debe ser ubicado previo a las clases de los analizadores en el código generado. Esta sección es opcional.

El siguiente ejemplo muestra la definición de una cabecera cuando el lenguaje *target* es JAVA.

**Ejemplo 4.2.1.** *La definición de las cabeceras indican que el paquete correspondiente debe declararse previo al resto del código.*

```
@parser::header {package uk.ac.bham.cs.zas.pddl antlr;}
@lexer::header {package uk.ac.bham.cs.zas.pddl antlr;}
```

- **Opciones.** Es una sección opcional, al igual que el *header*, aunque es importante en la definición de una gramática. Se pueden especificar parámetros de ANTLR como por ejemplo, el lenguaje *target* elegido, entre otros.

**Ejemplo 4.2.2.** *Las siguientes opciones son generales de la gramática:*

```
options {
    output=AST;
    backtrack=true;
}
```

donde:

- *output=AST:* especifica el tipo de salida del parser. Los valores válidos son *AST* y *template*. Un *AST* es una forma de representación intermedia que consiste de nodos cuyas raíces de los subárboles son sólo operadores. El valor *template* permite el uso de la librería *StringTemplate*<sup>16</sup> para la generación de código a partir de determinados templates. Esta configuración separa la estructura de datos de cómo la salida es presentada.
- *backtrack=true:* activa la recuperación de errores reglas no son exitosas. En este caso, ningún error es reportado durante el análisis.

Ningún lenguaje *target* es especificado en esta sección pero, por defecto, ANTLR asume que el código destino es JAVA.

<sup>14</sup>Zeyn Saigol. <http://www.zeynsaigol.com/>. Disponible en Septiembre de 2012.

<sup>15</sup>The ANTLR GUI Development Environment.

<sup>16</sup>La documentación oficial de esta librería puede consultarse en <http://www.antlr.org/wiki/display/ST/Introduction>. Disponible en Septiembre de 2012.

- **tokens:** En esta sección pueden definirse dos tipos de *tokens*. Los primeros, llamados “imaginarios”, no se corresponden con un símbolo de entrada real y pueden ser usados como *labels* para agrupar *tokens* de la entrada real. Estos *tokens* puede ser referenciados desde las reglas gramaticales. Además, también pueden especificarse *tokens* llamados “literales” que suelen usarse para asignar algún *label* a *tokens* de la entrada.

**Ejemplo 4.2.3.** *El siguiente es un subconjunto de tokens definidos en la gramática ANTLR.*

```
tokens {
    DOMAIN;
    DOMAIN_NAME;
    REQUIREMENTS;
    TYPES;
    CONSTANTS;
    PREDICATES;
    ACTION;
    PROBLEM;
    PROBLEM_NAME;
    PROBLEM_DOMAIN;
    OBJECTS;
    INIT;
    PRECONDITION;
    EFFECT;
    AND_GD;
    OR_GD;
    NOT_GD;
    FORALL_GD;
    AND_EFFECT;
    FORALL_EFFECT;
    WHEN_EFFECT;
    GOAL;
}
```

- **members:** en esta sección se definen métodos y variables que luego formarán parte de la clase del analizador.

**Ejemplo 4.2.4.** *La gramática define los siguientes métodos para la clase parser:*

```
@parser::members {
    private boolean wasError = false;
    public void reportError(RecognitionException e) {
        wasError = true;
        super.reportError(e);
    }
    public boolean invalidGrammar() {
        return wasError;
    }
}
```

- **Reglas:** la última sección de la gramática es la definición de reglas gramaticales. ANTLR PDDL incluye reglas para el *lexer*, que se utilizarán para la identificación de tokens y reglas

para el *parser*, que son patrones con el objetivo de identificar frases del lenguaje usando los tokens generados por el *lexer*. Las reglas están en notación EBNF.

**Ejemplo 4.2.5.** *Las siguientes son algunas de las reglas definidas para el análisis léxico. Las reglas están acotadas al subconjunto de PDDL definido para el presente trabajo.*

```

REQUIRE_KEY
: ':strips'
| ':disjunctive-preconditions'
| ':equality'
| ':universal-preconditions'
| ':quantified-preconditions'
| ':conditional-effects'
;

NAME:    LETTER ANY_CHAR* ;

fragment LETTER: 'a'..'z' | 'A'..'Z';
fragment ANY_CHAR: LETTER | '0'..'9' | '-' | '_';

VARIABLE : '?' LETTER ANY_CHAR* ;

NUMBER : DIGIT+ ('.' DIGIT+)? ;

fragment DIGIT: '0'..'9';

LINE_COMMENT
: ';' ~('\'\'|\'\'r')* \'\'r'? \'\'n' { $channel = HIDDEN; }
;

WHITESPACE
: (
  | '\t'
  | '\r'
  | '\n'
)+
{ $channel = HIDDEN; }
;

```

**Ejemplo 4.2.6.** *Las siguientes son algunas de las reglas definidas para el análisis sintáctico. Las primeras, corresponden a dominios PDDL (*domain* y *actionDef*) y las últimas, a problemas PDDL (*problem*).*

```

pddlDoc : domain | problem;}

domain
: '(' 'define' domainName
  requireDef?
  typesDef?
  constantsDef?

```



```

    predicatesDef?
    functionsDef?
    constraints?
    structureDef*
    '),
    -> ^(DOMAIN domainName requireDef? typesDef?
        constantsDef? predicatesDef? functionsDef?
        constraints? structureDef*)
;

actionDef
: '(' ':action' actionSymbol
  ':parameters' '(' typedVariableList ')'
  actionDefBody ')'
-> ^(ACTION actionSymbol typedVariableList actionDefBody)
;

problem
: '(' 'define' problemDecl
  problemDomain
  requireDef?
  objectDecl?
  init
  goal
  probConstraints?
  metricSpec?
  '),
-> ^(PROBLEM problemDecl problemDomain requireDef? objectDecl?
  init goal probConstraints? metricSpec?)
;

```

De manera similar a la sección anterior, las diferencias con nuestra propuesta radican en el tipo de solución que ofrece la gramática ANTLR y el lenguaje en el que está implementada. Si bien, es una alternativa flexible a los cambios en la gramática PDDL de entrada, los analizadores, generados a partir de la gramática ANTLR PDDL, no realizan ninguna traducción de PDDL a STRIPS.

### 4.3. Librería PDDL4J

PDDL4J es una librería de código abierto distribuida bajo licencia CECILL [10] y fue desarrollada por Damien Pellier<sup>17</sup>.

El propósito de PDDL4J es facilitar la implementación JAVA de planificadores basados en PDDL. La librería contiene un analizador de la especificación PDDL 3.0 con todas las clases necesarias para la manipulación de sus principales características. Acepta un importante conjunto de requerimientos PDDL entre los que se encuentran los requerimientos presentados para nuestro trabajo y se agregan tipos, precondiciones negativas y existenciales, fluents, predicados derivados, acciones con tiempo, preferencias y restricciones.

<sup>17</sup>Damien Pellier. <http://www.math-info.univ-paris5.fr/~pellier/home>. Disponible en Septiembre de 2012.

La propuesta de Pellier incluye una implementación del planificador Graphplan usando PDDL4J. Lo novedoso de esta implementación es que retorna, junto con el plan final, algunas estadísticas sobre la ejecución del algoritmo con el objetivo de comparar esta solución con otras implementaciones.

Graphplan es un planificador de propósito general para dominios especificados en STRIPS. Dado un problema, Graphplan construye, explícitamente, una estructura llamada *Grafo de Planificación* que es explorado para obtener un plan que resuelva un problema, si es que éste existe. Graphplan fue creado por Avrim Blum<sup>18</sup>, Merrick Furst<sup>19</sup> y John Langford<sup>20</sup>.

A continuación, mostramos la salida de Graphplan luego de ejecutarlo sobre el dominio y problema PDDL de una instancia del “Mundo de Bloques”.

**Ejemplo 4.3.1.** *Dada la meta  $on(a\ b)$  y las acciones  $stack$  y  $pickup$ , ejecutamos Graphplan pasando, como argumentos del planificador, los archivos “\*.pddl” correspondientes.*

```
java -server -jar graphplan.jar domain.pddl problem.pddl
```

*Luego, el resultado de la planificación es el siguiente:*

```
Parsing domain "bkw" done successfully ...
Parsing problem "pb1" done successfully ...

Preprocessing ...

time:    0,          5 facts,          0 and exclusive pairs (0,0003s)
          7 ops,          7 exclusive pairs
time:    1,          7 facts,          7 and exclusive pairs (0,0006s)
          11 ops,         35 exclusive pairs

goals first reachable in 2 times steps

found plan as follows:

time step    0: PICKUP a
time step    1: STACK a b

number of actions tried :          2
number of noops tried   :          1

Time spent :    0,01 seconds preprocessing
               0,00 seconds build graph
               0,00 seconds calculating exclusions
               0,00 seconds searching graph
               0,01 seconds total time
```

*La salida retorna el plan resultante para alcanzar la meta e información relacionada al tiempo de ejecución del algoritmo. Entre la información mostrada, podemos destacar que el problema y*

<sup>18</sup>Avrim Blum. <http://www.cs.cmu.edu/~avrim/home.html>. Disponible en Septiembre de 2012.

<sup>19</sup>Merrick Furst. <http://www.scs.gatech.edu/people/merrick-furst>. Disponible en Septiembre de 2012.

<sup>20</sup>John Langford. <http://hunch.net/~jl/>. Disponible en Septiembre de 2012.

*el dominio en PDDL fue analizado exitosamente, la meta fue alcanzada con sólo dos acciones `pickup a` y `stack a b`, y el tiempo total para alcanzar la solución fue de 0.01 segundos.*

Teniendo en cuenta lo analizado en esta sección, concluimos que la diferencia con nuestra propuesta radica en que la solución, desarrollada por Pellier, es una librería JAVA y su propósito es facilitar la implementación de planificadores en el mismo lenguaje. Este no es el caso del Planificador Continuo ya que está desarrollado, íntegramente, en Ciao Prolog. Es esperable una mejor integración si el traductor, propuesto en este trabajo, es implementado en el mismo ambiente del Planificador Continuo.

En resumen, en este capítulo hemos presentado y analizado tres implementaciones existentes que fueron estudiadas en el transcurso de este trabajo. A partir del próximo capítulo, vamos a iniciar el análisis de nuestro traductor PDDL y, para esto, comenzaremos definiendo el marco teórico subyacente.



## Capítulo 5

# Traducción de requerimientos PDDL

En el capítulo 2 analizamos el lenguaje STRIPS. Este formalismo describe acciones en términos de precondiciones y efectos. Las precondiciones de las acciones son conjunciones de literales positivos, al igual que los estados inicial y final. Los efectos de las acciones son expresados como una conjunción de literales. Esta definición de STRIPS, más el predicado de igualdad y su negación, conforman el lenguaje de representación soportado por el Planificador Continuo presentado en el capítulo 3. Luego, estudiamos el lenguaje PDDL e identificamos los siguientes requerimientos que están siendo considerados para el desarrollo del presente trabajo: igualdad, efectos condicionales, precondiciones disyuntivas y precondiciones universales.

En este capítulo vamos a identificar y definir variantes para STRIPS. Tales variantes surgen adicionando los requerimientos anteriores a la especificación de dicho lenguaje. Además, estableceremos una notación equivalente en PDDL para estas variantes y analizaremos la teoría subyacente. Por último, propondremos formas de procesar cada variante para una posterior implementación en el traductor propuesto.

La organización del presente capítulo es la siguiente. En la sección 5.1 se define un marco teórico incluyendo resultados preliminares, las nociones de **esquemas de compilación** y **compilabilidad** y algunos supuestos de simplificación. En las secciones subsiguientes se discute, de manera teórica, la traducción de cada una de las variantes STRIPS identificadas. En la sección 5.2 se muestra la traducción *trivial* para STRIPS estándar. Posteriormente, en la sección 5.3, se analiza STRIPS con literales negativos y el predicado de igualdad. A este formalismo lo denotaremos  $PDDL_L$ . En la sección 5.4 se analiza STRIPS con efectos condicionales,  $STRIPS_C$  y su equivalente  $PDDL_C$ . La sección 5.5 presenta STRIPS con precondiciones disyuntivas,  $STRIPS_D$  y  $PDDL_D$ . En la sección 5.6 se introduce uno de los resultados más importantes de nuestra investigación. Este resultado formaliza y demuestra una nueva variante de STRIPS que incluye precondiciones cuantificadas universalmente. Tal variante es llamada  $STRIPS_u$  con su equivalente  $PDDL_u$ . Por último, todos los ejemplos en este capítulo concluyen con la representación STRIPS para cada dominio traducido. Tal representación es similar a la definida para el Planificador Continuo.

### 5.1. Marco Teórico

Existe un *trade-off*<sup>1</sup> entre la expresividad y la “gestión”<sup>2</sup> de un lenguaje formal de definición de dominios de planificación. Un lenguaje práctico debería tener un nivel lo más alto posible para poder representar problemas de una manera natural, exacta y simple. Sin embargo, mientras más se incrementa la complejidad de lenguaje en cuestión, más se incrementa la dificultad para resolver el problema por parte de los planificadores. En [20], Gazen y Knoblock, definen dos enfoques posibles que permiten considerar un lenguaje más expresivo de representación de dominios. El

---

<sup>1</sup>Un balance aceptable entre dos cosas opuestas con el fin de lograr un objetivo.

<sup>2</sup>En Inglés, *manageability*.

primero, enuncia que para soportar un lenguaje más expresivo se debe extender el algoritmo de planificación. Mientras que el segundo enfoque, implica desarrollar un preprocesador que traduzca un dominio representado en un lenguaje más expresivo a uno más simple. Si bien, esta última propuesta no podría manejar constructores más expresivos de manera tan eficiente, es más simple desde el punto de vista conceptual. Consideramos esta segunda propuesta para desarrollar el presente trabajo.

Esta sección presenta resultados y definiciones, introducidas por Nebel en [40] y [41], y adaptadas a las necesidades de nuestra implementación.

### 5.1.1. Esquemas de Compilación

En primer lugar, introducimos la definición de instancia de planificación para luego continuar el estudio sobre los esquemas de compilación.

**Definición 5.1.1.** Sea  $\Sigma$  un conjunto finito de átomos proposicionales,  $O$  un conjunto finito de operadores y  $\hat{\Sigma} = \top \cup \perp \cup \text{literales}(\Sigma)$  donde,  $\top$  denota verdadero,  $\perp$  denota falso y  $\text{literales}(\Sigma)$  es el conjunto de literales definidos sobre  $\Sigma$ .

Una **instancia de planificación** es una tupla  $\Pi = \langle \Xi, I, G \rangle$  donde:

- $\Xi = \langle \Sigma, O \rangle$  es una estructura de dominio que consiste de un conjunto finito de átomos proposicionales y un conjunto finito de operadores,
- $I \subseteq \Sigma$  es el estado inicial, y
- $G \subseteq \hat{\Sigma}$  es la especificación de la meta.

La estructura  $\Xi$  es el dominio de planificación compuesto por acciones (también llamadas operadores en la bibliografía) mientras que los conjuntos  $I$  y  $G$  conforman la especificación del problema de planificación.

A continuación, damos una noción intuitiva del concepto de esquemas de compilación y, posteriormente, presentamos este concepto de manera formal, adaptando la definición propuesta por Nebel.

Un formalismo de planificación  $X$  es tan expresivo como otro formalismo  $Y$  si, los dominios de planificación y los planes formulados en  $Y$ , son expresables en  $X$  de manera concisa. Básicamente, un esquema de compilación es una solución que preserva el mapeo entre las estructuras de dominio en el formalismo  $X$  y en el formalismo  $Y$ . La siguiente definición expresa este concepto.

**Definición 5.1.2.** Un **esquema de compilación**<sup>3</sup> de  $X$  a  $Y$  es una tupla de funciones  $f = \langle f_\xi, f_i, f_g \rangle$  que inducen una función  $F$  de instancias de  $X$ ,  $\Pi = \langle \Xi, I, G \rangle$ , a instancias de  $Y$ ,  $F(\Pi)$ , como sigue:

$$F(\Pi) = \langle f_\xi(\Xi), I \cup f_i(\Xi), G \cup f_g(\Xi) \rangle$$

y satisfacen las siguientes condiciones:

- existe un plan para  $\Pi$ , si y sólo si, existe un plan para  $F(\Pi)$ , y
- el tamaño de los resultados de  $f_\xi, f_i$  y  $f_g$  es polinomial en el tamaño de los argumentos.

Si bien, esta definición establece condiciones referidas a la complejidad computacional de las funciones  $f_\xi, f_i$  y  $f_g$ , esto no es un factor limitante en nuestra investigación. Sin embargo, en el caso general, procuramos definir esquemas de compilación eficientes en términos de complejidad computacional.

En este sentido, establecemos los siguientes supuestos relacionados a este tipo de mapeos:

---

<sup>3</sup>En Inglés, *compilation scheme*.

1. No haremos ninguna restricción sobre los recursos computacionales, por lo tanto, un análisis exhaustivo de la complejidad computacional queda fuera del alcance del presente trabajo.
2. Las estructuras de dominio son traducidas de manera independiente respecto de la descripción del estado inicial y la meta del problema de planificación.
3. No existe ninguna restricción sobre el tamaño de los dominios involucrados en el mapeo.
4. La longitud del plan resultante es preservada en *cierto grado*. Este supuesto será retomado más adelante cuando expliquemos compilabilidad.

En la siguiente sección analizaremos y definiremos formalmente la noción de compilabilidad.

### 5.1.2. Compilabilidad

A partir de los conceptos que aparecen en [41], vamos a enunciar una definición relacionada al tamaño de los planes.

**Definición 5.1.3.** Sea  $f$  un esquema de compilación,  $\Delta$  un plan que resuelve la instancia de planificación  $\Pi$  y  $\Delta'$  un plan que resuelve la instancia de planificación  $F(\Pi)$ . Mediante  $||\Delta||$  denotamos el tamaño de un plan para el formalismo fuente y mediante  $||\Delta'||$  denotamos el tamaño de un plan para el formalismo destino. Asimismo, mediante  $||\Pi||$  se denota el tamaño de la instancia.

Entonces:

- $f$  es un esquema de compilación que preserva exactamente el tamaño del plan si  $||\Delta'|| \leq ||\Delta|| + k$ , para alguna constante entera positiva  $k$ .
- $f$  es un esquema de compilación que preserva linealmente el tamaño del plan si  $||\Delta'|| \leq c||\Delta|| + k$ , para las constantes positivas  $c$  y  $k$ .
- $f$  es un esquema de compilación que preserva polinomialmente el tamaño del plan si  $||\Delta'|| \leq p(||\Delta||, ||\Pi||)$ , para algún polinomio  $p$ .

A continuación, formalizamos la relación de compilabilidad<sup>4</sup>.

**Definición 5.1.4.** Un formalismo de planificación  $X$  es **compilable** al formalismo  $Y$ , expresado como  $X \preceq^x Y$ , si y sólo si, existe un esquema de compilación de  $X$  a  $Y$ .

Entonces:

- Si  $X \preceq^1 Y$ , entonces el tamaño del plan es preservado exactamente.
- Si  $X \preceq^c Y$ , entonces el tamaño del plan es preservado linealmente (en  $||\Delta||$ ).
- Si  $X \preceq^p Y$ , entonces el tamaño del plan es preservado polinomialmente (en  $||\Delta||$  y  $||\Pi||$ ).
- Si  $X \preceq_p^x Y$ , entonces la compilación es en tiempo polinomial y el tamaño del plan es preservado polinomialmente (en  $||\Delta||$  y  $||\Pi||$ ).

Los casos en los que la compilación de formalismos preserva el tamaño de un plan, ya sea exacta o linealmente, indican que el formalismo destino que estamos considerando es, al menos, tan expresivo como el formalismo fuente. Sin embargo, si un esquema de compilación requiere de un crecimiento polinomial respecto al tamaño de los planes, entonces el formalismo fuente es más expresivo que el formalismo destino.

Por último, en [41], se enuncian dos propiedades importantes para la relación de compilabilidad:

---

<sup>4</sup>En Inglés, *compilability*.

**Proposición 5.1.1.** *Las relaciones  $\preceq^x$  y  $\preceq_p^x$  son reflexivas y transitivas.*

**Proposición 5.1.2.** *Si  $X \subseteq Y$ , entonces  $X \preceq_p^1 Y$ .*

En las siguientes secciones clasificaremos y ejemplificaremos las variantes STRIPS consideradas y propondremos una notación equivalente para PDDL. Además, usando la relación de compilabilidad, definiremos esquemas de compilación que luego serán implementados en nuestro traductor. Para esto, proponemos las capas de traducción que se indican en la figura 5.1.

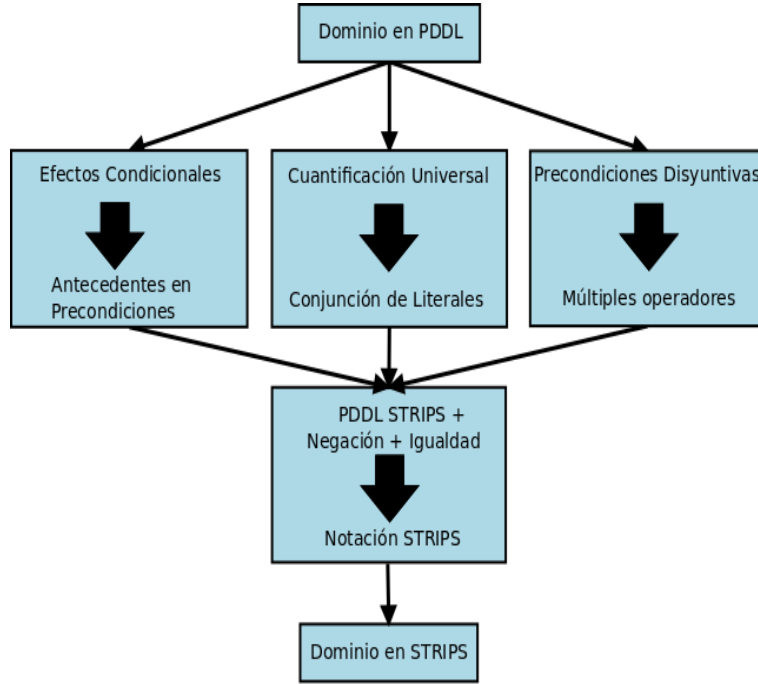


Figura 5.1: Capas de Traducción

## 5.2. PDDL<sub>STRIPS</sub>

El STRIPS estándar, estudiado en el capítulo 2 y definido por Russell y Norving en [48], es el formalismo más básico de esta clasificación en términos de expresividad. Básicamente, describe acciones mediante precondiciones y efectos. Las precondiciones son conjunciones de literales positivos al igual que los estados inicial y final. Los efectos son expresados como una conjunción de literales. Los literales positivos son incluidos en la lista de agregados y los negativos en la lista de borrados de la acción. STRIPS estándar no permite efectos condicionales, disyunción ni cuantificación universal en precondiciones.

A continuación, estudiaremos la traducción de STRIPS expresado en PDDL.

**Proposición 5.2.1.** *STRIPS  $\preceq^1$  STRIPS.*

*Demostración.* Por propiedad reflexiva de la relación de compilabilidad, proposición 5.1.1, concluimos que STRIPS es *compilable* a STRIPS preservando exactamente el tamaño de los planes.  $\square$

Luego, si denotamos como PDDL<sub>STRIPS</sub> al formalismo PDDL acotado a la expresividad de STRIPS y lo reemplazamos en la proposición anterior, podemos concluir que:

**Corolario 5.2.1.** *PDDL<sub>STRIPS</sub>  $\preceq^1$  STRIPS.*



Por lo tanto, PDDLSTRIPS es *compilable* a STRIPS estándar preservando exactamente el tamaño de los planes. Este lenguaje es el más básico en términos de expresividad.

Veamos, mediante un ejemplo, cómo PDDLSTRIPS es traducido a STRIPS.

**Ejemplo 5.2.1.** *Dada una instancia del problema del “Mundo de Bloques” en PDDLSTRIPS, la traducción procede adicionando efectos negados a la lista de borrados y efectos positivos a la lista de agregados de la especificación STRIPS.*

```
% Dominio PDDL
(define (domain bkw)
  (:requirements :strips)
  (:predicates (clear ?x)
               (ontable ?x)
               (armempty)
               (holding ?x)
               (on ?x ?y))

  (:action pickup
    :parameters (?ob)
    :precondition (and (clear ?ob) (armempty))
    :effect (and (holding ?ob) (not (clear ?ob)) (not (armempty))))

  (:action stack
    :parameters (?ob ?underob)
    :precondition (and (clear ?underob) (holding ?ob))
    :effect (and (clear ?ob) (on ?ob ?underob) (armempty)
                 (not (clear ?underob)) (not (holding ?ob))))
)

% Problema PDDL
(define (problem pb1)
  (:domain bkw)
  (:objects a b c)
  (:goal (on a b))
  (:init (ontable c) (ontable b)
         (on a c) (clear a) (clear b) (armempty))
)
```

*La representación en STRIPS del problema anterior es la siguiente:*

```
% pickup(X,Y)
Precondiciones: clear(X), armempty
Borrados: clear(X), armempty
Agregados: holding(X)

% stack(X,Y)
Precondiciones: clear(Y), holding(X)
Borrados: clear(Y), holding(X)
Agregados: armempty, clear(X), on(X,Y)
```

```
% Problema STRIPS
holds(ontable(c),init)
holds(ontable(b),init)
holds(on(a,c),init)
holds(clear(a),init)
holds(clear(b),init)
holds(armempty,init)
```

En base a lo expuesto, podemos concluir que  $PDDL_{STRIPS}$  es soportado por el Planificador Continuo.

### 5.3. $PDDL_L$

En el caso del predicado de igualdad, el escenario que se presenta es diferente. STRIPS estándar no soporta el uso de este predicado en las precondiciones, no obstante, es aceptado en algunas variantes STRIPS. Ocurre algo similar para el caso de la negación ya que no está incluida en la definición de STRIPS. Algunos planificadores aceptan negación pero, únicamente, del predicado igualdad. Este es el caso del lenguaje de representación del Planificador Continuo.

Por su parte, PDDL si permite incluir negación en las precondiciones y en las metas de los problemas de planificación posibilitando la definición de acciones como indicamos a continuación:

**Ejemplo 5.3.1.** *Podemos modelar en PDDL la acción stack del “Mundo de Bloques” permitiendo que un bloque,  $?X$ , sea apilado sobre otro,  $?Z$ , si  $?Z$  no es un bloque.*

```
(:action stack
:parameters (?X ?Y ?Z)
:precondition (and (clear ?X) (clear ?Z) (on ?X ?Y) (not (block ?Z)))
:effects (and (on ?X ?Z) (clear ?Y) (not (on ?X ?Y)))
)
```

Sin embargo, como mencionamos al inicio de esta sección, esta característica no es soportada directamente por STRIPS estándar, por lo tanto, es necesaria una traducción.

La adición de la negación a STRIPS da lugar a otra variante del formalismo llamada  $STRIPS_L$ . Esta variante tiene la particularidad, a diferencia de STRIPS estándar, de permitir negación de literales en las precondiciones.

A continuación, enunciamos, sin demostrar, el siguiente resultado postulado por Nebel en [40]:

**Proposición 5.3.1.**  $STRIPS_L \preceq_p^1 STRIPS$

A partir de este resultado, concluimos que  $STRIPS_L$  es *compilable* a STRIPS, en tiempo polinomial y el tamaño de los planes es preservado exactamente.

En este sentido, también definimos a  $PDDL_L$  como el formalismo obtenido de restringir PDDL al requerimiento de negación (STRIPS está implícito por ser el requerimiento por defecto de PDDL). Dado que  $PDDL_L$  es lo mismo que  $STRIPS_L$  en términos de nivel expresivo y, por la proposición 5.3.1, podemos concluir que:

**Corolario 5.3.1.**  $PDDL_L \preceq_p^1 STRIPS$

Es decir,  $PDDL_L$  es *compilable* a STRIPS en tiempo polinomial y el tamaño de los planes es preservado exactamente.

La definición original de STRIPS<sub>L</sub> enunciada por Nebel sólo establece el uso de literales negativos en precondiciones, pero nada concluye sobre la aceptación, o no, del predicado de igualdad. Además, la igualdad no es soportada por STRIPS estándar, aunque sí es soportada por el lenguaje de representación del Planificador Continuo.

Entonces, con el objetivo de incluir el predicado en el contexto de un subconjunto acotado de PDDL, en este trabajo de tesis, vamos a considerar el predicado de igualdad como parte de STRIPS<sub>L</sub> y, en consecuencia, de PDDL<sub>L</sub>

El próximo ejemplo muestra cómo modelar acciones en PDDL<sub>L</sub> usando el predicado de igualdad:

**Ejemplo 5.3.2.** *La acción `stack` permite apilar un bloque sólo si el destino es la mesa. `table` es una constante.*

```
(:action stack
  :parameters (?X ?Y)
  :precondition (and (clear table) (= ?Y table))
  :effect (and (on ?X table) (not (clear table)))
)
```

Por último, es necesario definir una restricción importante sobre PDDL<sub>L</sub> para que pueda ser íntegramente soportado por el Planificador Continuo. Dicha restricción es impuesta sobre el requerimiento de negación:

- Permitiremos el uso del `not`, en precondiciones, sólo cuando es aplicado junto con el predicado de igualdad, tal es el caso del término `(not (= ?Y ?Z))`<sup>5</sup>.

Veamos, mediante un ejemplo, cómo traducir PDDL<sub>L</sub> a la representación STRIPS del Planificador Continuo.

**Ejemplo 5.3.3.** *Traducimos la siguiente acción con el predicado de igualdad a notación STRIPS:*

```
(:action stack
  :parameters (?X ?Y)
  :precondition (and (clear table) (= ?Y table))
  :effect (and (on ?X table) (not (clear table)))
)
```

```
% stack(X,Y)
Precondiciones: clear(table), Y == table
Borrados: clear(table)
Agregados: on(X,table)
```

*Y la negación del predicado de igualdad en precondiciones:*

```
(:action stack
  :parameters (?X ?Y)
  :precondition (and (clear ?Y) (not(= ?Y table)))
  :effect (and (on ?X ?Y) (not (clear ?Y)))
)
```

*es traducido como:*

---

<sup>5</sup>Existen otras alternativas para tratar con la negación en precondiciones como la presentada por Gazen y Knoblock [20] para el planificador Graphplan. Sin embargo, un análisis exhaustivo de estas alternativas está fuera del alcance de esta investigación.

```
% stack(X,Y)
Precondiciones: clear(Y), Y \== table
Borrados: clear(Y)
Agregados: on(X,Y)
```

A partir de lo expuesto, podemos concluir que  $PDDL_L$ , con la restricción impuesta sobre el predicado PDDL de igualdad y la negación, es soportado por el Planificador Continuo.

## 5.4. PDDL<sub>C</sub>

El formalismo obtenido de adicionar **efectos condicionales** a STRIPS es llamado  $STRIPS_C$ . Gazen y Knoblock [20] proponen una manera particular de tratar con estos efectos postulando el siguiente resultado, también estudiado por Nebel en [41]:

**Proposición 5.4.1.**  $STRIPS_C \preceq_p^x STRIPS_L$

El esquema propuesto por Gazen y Knoblock, también formulado por Weld en [57], es denominado **expansión total**<sup>6</sup>. Este enfoque es el más simple<sup>7</sup>. Como indica la figura 5.2, a partir de una acción con efectos condicionales, se generan esquemas de acciones STRIPS considerando todas las combinaciones consistentes de los antecedentes en los efectos condicionales. Esta alternativa tiene como ventaja la simplicidad, sin embargo, puede generar una cantidad exponencial de esquemas, a pesar de que no es común a menos que se definan en conjunto con efectos cuantificados universalmente<sup>8</sup>. Si una acción tiene  $n$  efectos condicionales y cada uno contiene  $m$  conjuntos de antecedentes, la expansión completa de una acción puede producir  $n^m$  acciones STRIPS. Este esquema de traducción es también analizado por Nebel en [41] donde concluye que esta traducción no puede ser mejorada aún si se aceptan planes que preservan su tamaño linealmente.

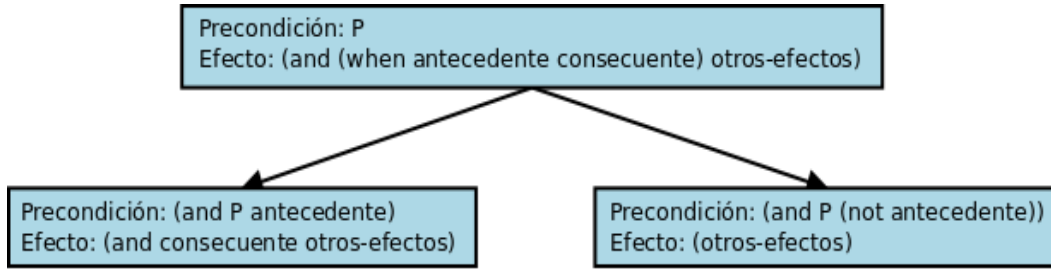


Figura 5.2: Expansión Total para Efectos Condicionales

En este sentido, enunciaremos la siguiente proposición, asumiendo por simplificación, que aceptamos planes que preservan su tamaño polinomialmente.

**Proposición 5.4.2.**  $STRIPS_C \preceq_p^x STRIPS$

*Demostración.* Aplicando la proposición 5.4.1 y luego, por propiedad transitiva de la relación de compilabilidad, concluimos que  $STRIPS_C \preceq_p^x STRIPS$ .  $\square$

<sup>6</sup>En Inglés, *full expansion*.

<sup>7</sup>Weld [57] también propone dos enfoques más para realizar esta traducción: Expansión Factorizada, *factored expansion* y Expansión Parcialmente Factorizada, *partially factored*. Ambos están fuera del alcance de la investigación.

<sup>8</sup>Notar que la cuantificación universal en efectos no es soportada por nuestro traductor y está fuera del alcance de la investigación. Por lo tanto, esta es una simplificación resultante de la definición inicial del subconjunto PDDL establecido en el capítulo 2.

Definiendo PDDL<sub>C</sub> como una variante de PDDL con, únicamente, efectos condicionales y, dado que es lo mismo que STRIPS<sub>C</sub> en términos de expresividad, reemplazamos en la proposición anterior y obtenemos:

**Corolario 5.4.1.**  $PDDL_C \preceq_p^x STRIPS$

De este resultado, concluimos que PDDL<sub>C</sub> es *compilable* a STRIPS en tiempo polinomial y el tamaño de los planes es preservado polinomialmente.

A continuación, vemos un ejemplo para analizar el esquema de traducción definido en esta sección y luego concluimos sobre los supuestos de simplificación en relación al uso de efectos condicionales en acciones.

**Ejemplo 5.4.1.** *Redefinimos la acción `stack` del “Mundo de Bloques” en PDDL<sub>C</sub>, aplicamos expansión completa y mostramos los STRIPS equivalentes.*

```
(:action stack
  :parameters (?X ?Y ?Z)
  :precondition (and (clear ?X) (clear ?Z) (on ?X ?Y))
  :effects (and (on ?X ?Z) (clear ?Y) (not (on ?X ?Y))
                (when (not (= table ?Z)) (not (clear ?Z))))
)
```

*Generamos las dos instancias de `stack` aplicando todas las combinaciones consistentes del antecedente del efecto condicional:*

```
(:action stack
  :parameters (?X ?Y ?Z)
  :precondition (and (clear ?X) (clear ?Z)
                    (on ?X ?Y) (not (= table ?Z)))
  :effects (and (on ?X ?Z) (clear ?Y)
                (not (on ?X ?Y))
                (not (clear ?Z)))
)

(:action stack1
  :parameters (?X ?Y ?Z)
  :precondition (and (clear ?X) (clear ?Z)
                    (on ?X ?Y) (= table ?Z))
  :effects (and (on ?X ?Z) (clear ?Y) (not (on ?X ?Y)))
)
```

*Finalmente, expresamos estas acciones usando notación STRIPS:*

```
% stack(X,Y,Z)
Precondiciones: clear(X), clear(Z), on(X,Y), Z \== table
Borrados: on(X,Y), clear(Z)
Agregados: on(X,Z), clear(Y)

% stack1(X,Y,Z)
Precondiciones: clear(X), clear(Z), on(X,Y), Z == table
Borrados: on(X,Y)
Agregados: on(X,Z), clear(Y)
```

A partir de este análisis, establecemos los siguientes supuestos de simplificación para modelar dominios de planificación con efectos condicionales:

1. Los antecedentes sólo pueden ser de la forma:  $(= X Y)$  y  $(\text{not } (= X Y))$ . Esta restricción se debe a que además de STRIPS estándar aceptamos el uso del predicado de igualdad y su negación en las precondiciones de las acciones.
2. Los efectos condicionales pueden definir sólo un antecedente y sólo un consecuente.

En base a lo expuesto, concluimos que PDDL<sub>C</sub> es soportado por el Planificador Continuo con los supuestos establecidos previamente.

## 5.5. PDDL<sub>D</sub>

Las **precondiciones disyuntivas** permite el uso del operador ‘or’ en precondiciones y descripciones de metas. Este operador no es soportado por STRIPS estándar y, por lo tanto, es necesario realizar un preprocesamiento.

A continuación, enunciamos y demostramos formalmente la relación entre STRIPS<sub>D</sub> y STRIPS.

### Proposición 5.5.1. $STRIPS_D \preceq_p^1 STRIPS$

*Demostración.* En base a la proposición  $STRIPS_D \preceq_p^1 STRIPS_L$ , demostrada por Nebel en [41], la proposición 5.3.1 y la propiedad transitiva de la relación de compilabilidad, concluimos que existe un esquema de compilación de tiempo polinomial de  $STRIPS_D$  a  $STRIPS$  que preserva exactamente el tamaño del plan. El siguiente esquema de compilación sería una alternativa posible.

Sea  $\Sigma$  un conjunto finito de átomos proposicionales y  $L_\Sigma$  un conjunto de fórmulas conjuntivas compuestas por literales sobre  $\Sigma$ . Sea también,  $L$  un conjunto de literales y  $L \subseteq \hat{\Sigma}$  donde  $\hat{\Sigma} = \top \cup \perp \cup \text{literales}(\Sigma)$ .

Por cada operador:

$$o = \langle (c_1 \vee \dots \vee c_n), L \rangle$$

donde  $c_i \in L_\Sigma$ , se generan los siguientes operadores:

$$o_i = \langle (c_i), L \rangle$$

□

Dado que STRIPS<sub>D</sub> es equivalente, en términos de expresividad, a PDDL con precondiciones disyuntivas como único requerimiento, podemos denotar este formalismo como PDDL<sub>D</sub>. Por lo tanto, concluimos el siguiente corolario:

### Corolario 5.5.1. $PDDL_D \preceq_p^1 STRIPS$

Este corolario nos indica que PDDL<sub>D</sub> puede ser compilado a STRIPS en tiempo polinomial y preservando exactamente el tamaño de los planes.

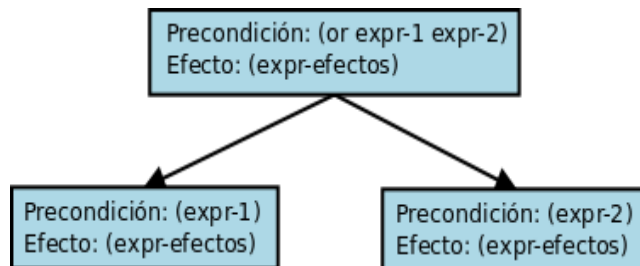


Figura 5.3: Traducción de Precondiciones Disyuntivas

Las precondiciones de las acciones que contienen disyunción son expresadas en forma normal conjuntiva (i.e., como una conjunción de disyunciones) y la compilación a STRIPS implica expresar esta precondición como forma normal disyuntiva (i.e., como disyunción de conjunciones). Por último, cada subexpresión de la disyunción es la precondición de una nueva acción STRIPS como muestra la figura 5.3. A diferencia de los efectos condicionales, no hay un crecimiento exponencial de acciones dado que esta cantidad es sólo proporcional a la cantidad de disyuntores en la precondición de la acción.

El próximo ejemplo ilustra la traducción propuesta:

**Ejemplo 5.5.1.** *La precondición incluye el nuevo requerimiento **or** estableciendo que un objeto, ?X, puede ser apilado sobre un objeto, ?Z, si ?Z es la mesa (**istable**) u otro bloque sin ningún bloque apilado.*

```
(:action stack
  :parameters (?X ?Y ?Z)
  :precondition (and (or (istable ?Z) (clear ?Z))
                    (clear ?X) (on ?X ?Y))
  :effect (and (on ?X ?Z) (clear ?Y)
              (not (clear ?Y)) (not (on ?X ?Y)))
)
```

*Luego, la precondición queda expresada de la siguiente manera:*

```
(:action stack
  :parameters (?X ?Y ?Z)
  :precondition (or (and (istable ?Z) (clear ?X) (on ?X ?Y))
                  (and (clear ?Z) (clear ?X) (on ?X ?Y)))
  :effect (and (on ?X ?Z) (clear ?Y)
              (not (clear ?Y)) (not (on ?X ?Y)))
)
```

*El paso final del algoritmo consiste en generar una nueva acción **stack1** con los mismos efectos que la acción original.*

```
(:action stack
  :parameters (?X ?Y ?Z)
  :precondition (and (istable ?Z) (clear ?X) (on ?X ?Y))
  :effect (and (on ?X ?Z) (clear ?Y)
              (not (clear ?Y)) (not (on ?X ?Y)))
)

(:action stack1
  :parameters (?X ?Y ?Z)
  :precondition (and (clear ?Z) (clear ?X) (on ?X ?Y))
  :effect (and (on ?X ?Z) (clear ?Y)
              (not (clear ?Y)) (not (on ?X ?Y)))
)
```

*Por último, escribimos las acciones en notación STRIPS:*

```
% stack(X,Y,Z)
Precondiciones: istable(Z), clear(X), on(X,Y)
```

Borrados: `clear(Y), on(X,Y)`  
 Agregados: `clear(Y), on(X,Z)`

```
% stack1(X,Y,Z)
Precondiciones: clear(Z), clear(X), on(X,Y)
Borrados: clear(Y), on(X,Y)
Agregados: clear(Y), on(X,Z)
```

En base a este análisis, concluimos que  $PDDL_D$  puede ser traducido a STRIPS y, por lo tanto, es soportado por el Planificador Continuo.

## 5.6. $PDDL_u$

Las **precondiciones universales** implican el uso del cuantificador universal ‘ $\forall$ ’, usualmente leído como “para todo”. El uso de este cuantificador en precondiciones permite modelar acciones del mundo real como, por ejemplo, el comando `rmdir` de *UNIX*, que elimina un determinado directorio si *todos* los archivos dentro de él fueron eliminados previamente.

Las precondiciones universales no son soportadas en STRIPS estándar y, por lo tanto, es necesaria una traducción de este requerimiento. El enfoque es simple. Los cuantificadores universales son traducidos a una conjunción de literales. Cada uno de esos literales se genera instanciando el parámetro cuantificado con cada uno de los objetos declarados en la definición del problema de planificación.

**Ejemplo 5.6.1.** Sean los objetos: *block1* y *block2* y supongamos la siguiente precondición: (*forall* (*?a*) (*clear*(*?a*))). Instanciando *?a* con los objetos *block1* y *block2* obtenemos la precondición equivalente: (*and* (*clear*(*block1*)) (*clear*(*block2*))).

Antes de formalizar la traducción de precondiciones universales, vamos a introducir la definición de *Base de Herbrand* extraída de [57]:

**Definición 5.6.1.** Sea  $\Delta$  una sentencia de primer orden y libre de funciones. La *Base de Herbrand*  $\Upsilon$  es definida, recursivamente, de la siguiente manera:

$$\Upsilon(\Delta) = \Delta, \text{ si } \Delta \text{ no contiene cuantificadores}$$

$$\Upsilon(\forall_{t_1} x \Delta(x)) = \Upsilon(\Delta_1) \wedge \dots \wedge \Upsilon(\Delta_n)$$

donde  $\Delta_i$  corresponde a cada posible interpretación de  $\Delta(x)$  bajo el universo compuesto por los objetos de tipo  $t_1$  del problema de planificación.

Además, establecemos los siguientes supuestos de simplificación:

1. La cuantificación universal sólo es permitida en precondiciones<sup>9</sup>.
2. El mundo a modelar incluye un conjunto finito y estático de objetos. Para el ejemplo del “Mundo de Bloques”, la cantidad de *bloques* debe ser finita y permanecer invariable mientras se planifica la solución.
3. Todos los objetos involucrados deben ser del mismo tipo. Por lo tanto, no explicitamos el tipo  $t_1$  de los objetos, tal como lo expresa la definición 5.6.1. Continuando con el ejemplo del “Mundo de Bloques”, los únicos objetos declarados deben ser *bloques*.

---

<sup>9</sup>PDDL permite el uso de cuantificación universal también en metas.



En este sentido, estamos en condiciones de formular y demostrar el siguiente teorema. Sea STRIPS<sub>u</sub>, el formalismo resultante de adicionar precondiciones universales al STRIPS estándar. Entonces:

**Teorema 5.6.1.**  $STRIPS_u \preceq_p^1 STRIPS$

*Demostración.* Vamos a demostrar que existe una esquema de compilación de tiempo polinomial que preserve exactamente el tamaño de los planes. Por definición de *Base de Herbrand* y por los supuestos de simplificación, podemos definir el siguiente esquema de compilación.

Sea  $\Sigma$  un conjunto finito de átomos proposicionales y  $L_\Sigma$  un conjunto de fórmulas conjuntivas compuestas por literales sobre  $\Sigma$ . Sea también,  $L$  un conjunto de literales y  $L \subseteq \hat{\Sigma}$  donde  $\hat{\Sigma} = \top \cup \perp \cup \text{literales}(\Sigma)$ .

Para cada operador:

$$o = \langle (\forall x_i (c_j(x_i))), L \rangle$$

donde  $c_j(x_i)$  son predicados de aridad  $n$  y  $c_j(x_i) \in L_\Sigma$ , es posible generar un nuevo operador expresado como una conjunción de literales:

$$o = \langle (c(x_1) \wedge \dots \wedge c(x_n)), L \rangle$$

Por lo tanto, aplicando la proposición 5.3.1 y luego la propiedad transitiva de la relación de compilabilidad, concluimos que  $STRIPS_u \preceq_p^1 STRIPS$ .  $\square$

Dado que STRIPS<sub>u</sub> es equivalente, en términos de expresividad, a PDDL con precondiciones universales como único requerimiento, podemos denotar este formalismo como PDDL<sub>u</sub>. Luego, enunciamos el siguiente corolario:

**Corolario 5.6.1.**  $PDDL_u \preceq_p^1 STRIPS$

Este corolario nos indica que PDDL<sub>u</sub> puede ser compilado a STRIPS en tiempo polinomial y preservando exactamente la longitud de los planes.

Aplicamos este resultado al siguiente ejemplo:

**Ejemplo 5.6.2.** *La precondición para la acción **stack** es que todo objeto en el problema debe estar sobre la mesa (**ontable**) para que dos bloques puedan ser apilados.*

*Dado el siguiente problema PDDL y la acción **stack** definida en PDDL<sub>u</sub>:*

```
(define (problem pb1)
  (:domain bkwup)
  (:objects a b c)
  (:goal (on a b))
  (:init (ontable c) (ontable b) (ontable a)
         (on a c) (clear a) (clear b) (armempty))
)

(:action stack
  :parameters (?ob ?underob)
  :precondition (and (forall (?block) (ontable ?block))
                    (clear ?underob) (holding ?ob))
  :effect (and (clear ?ob) (on ?ob ?underob) (armempty)
              (not (clear ?underob)) (not (holding ?ob)))
)
```

*Aplicamos el esquema de compilación presentado y obtenemos la siguiente acción:*

```
(:action stack
:parameters (?ob ?underob)
:precondition (and (ontable a) (ontable b)(ontable c)
                  (clear ?underob)(holding ?ob))
:effect (and (clear ?ob) (on ?ob ?underob) (armempty)
            (not (clear ?underob)) (not (holding ?ob)))
)
```

*Finalmente, la representamos en notación STRIPS.*

```
% stack(X,Y)
Precondiciones: ontable(a), ontable(b), ontable(c), clear(Y), holding(X)
Borrados: clear(Y), holding(X)
Agregados: clear(X), on(X,Y), armempty
```

A partir de lo expuesto en esta sección, podemos concluir que la variante PDDL<sub>u</sub> es soportada por el Planificador Continuo.

En este capítulo hemos definido los esquemas de traducción correspondientes para los requerimientos PDDL incluidos en la investigación. Tales esquemas permiten traducir una especificación en un subconjunto de PDDL a una representación equivalente en la notación STRIPS soportada por el Planificador Continuo.

En el próximo capítulo estudiaremos la arquitectura propuesta para el traductor, formalizaremos el lenguaje fuente y destino y detallaremos aspectos principales de la implementación en Ciao Prolog.

## Capítulo 6

# Implementación Propuesta

Tal como comentamos en capítulo 2, la irrupción de PDDL como lenguaje estándar para representación de problemas de planificación ha generado nuevos desafíos. Estos desafíos están motivados por la necesidad de desarrollar y/o extender herramientas que lo soporten y también, en ampliar su expresividad para adecuarlo a los dominios de aplicación destino.

Un caso puntual es el algoritmo de Planificación Continua, implementado por Moya en [36], y que hemos descripto en el capítulo 3. En este algoritmo, que hereda algunos conceptos del planificador POP [30], todas las especificaciones son representadas usando variantes de STRIPS y, por lo tanto, es necesario una traducción para que el planificador pueda beneficiarse de las características de PDDL.

El capítulo está organizado de la siguiente manera. La sección 6.1 presenta una descripción general de la solución, la definición de los lenguajes fuente y destino del traductor propuesto y la arquitectura diseñada para su implementación. En la sección 6.2 se analizan las características más relevantes de Ciao Prolog (principalmente, Expansiones Sintácticas y DCG) y se concluye acerca de su elección como entorno de desarrollo. Por último, la sección 6.3 detalla la implementación del traductor y la sección 6.4 presenta algunos ejemplos de uso.

### 6.1. Descripción General de la Solución

En esta tesis proponemos el diseño y desarrollo de un módulo traductor de un subconjunto de lenguaje PDDL para la especificación de problemas de planificación. El objetivo central es que estos problemas puedan ser manipulados por el Framework de Planificación Continua. El traductor propuesto toma una especificación PDDL como entrada y la traduce a una especificación equivalente STRIPS. La implementación es una expansión sintáctica para Ciao Prolog.

La arquitectura inicial del traductor es ilustrada en la figura 6.1. El módulo recibe una especificación en un **lenguaje fuente** y retorna una especificación equivalente en un **lenguaje destino**.

A continuación, especificaremos los lenguajes fuente y destino y luego estudiaremos el módulo traductor con más detalle.

#### 6.1.1. El Lenguaje Fuente

El lenguaje fuente abarca los siguientes formalismos definidos en el capítulo 5:

- PDDL<sub>STRIPS</sub>. PDDL con el requerimiento **strips**. La expresividad subyacente es equivalente a STRIPS estándar.
- PDDL<sub>L</sub>. PDDL con los requerimientos **strips** y **equality**. También permite la inclusión de negación de igualdad en precondiciones. El predicado de igualdad y su negación son

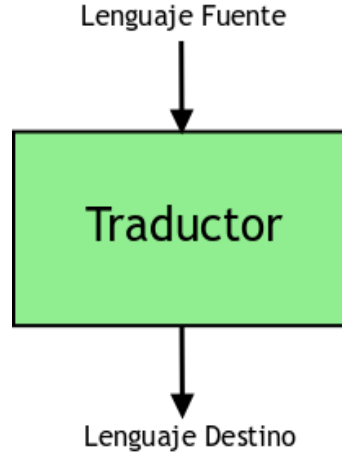


Figura 6.1: Módulo Traductor

soportados por el Planificador Continuo y fueron considerados como parte del lenguaje fuente aunque no estén definidos en STRIPS estándar.

- PDDL<sub>C</sub>. Esta variante PDDL soporta **strips**, **equality** y **conditional-effects**. Los efectos condicionales están restringidos a la definición de un solo antecedente y un solo consecuente. Los antecedentes deben ser de la forma `(= X Y)` o `(not (= X Y))`.
- PDDL<sub>D</sub>. PDDL con los requerimientos **strips**, **equality** y **disjunctive-preconditions**.
- PDDL<sub>u</sub>. PDDL con los requerimientos **strips**, **equality** y **universal-preconditions**. Las precondiciones cuantificadas universalmente requieren que el problema defina por un conjunto finito y estático de objetos. Todos estos objetos deben ser del mismo tipo.

Cada problema de planificación de entrada debe estar especificado en cualquiera de estos formalismos pertenecientes al lenguaje fuente.

### 6.1.2. El Lenguaje Destino

Como parte del lenguaje destino, vamos a identificar dos representaciones similares. Por un lado, la notación STRIPS genérica que ofrece el traductor y por otro, cómo esta representación es adaptada al algoritmo de Planificación Continua.

El módulo traductor propuesto ofrece una notación genérica que permite personalizar la definición de STRIPS según el algoritmo planificador al que se quiera integrar dicho módulo. Esta representación permite reutilizar el traductor en otros algoritmos, cuyo lenguaje de representación sea similar a STRIPS, sin la necesidad de modificar los módulos principales de traducción. La notación genérica definida, llamada *Prolog-like*, es la siguiente:

- *Precondiciones, Agregados y Borrados*. Por cada acción en el dominio de entrada, el traductor genera los predicados **preconditions**, **achieves** y **deletes** respectivamente, con los siguientes argumentos: **action\_name(parameters)** y una lista de predicados [**predicate**<sub>1</sub>(**parameters**<sub>1</sub>), ..., **predicate**<sub>N</sub>(**parameters**<sub>N</sub>)], donde **parameters**<sub>i</sub> es una lista de variables Prolog separada por comas.
- *Objetos, Meta y Estado Inicial*. El traductor genera un predicado **objects(obj<sub>1</sub>, obj<sub>2</sub>, ..., obj<sub>N</sub>)** donde **obj<sub>i</sub>** es un átomo Prolog. Además, genera **goal(fact<sub>g</sub>)** donde **fact<sub>g</sub>** son hechos Prolog de la forma **fact(parameters<sub>g</sub>)** y **parameters<sub>g</sub>** es una lista de átomos separados por comas. Por último, **init(fact<sub>1</sub>, fact<sub>2</sub>, ..., fact<sub>N</sub>)** es la definición del estado inicial donde, **fact<sub>i</sub>** es de la forma **init(parameters<sub>i</sub>)** y **parameters<sub>i</sub>** es una lista de átomos separados por comas.

A continuación, mostramos un ejemplo con la notación genérica definida para el traductor.

**Ejemplo 6.1.1. % Dominio**

```
preconditions(action_name_1(parameters),
              [predicate_1(parameters_1),
               ...
               predicate_N(parameters_N)]).
achieves(action_name_1(parameters),
          [predicate_1(parameters_1),
           ...
           predicate_N(parameters_N)]).
deletes(action_name_1(parameters),
         [predicate_1(parameters_1),
          ...
          predicate_N(parameters_N)]).

% Problema
(domain(domain_name),
 objects(obj1,obj2,...,objN),
 goal(fact_g),
 init(fact_1,fact_2,...,fact_N)).
```

Teniendo en cuenta que uno de los objetivos de nuestra investigación es integrar el módulo al Framework de Planificación Continua, el lenguaje destino del traductor está restringido al lenguaje de representación soportado por el algoritmo de planificación que implementa el framework. En este caso, los dominios y problemas de planificación están especificados en STRIPS estándar más los predicados de igualdad (denotado como ‘==’) y distinto (denotado como ‘\==’). Cada acción se compone de un predicado **preconditions**, un nuevo predicado **deletes** por cada predicado en la lista de borrados genérica y un predicado **achieves** por cada predicado en la lista genérica de agregados. Para adaptar los problemas, la traducción es similar. Por cada predicado genérico en **init** se genera un nuevo hecho Prolog con la sintaxis **holds(fact<sub>i</sub>,init)**.

Luego, la representación Ciao Prolog lograda es la siguiente:

```
% action(X,Y,Z)
preconditions(action(X,Y,Z),[pred1(X),pred2(Y),pred3(Z)]):- X == Y,Y \== Z.
achieves(action(X,Y,Z),pred1(X)).
achieves(action(X,Y,Z),pred3(Z)).
deletes(action(X,Y,Z),pred2(Y)).

% Problem
holds(pred1(object1),init).
holds(pred2(object2),init).
holds(pred3,init).
```

A continuación, detallaremos la arquitectura del traductor y analizaremos el comportamiento de cada uno de sus módulos.

### 6.1.3. Módulos de la Arquitectura

La arquitectura del traductor está organizada en los siguientes módulos:

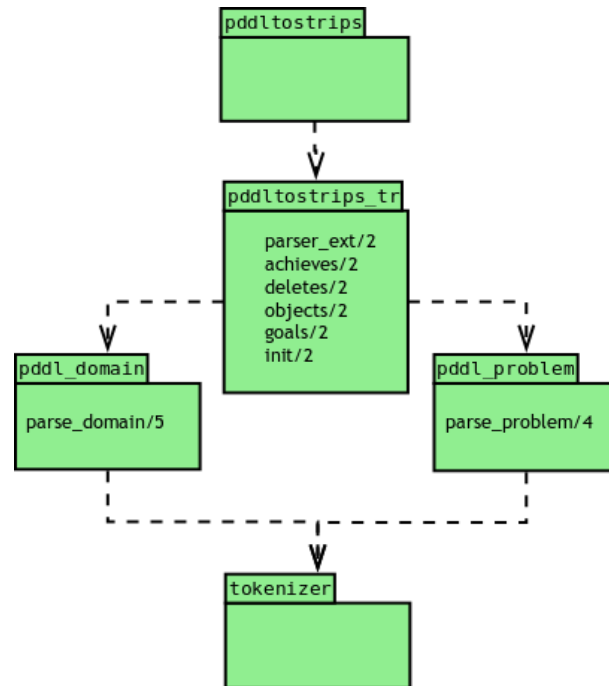


Figura 6.2: Arquitectura Modular del Traductor

- **pddltostrrips**: es el módulo “controlador” y debe incluirse en la especificación PDDL de entrada usando la sentencia Prolog `use_package(pddltostrrips)`. Permite cargar el módulo **pddltostrrips\_tr** y el predicado **parser\_ext** en tiempo de compilación para comenzar la traducción.
- **pddltostrrips\_tr**: este módulo incluye la definición del predicado **parser\_ext**. El predicado recibe un dominio y un problema en PDDL y retorna la especificación STRIPS correspondiente. Interactúa con los módulos **tokenizer**, **parse\_problem** y **parse\_domain**. Este módulo también es el que interpreta la representación genérica STRIPS para adaptar el traductor al planificador destino.
- **pddl\_domain**: usando el paquete **dcg**, traduce la definición de un dominio PDDL, expresado en el lenguaje fuente, a una lista de predicados Prolog en la notación STRIPS genérica definida en la sección anterior. El predicado principal de este módulo es **parse\_domain**.
- **pddl\_problem**: usando el paquete **dcg**, traduce la definición de un problema PDDL, expresado en el lenguaje fuente, a una lista de predicados Prolog en la notación STRIPS genérica definida. El predicado principal de este módulo es **parse\_problem**.
- **tokenizer**: este módulo actúa como analizador léxico<sup>1</sup> y obtiene los *tokens* de la definición PDDL de entrada.

Para utilizar este módulo traductor, la especificación PDDL de entrada debe ser importada desde el planificador destino. Esta especificación, a su vez, debe importar el paquete **pddltostrrips** implementado. El ambiente provisto por Ciao Prolog generará, en compilación, los STRIPS equivalentes y el planificador podrá hacer uso de ellos para producir un plan. La figura 6.2 ilustra la disposición e interacción de los módulos mencionados<sup>2</sup>.

<sup>1</sup>Un analizador léxico lee una cadena de caracteres de la especificación en el lenguaje fuente y los agrupa en secuencias llamadas *lexemas*. Para cada *lexema*, el analizador genera un *token* como salida [1].

<sup>2</sup>Las líneas puntuadas entre los módulos representan la dependencia entre ellos.

En la sección siguiente analizaremos y justificaremos la elección de Ciao Prolog como el ambiente de programación ideal para la implementación del traductor PDDL. También, discutiremos brevemente, la noción de expansiones sintácticas y, en particular, la expansión DCG.

## 6.2. Ciao Prolog

El sistema Ciao [7] es un ambiente de programación para el desarrollo de aplicaciones en lenguaje Prolog [52] y en varios otros lenguajes que son extensiones y modificaciones de Prolog. Un aspecto importante de Ciao es que, además de dar soporte para la programación lógica, provee al programador de un conjunto de características de diferentes paradigmas y estilos de programación que pueden ser incluídas en cada módulo de un programa. El lenguaje está diseñado para ser extendido de una manera simple y modular.

Ciao Prolog presenta el siguiente conjunto de características:

- Ofrece un sistema Prolog completo, soportando ISO-Prolog<sup>3</sup> [13], que permite tanto restringir como expandir el lenguaje mediante un sistema modular novedoso.
- Soporta, a través de expansiones, programación funcional, restricciones, objetos, registros, persistencia, reglas de control (*breadth-first search*, *iterative deepening*, etc.), concurrencia, ejecución distribuida y paralela. También incluye librerías para programación Web (por ejemplo, paquetes para manipular lenguaje html, xml, entre otros), sockets e interfaces externas (C, JAVA, Bases de Datos Relacionales, etc.).
- Ofrece un soporte para programación a gran escala con un sistema robusto de módulos, compilación incremental, un lenguaje de aserciones para la declaración de propiedades de programas, inferencia estática y comprobación estática/dinámica de tales aserciones.
- Su compilador genera varias formas de ejecutables *stand-alone*<sup>4</sup> e independientes de la arquitectura. Los módulos de las librerías pueden ser compilados como *bytecode* o como compactos archivos fuentes C y ligados de manera estática, dinámica o cargados automáticamente.
- Finalmente, es distribuido bajo LGPL GNU<sup>5</sup> [18].

Ciao es desarrollado por el grupo CLIP de la Universidad Politécnica de Madrid<sup>6</sup>.

### 6.2.1. Expansiones Sintácticas

El enfoque orientado a extensibilidad de Ciao posibilita la expansión del lenguaje tanto sintáctica como semánticamente. Estas expansiones pueden ser referenciadas en los módulos, sin interferir con otros, gracias a la noción de “paquetes” (*packages*).

Una **expansión sintáctica** [8] toma como entrada una especificación con una determinada sintaxis, posiblemente diferente a la que permite Prolog, y genera una nueva representación, equivalente a su entrada, que puede ser interpretada en Prolog.

En Ciao, este procesamiento es soportado por la directiva `load_compilation_module/1`. Esta directiva permite separar el código que será usado en tiempo de compilación del código que será usado en tiempo de ejecución. De esta manera, permite cargar “en el compilador”, el módulo especificado en su argumento.

Para facilitar la definición de expansiones, Ciao también incluye cuatro directivas más específicas. El compilador invoca estos predicados en el momento apropiado e instancian su primer

<sup>3</sup>International Standard ISO/IEC 13211-1

<sup>4</sup>Un programa stand-alone es uno que puede ejecutarse como un proceso separado.

<sup>5</sup>Lesser GNU General Public License.

<sup>6</sup>CLIP Group. <http://clip.dia.fi.upm.es/index.html>. Disponible en Septiembre de 2012.

argumento con los ítems a ser traducidos. Si el predicado es de aridad 3, el tercer argumento opcional es instanciado con el nombre del módulo donde la traducción se está llevando a cabo (algunas veces necesario durante ciertas expansiones). Si la llamada al predicado de expansión es exitosa, el término retornado en el segundo argumento es usado para reemplazar al original.

En [8], Daniel Cabeza y Manuel Hermenegildo, definen las siguientes directivas de traducción:

- `add_sentence_trans/1`: define la traducción de términos, leídos por el compilador, en el resto de la entrada actual. Para cada término subsecuente, el predicado es nuevamente invocado para obtener un nuevo término que será usado en lugar del término actual. Un ejemplo de este tipo de traducción es el DCG (*Definite Clause Grammar*).
- `add_term_trans/1`: define la traducción de términos y subtérminos, leídos por el compilador, en la entrada actual. Para cada término subsecuente, y recursivamente, para cada subtérmino incluido, el predicado de traducción es invocado para obtener un nuevo término en reemplazo del anterior. Esta traducción comienza luego de que todas las traducciones definidas en `add_sentence_trans/1` son ejecutadas.
- `add_goal_trans/1`: define la traducción de las metas presentes en las cláusulas de la entrada actual. Esta traducción comienza luego de que todas las definidas en `add_sentence_trans/1` y `add_term_trans/1` son ejecutadas.
- `add_clause_trans/1`: define la traducción de las cláusulas de la entrada actual. La traducción comienza antes de `add_goal_trans/1` pero luego de las traducciones `add_sentence_trans/1` y `add_term_trans/1`.

A continuación, y en el contexto de estas directivas, podemos analizar el módulo `pddltostrips.pl` de la implementación propuesta:

```
:- package(pddltostrips).
:- load_compilation_module('pddltostrips_tr').
:- add_sentence_trans(parser_ext/2).
```

La primer declaración indica que `pddltostrips` es un paquete. La directiva `load_compilation_module` carga, en el compilador, el código definido en `pddltostrips_tr`. Por último, la declaración `add_sentence_trans(parser_ext/2)` invoca al predicado `parser_ext/2` exportado por `pddltostrips_tr`. El predicado realiza la traducción correspondiente en tiempo de compilación.

### 6.2.2. Definite Clause Grammar (DCG)

La librería DCG<sup>7</sup> es una expansión sintáctica para las gramáticas libres de contexto. Provee a Prolog de una notación conveniente para definir estas gramáticas. Las reglas gramaticales son un *syntactic sugar* para cláusulas Prolog ordinarias. Cada regla toma una cadena de entrada, analiza alguna subcadena inicial y produce la subcadena restante como salida para un posterior análisis. Los argumentos requeridos para las cadenas de entrada y salida no son indicados explícitamente en la regla gramatical ya que la sintaxis las define implícitamente.

Una regla DCG en Prolog tiene la siguiente forma:

```
head --> body
```

Esto significa que una forma posible para `head` es `body`. Ambos son secuencias de uno o más ítems ligados por el operador de conjunción estándar de Prolog, notado como `','`.

La gramáticas libres de contexto son extendidas, usando DCG, de la siguiente manera:

---

<sup>7</sup>Definite Clause Grammars.



1. Un símbolo no terminal puede ser cualquier término Prolog.
2. Un símbolo terminal puede ser cualquier término Prolog. Los símbolos terminales son escritos como listas Prolog y la lista vacía `[]` representa una secuencia vacía de terminales. Si los terminales son caracteres ASCII, los símbolos son escritos como cadenas.
3. Las condiciones extras pueden ser incluidas en el **body** de la regla gramatical. Las invocaciones se explicitan entre `{ }`.
4. El **head** de la regla consiste de símbolos no terminales. Opcionalmente, puede estar continuado por terminales (en forma de listas).
5. Las alternativas puede definirse en el **body** utilizando el operador `‘;’`.
6. El símbolo de *cut*, `‘!’`, puede ser incluido en el **body** de la regla. No es necesario utilizar `{ }`.

Ilustramos el uso de las DCG con el siguiente ejemplo:

**Ejemplo 6.2.1.** *Extraemos el predicado `pddl2stripsdomain` implementado en el módulo `pddl_domain` de nuestro traductor.*

```
pddl2stripsdomain(Objs,Preconditions, Achieves, Deletes) -->
['(', 'define', '(', 'domain'], name(_Name_Domain),!, [')'],
((requirements_def(Reqs), {_Requirements =..[requirements|Reqs]}) | []),!,
((predicates_def(Dict,Preds), {_Predicates =..[predicates|Preds]}) | []),!,
((constants_def(Consts), {_Constants =..[constants|Consts]}) | []),!,
(actions_def(Dict,Objs,Preconditions, Achieves, Deletes) | []),!,
[')'].
```

*Este predicado es invocado desde:*

```
parse_domain(Input,Objs,Preconditions,Achieves,Deletes)
```

*Como resultado, retorna las listas **Precondiciones**, **Agregados** y **Borrados** correspondientes a la entrada **PDDL** en el argumento **Input**.*

Hasta aquí, hemos presentado los conceptos principales que soportan la implementación de nuestro traductor. Estamos en condiciones de analizar paso a paso, a partir de la próxima sección, la implementación propuesta.

## 6.3. Implementación del Traductor

La traducción comienza cuando se invoca, en tiempo de compilación, al predicado `parser_ext/2`, cuyo código es presentado a continuación:

```
parser_ext((Problem,Domain),STRIPS_POP):-
    atom_codes(Problem,CH),
    atom_codes(Domain,CH2),
    tokenizer(CH,TokensProblem),
    tokenizer(CH2,TokensDomain),
    parse_problem(TokensProblem,Objects,Goals,Init),

    % Problem customizable predicates to the target planner
    objects_pop(Objects,OBJ_POP),
```

```

goals_pop(Goals,GOALS_POP),
init_pop(Init,INIT_POP),
append(OBJ_POP,GOALS_POP,PBL),
append(INIT_POP,PBL,Problem),

parse_domain(TokensDomain,OBJ_POP,Preconditions,Achieves,Deletes),

% Domain customizable predicates to the target planner
achieves_pop(Achieves,ACHIEVES_POP),
deletes_pop(Deletes,DELETES_POP),
append(ACHIEVES_POP,DELETES_POP,D1),
append(Preconditions,D1,Domain),
append(Domain,Problem,STRIPS_POP).

```

Este predicado recibe como entrada el par (Domain,Problem) donde Domain y Problem son cadenas de caracteres escritos como átomos Prolog<sup>8</sup>. El predicado `Ciao atom_codes` toma estos átomos y retorna la lista de códigos ASCII correspondientes. Ambos, dominio y problema PDDL, están en listas separadas de códigos.

El próximo paso es generar, a partir de estas listas, secuencias significativas del lenguaje PDDL. El predicado `tokenizer` implementa esta conducta realizando las siguientes acciones. Por cada conducta mostramos el fragmento de código correspondiente.

- Remueve espacios, saltos de línea y tabulaciones de la lista de códigos ASCII.

```

%layout spaces,newlines,tabs...
tokenizer([C|RC],Words):- ( C=32 ; C=10 ; C=9 ; C=13 ; C=92 ), !,
                           tokenizer(RC,Words).

```

- Trata como palabras a los siguientes símbolos: paréntesis ( ), coma ‘,’ , punto ‘.’ , guión medio ‘-’ , dos puntos ‘:’ y al signo de interrogación ‘?’.

```

% Brackets, comma, period or question marks are treated as separed words
tokenizer([C|RC], [Char|Words_1]) :- ( C=40 ; C=41 ; C=44 ; C=45 ;
                                         C=46 ; C=63 ; C=58 ),
                                       name(Char, [C]), !,
                                       tokenizer(RC, Words_1).

```

- El resto de los caracteres son retornados como *lexemas*. En PDDL estos lexemas son nombres de variables y predicados, palabras reservadas, constantes y el operador de igualdad ‘=’ y negación `not`.

```

% Words
tokenizer([C|RC], [Word|Words_1]):- word([C|RC],Chars,Next),
                                       name(Word,Chars),
                                       tokenizer(Next,Words_1).

```

```

word([C|RC], [], [C|RC]) :- ( C=32 ; C=44 ; C=10 ; C=9 ; C=13 ; C=46 ;

```

---

<sup>8</sup> Átomos Prolog son cadenas de caracteres escritos entre comas simples.

```
C=63 ; C=40 ; C=41 ; C=58 ; C= -1 ) , !.
```

```
word([C|RC],[LC|Chars],Next):- lower_case(C,LC),
                                word(RC,Chars,Next).
```

En este instante de la traducción tenemos la entrada PDDL reducida a listas que serán interpretadas por las gramáticas definidas en los módulos `pddl_problem` y `pddl_domain`.

El predicado `parse_problem/4` implementado en el módulo `pddl_problem` comienza la traducción del problema PDDL. Este predicado recibe la lista de *tokens* retornadas por el `tokenizer` y retorna el problema PDDL expresado en la notación Prolog genérica definida en la sección 6.1.2. La regla principal es analizada a continuación:

```
pddl2stripsproblem(Objects, Goals, Init) -->
    ['(', 'define', '(', 'problem', name(_Name_Problem),!, [')'],
    ['(', ':', 'domain', name(_Name_Domain), !, [')'],
    ((objects_def(Objs), {Objects =..[objects|Objs]} ) | []),!,
    ((goals_def(Goal,_,_), {Goals =..[goal|Goal]} ) | []),!,
    ((init_def(Ini), {Init =..[init|Ini]} ) | []),!,
    [')'].
```

`pddl2stripsproblem` retorna los predicados correspondientes a los objetos, la meta y el estado inicial del problema PDDL. Cada una de estas secciones es analizada, respectivamente por las siguientes reglas invocadas: `objects_def`, `goal_def` e `init_def`.

El próximo paso del traductor es analizar el dominio PDDL, por lo tanto, el predicado `parse_domain/5` es invocado desde `parser_ext` para comenzar la traducción. Recibe, como entrada, la lista de *tokens* del dominio PDDL y los objetos definidos en el problema y retorna tres listas: `Preconditions`, `Achieves` y `Deletes`.

La regla principal en este caso es la siguiente:

```
pddl2stripsdomain(Objs,Preconditions, Achieves, Deletes) -->
    ['(', 'define', '(', 'domain', name(_Name_Domain),!, [')'],
    ((requirements_def(Reqs), {Requirements =..[requirements|Reqs]} ) | []),!,
    ((predicates_def(Dict,Preds), {_Predicates =..[predicates|Preds]} ) | []),!,
    ((constants_def(Consts), {_Constants =..[constants|Consts]} ) | []),!,
    (actions_def(Dict,Objs,Preconditions, Achieves, Deletes) | []),!,
    [')'].
```

Cada una de las secciones en la especificación del dominio PDDL es analizada por las siguientes reglas: `requirements_def`, `predicates_def`, `constants_def` y `actions_def`.

Vamos a estudiar puntualmente la regla `actions_def` debido a que es la que genera las listas STRIPS. El parámetro `Objs` es la lista de objetos definidos en el problema de planificación y `Dict` es un diccionario de variables, donde cada elemento del diccionario es un par (`PDDL Parameter`, `Prolog Variable`) para gestionar la correspondencia de nombres de parámetros en predicados.

El fragmento de código para `actions_def` es el mostrado a continuación.

```

actions_def(Dict,Objs,Preconditions,Achieves,Deletes) -->
  ['(', ':', 'action'], name(Name), !,

  [':', 'parameters'],
  ['('], list_parameters(Dict,Param), !, [')'],
  {nth(1, Name, Name_Ac), Ac = ..[Name_Ac|Param]},

  [':', 'precondition'],
  ['('], list_preconditions(Dict,Objs,P),
  {Pre = ..[preconditions|[Ac,P]]}, !, [')'],

  [':', 'effect'],
  ['('], list_effects(Dict,A, D, P),
  {Ach = ..[achieves|[Ac,A]], Del = ..[deletes|[Ac,D]]}, !, [')'], [')'],

  actions_def(Dict,Objs,PRE2,ACH2,DEL2), !,
  {((instances(P,_), append([Pre],PRE2,Preconditions),
    append([Ach],ACH2,Achieves), append([Del],DEL2,Deletes));
    (generateInstances(Pre,Ach,Del,PRE1,ACH1,DEL1),
    append(PRE1,PRE2,Preconditions), append(ACH1,ACH2,Achieves),
    append(DEL1,DEL2,Deletes)))}.

```

En primer lugar, la regla identifica el nombre de la acción mediante el predicado **name** y posteriormente, retorna la lista de parámetros declarados para la acción. Esta lista es generada desde **list\_parameters**. La regla **list\_preconditions** inicia la traducción de las precondiciones de la acción y retorna todos los predicados definidos para dicha acción. Por último, **list\_effects** retorna los predicados correspondientes a las listas de agregados y borrados. Los efectos negados van a la lista de borrados y el resto a la de agregados.

Finalmente, el control de la traducción vuelve a **parser\_ext**. En este punto, el traductor permite procesar las estructuras genéricas definidas para los problemas y los dominios con el objetivo principal de generar la notación STRIPS adecuada para el planificador *target*. La manipulación de esta notación permite el reuso del traductor sobre diferentes planificadores que utilicen una representación similar a STRIPS como lenguaje de representación.

Luego de analizar los fragmentos más importantes de la implementación del traductor, en la próxima sección, vamos a mostrar cómo generar la especificación PDDL de entrada en Ciao Prolog, cómo invocar el traductor y algunos ejemplos de planificación utilizando el algoritmo POP y distintas instancias del “Mundo de Bloques”.

## 6.4. Demostración del Traductor

Sea el estado inicial para una instancia del “Mundo de Bloques” tal como indica la figura 6.3, y la meta, como se muestra en la figura 6.4.

En primer lugar, creamos un archivo llamado **bkw.pl** y definimos el problema y el dominio para esta instancia del “Mundo de Bloques” como indica la próxima figura 6.5. En esta definición incluimos el paquete traductor mediante la cláusula Prolog:

```
:- use_package(pddltostrips.pl).
```

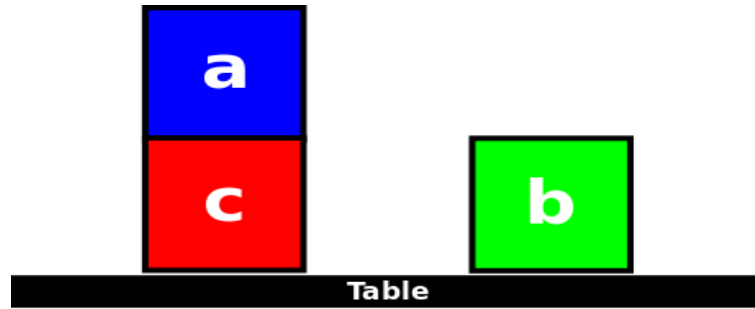


Figura 6.3: Estado Inicial del “Mundo de Bloques”

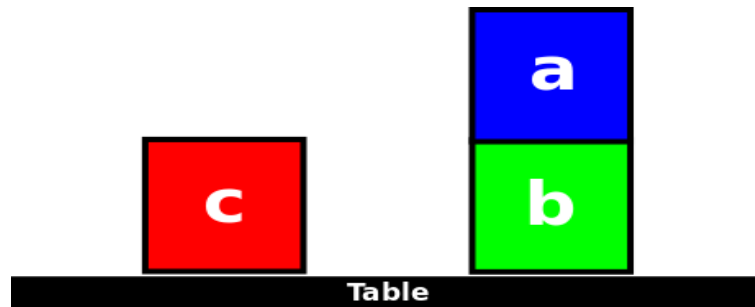


Figura 6.4: Estado Final del “Mundo de Bloques”

```

File Edit Options Buffers Tools CiaoSys CiaoDbg CiaoPP LPdoc CiaoOpts CiaoHelp Help

:- module(bkw,_,_).
:- use_package(library(show_trans)).
:- use_package('/home/lenovook/Documents/Tesis/trunk/pddltostrips.pl').

'(define (problem pb2)
  (:domain blocksworld)
  (:objects a b c)
  (:goal (on a b))
  (:init (ontable c) (ontable b) (on a c) (clear a) (clear b) (armempty)))
)',

'(define (domain blocksworld)
  (:requirements :strips :equality)
  (:predicates (clear ?x)
               (ontable ?x)
               (armempty)
               (holding ?x)
               (on ?x ?y)))

(:action pickup
  :parameters (?ob)
  :precondition (and (clear ?ob) (armempty))
  :effect (and (holding ?ob) (not (clear ?ob)) (not (armempty))))

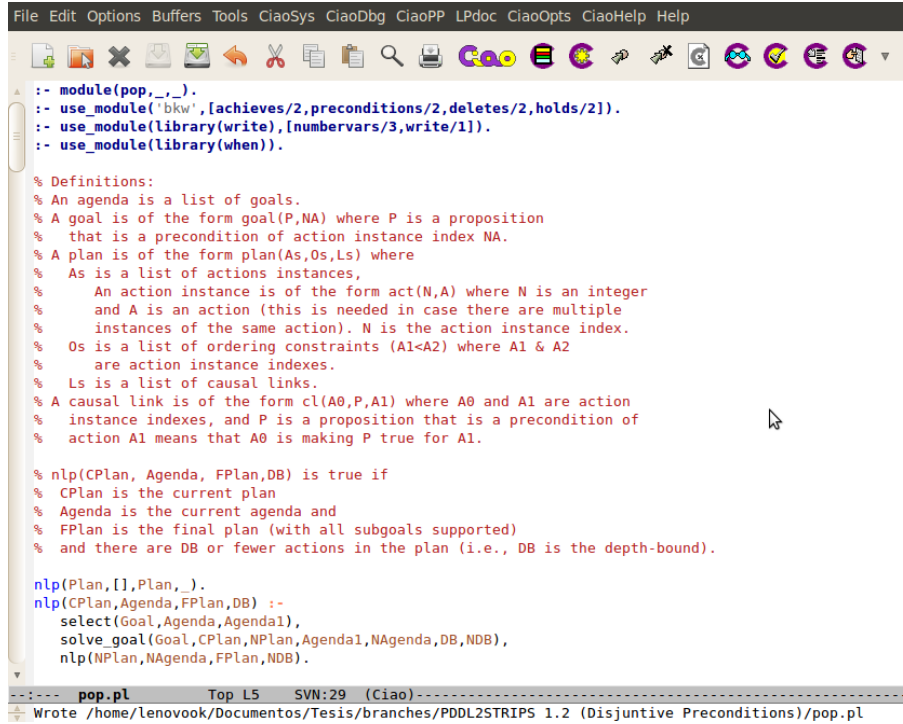
(:action stack
  :parameters (?ob ?underob)
  :precondition (and (clear ?underob) (holding ?ob))
  :effect (and (clear ?ob) (on ?ob ?underob) (armempty)
              (not (clear ?underob)) (not (holding ?ob))))
)',

```

Figura 6.5: Especificación PDDL en Ciao Prolog

Luego, desde el planificador (en este caso el POP), referenciamos la especificación PDDL anterior que será utilizada por el algoritmo planificador. Incluimos esta especificación usando la siguiente cláusula, como indica la figura 6.6.

```
:- use_module('bkw',[achieves/2,preconditions/2,deletes/2,holds/2]).
```



```

File Edit Options Buffers Tools CiaoSys CiaoDbg CiaoPP LPdoc CiaoOpts CiaoHelp Help

:- module(pop, []).
:- use_module('bkw', [achieves/2, preconditions/2, deletes/2, holds/2]).
:- use_module(library(write), [numbervars/3, write/1]).
:- use_module(library(when)).

% Definitions:
% An agenda is a list of goals.
% A goal is of the form goal(P,NA) where P is a proposition
% that is a precondition of action instance index NA.
% A plan is of the form plan(As,Os,Ls) where
% As is a list of actions instances,
%   An action instance is of the form act(N,A) where N is an integer
%   and A is an action (this is needed in case there are multiple
%   instances of the same action). N is the action instance index.
% Os is a list of ordering constraints (A1<A2) where A1 & A2
%   are action instance indexes.
% Ls is a list of causal links.
% A causal link is of the form cl(A0,P,A1) where A0 and A1 are action
% instance indexes, and P is a proposition that is a precondition of
% action A1 means that A0 is making P true for A1.

% nlp(CPlan, Agenda, FPlan,DB) is true if
% CPlan is the current plan
% Agenda is the current agenda and
% FPlan is the final plan (with all subgoals supported)
% and there are DB or fewer actions in the plan (i.e., DB is the depth-bound).

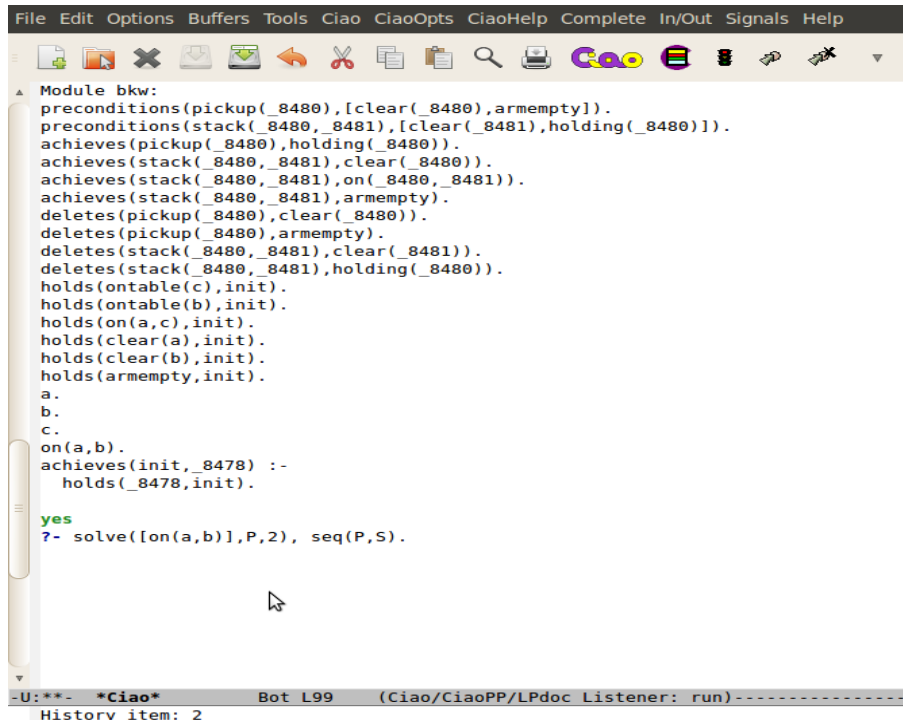
nlp(Plan,[],Plan,_).
nlp(CPlan,Agenda,FPlan,DB) :-
    select(Goal,Agenda,Agenda1),
    solve_goal(Goal,CPlan,NPlan,Agenda1,NAgenda,DB,NDB),
    nlp(NPlan,NAgenda,FPlan,NDB).

--:-- pop.pl Top L5 SVN:29 (Ciao)-----
Wrote /home/lenovook/Documents/Tesis/branches/PDDL2STRIPS 1.2 (Disjunctive Preconditions)/pop.pl

```

Figura 6.6: Planificador POP en Ciao Prolog

Compilamos y consultamos al planificador con la siguiente *query*: `solve([on(a,b)],P,2), seq(P,S)`. para obtener un plan. Notar que la figura 6.7 también muestra la salida del traductor con los STRIPS equivalentes a la especificación PDDL de entrada.



```

File Edit Options Buffers Tools Ciao CiaoOpts CiaoHelp Complete In/Out Signals Help

Module bkw:
preconditions(pickup(_8480),[clear(_8480),armempty]).
preconditions(stack(_8480,_8481),[clear(_8481),holding(_8480)]).
achieves(pickup(_8480),holding(_8480)).
achieves(stack(_8480,_8481),clear(_8480)).
achieves(stack(_8480,_8481),on(_8480,_8481)).
achieves(stack(_8480,_8481),armempty).
deletes(pickup(_8480),clear(_8480)).
deletes(pickup(_8480),armempty).
deletes(stack(_8480,_8481),clear(_8481)).
deletes(stack(_8480,_8481),holding(_8480)).
holds(ontable(c),init).
holds(ontable(b),init).
holds(on(a,c),init).
holds(clear(a),init).
holds(clear(b),init).
holds(armempty,init).
a.
b.
c.
on(a,b).
achieves(init,_8478) :-
    holds(_8478,init).

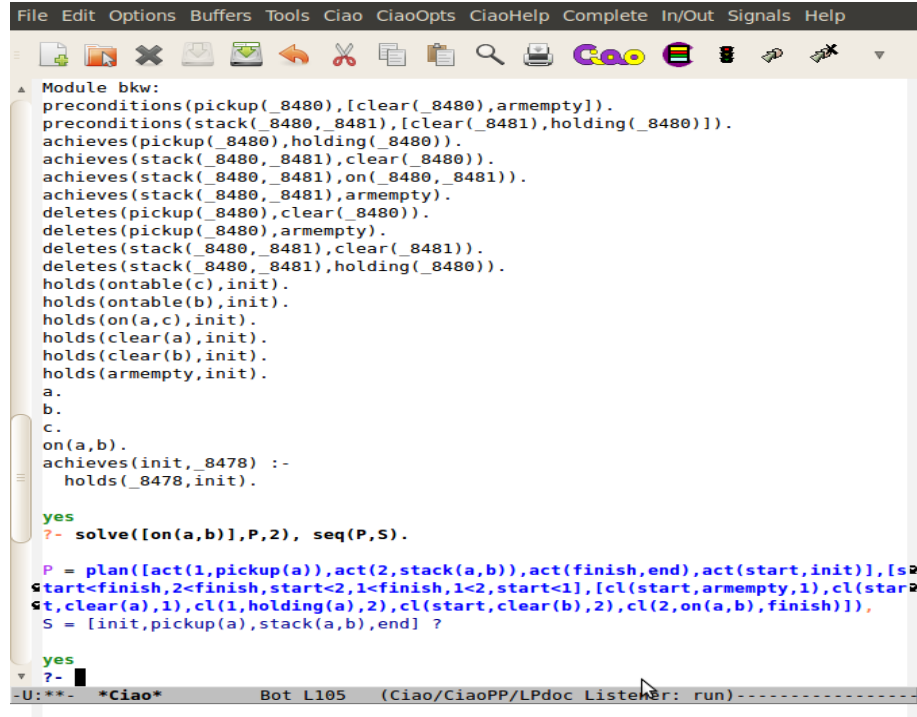
yes
?- solve([on(a,b)],P,2), seq(P,S).

-U:*** *Ciao* Bot L99 (Ciao/CiaoPP/LPdoc Listener: run)-----
History item: 2

```

Figura 6.7: Consulta para el planificador POP

Por último, el planificador retorna el plan deseado que incluye las siguientes acciones, como también ilustra la figura 6.8: `init`, `pickup(a)`, `stack(a,b)`, `end`.



```

File Edit Options Buffers Tools Ciao CiaoOpts CiaoHelp Complete In/Out Signals Help
Module bkw:
preconditions(pickup(_8480),[clear(_8480),armempty]).
preconditions(stack(_8480,_8481),[clear(_8481),holding(_8480)]).
achieves(pickup(_8480),holding(_8480)).
achieves(stack(_8480,_8481),clear(_8480)).
achieves(stack(_8480,_8481),on(_8480,_8481)).
achieves(stack(_8480,_8481),armempty).
deletes(pickup(_8480),clear(_8480)).
deletes(pickup(_8480),armempty).
deletes(stack(_8480,_8481),clear(_8481)).
deletes(stack(_8480,_8481),holding(_8480)).
holds(ontable(c),init).
holds(ontable(b),init).
holds(on(a,c),init).
holds(clear(a),init).
holds(clear(b),init).
holds(armempty,init).
a.
b.
c.
on(a,b).
achieves(init,_8478) :-
    holds(_8478,init).
yes
?- solve([on(a,b)],P,2), seq(P,S).
P = plan([act(1,pickup(a)),act(2,stack(a,b)),act(finish,end),act(start,init)],[start<finish,2<finish,start<2,1<finish,1<2,start<1],[cl(start,armempty,1),cl(start,clear(a),1),cl(1,holding(a),2),cl(start,clear(b),2),cl(2,on(a,b),finish)]),
S = [init,pickup(a),stack(a,b),end] ?
yes
?-
-U:***- *Ciao* Bot L105 (Ciao/CiaoPP/LPdoc Listener: run)-----

```

Figura 6.8: Resultado obtenido de la planificación

En este capítulo hemos analizado el traductor propuesto, definimos su arquitectura y estudiamos el código de la implementación. También vimos cómo invocar este paquete desde Ciao Prolog y cómo el traductor es integrado a un planificador para poder obtener un plan dada una representación inicial en PDDL.

En el próximo capítulo abordamos las conclusiones de esta tesis, sus resultados, contribución y proponemos algunos trabajos futuros.





## Capítulo 7

# Conclusiones

El trabajo principal de esta tesis estuvo motivado por la necesidad de dotar al Framework de Planificación Continua de un módulo traductor para el lenguaje PDDL. Este objetivo principal ha ido evolucionando durante el transcurso de este trabajo y, en consecuencia, nos ha permitido también concluir una serie de resultados teóricos y prácticos complementarios a la implementación propuesta.

La investigación realizada nos ha posibilitado implementar un traductor para un subconjunto de PDDL y, de esta manera, facultar a un agente para que pueda percibir y actuar mediante especificaciones en este lenguaje. Con este aporte, dicho agente puede disponer de un conjunto de acciones con el nivel de expresividad y abstracción necesario para operar en ambientes más complejos. Bajo este novedoso enfoque, es esperable poder abordar problemas de planificación sobre cualquier dominio real que requiera la intervención de agentes inteligentes.

La presente tesis comenzó, en el capítulo 2, con un análisis del lenguaje PDDL y sus características principales. En el capítulo siguiente, se presentó el Framework de Planificación Continua desarrollado en [36]. También se estudió el algoritmo de planificación continua implementado en dicho framework y se analizó su lenguaje de representación. Luego, en el capítulo 4, se estudiaron algunas implementaciones existentes y se remarcaron las diferencias con respecto a nuestra propuesta. En el capítulo 5 se definió el marco teórico subyacente introduciendo los conceptos de esquemas de compilación y compilabilidad. Además, se definieron y ejemplificaron algunas variantes PDDL que se mapearon a STRIPS mediante los esquemas de compilación correspondientes. Por último, en el capítulo 6, se analizó la solución propuesta y se trabajó sobre un ejemplo real a fin de ilustrar cómo escribir PDDL en Ciao Prolog y cómo planificar con estas especificaciones.

### 7.1. Resultados y Contribuciones

A continuación, resumimos los principales resultados y contribuciones de esta tesis.

- Se formalizó y demostró una nueva variante del lenguaje STRIPS llamada  $\text{STRIPS}_u$ . Esta variante surge de adicionar cuantificación universal a la definición de precondiciones para las acciones del dominio de planificación.
- Se enunciaron las demás variantes de STRIPS:  $\text{STRIPS}_C$ ,  $\text{STRIPS}_D$  y  $\text{STRIPS}_L$  y se definieron notaciones equivalentes en PDDL que se denotaron  $\text{PDDL}_C$ ,  $\text{PDDL}_D$  y  $\text{PDDL}_L$ , respectivamente. Estas variantes de PDDL conforman el lenguaje fuente del traductor implementado.
- La abstracción lograda, mediante las variantes de PDDL, posibilita la especificación de problemas de planificación complejos.

- El producto final consta de un módulo traductor del lenguaje fuente definido hacia un lenguaje destino. La implementación ofrece una representación genérica que permite adaptar este módulo a otros planificadores cuyo lenguaje de representación tiene una estructura similar a STRIPS.
- En particular, la integración del módulo traductor al Framework, presentado por Moya en [36], permite redefinir el sistema de creencias del agente de planificación continua. Esta nueva arquitectura posibilita la definición, en PDDL, de las percepciones y las acciones del agente.
- El traductor es una expansión sintáctica para Ciao Prolog. Esta novedosa extensión genera una representación STRIPS, con sintaxis Prolog, equivalente a una entrada especificada en un subconjunto del lenguaje PDDL. Hasta el momento de presentación de esta tesis, el sistema Ciao no ofrece ningún soporte para la manipulación del lenguaje PDDL.

## 7.2. Trabajo Futuro

Esta primera etapa de la investigación abarca un subconjunto acotado del lenguaje PDDL. Se espera poder ampliar el lenguaje fuente de este traductor incluyendo otros requerimientos y definiendo los esquemas de compilación asociados. De la misma manera, se espera poder realizar un análisis más exhaustivo de la complejidad de la implementación de manera tal que las traducciones propuestas tengan el menor impacto posible en la performance de los planificadores.

Uno de los aspectos pendientes es la implementación de una interface para el Framework de Planificación Continua. Se espera poder aplicarlo a dominios reales y, por lo tanto, es necesario realizar traducciones, en tiempo real, de las percepciones del agente.

Finalmente, planteamos dos desafíos. Por un lado, con el objetivo de mejorar el traductor, proponemos la aplicación de otros conceptos de Compiladores e Intérpretes, como la manipulación y recuperación de errores [1]. Por último, proponemos combinar el traductor con otros planificadores implementados en Ciao Prolog y realizar comparaciones de performance entre ellos.

# Bibliografía

- [1] Aho, A.V. and Sethi, R. and Ullman, J.D. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2007.
- [2] Allen, J. and Lehrer, N. Knowledge Representation Specification Language (KRSL). Technical report, DARPA Rome Laboratory Planning and Scheduling, 1992.
- [3] Barrett, A. UCPOP: User's Manual (Version 4.0). Technical report, Department of Computer Science and Engineering, University of Washington, 1995.
- [4] Blum, A. and Furst, M. Graphplan. Website, 1995. <http://www.cs.cmu.edu/~avrim/graphplan.html>. Disponible en Septiembre de 2012.
- [5] Bovet, J. ANTLRWorks: The ANTLR GUI Development Environment. Website, 2010. <http://www.antlr.org/works/index.html>. Disponible en Septiembre de 2012.
- [6] Braun, G., Moya, M. and Parra, G. Sistemas Multiagentes en Ambientes Dinámicos: Planificación Continua mediante PDDL. *XIII Workshop de Investigación en Ciencias de la Computación*, 2011.
- [7] Bueno, F., Cabeza, D., Carro, M., et al. The Ciao Prolog system. Reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997.
- [8] Cabeza, D. and Hermenegildo, M. A New Module System for Prolog. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 131–148, London, UK, UK, 2000. Springer-Verlag.
- [9] Carbonell, J., Blythe, J., Etzioni, O., et al. PRODIGY 4.0: The Manual and Tutorial. Technical report, 1992.
- [10] CEA, CNRS and INRIA. CEA CNRS INRIA Logiciel Libre (CeCILL). Website, 2006. <http://www.cecill.info/index.en.html>. Disponible en Septiembre de 2012.
- [11] Cecchi, L., Vaucheret, C., Moya, M., et al. Un enfoque basado en competencias de sistemas multiagentes para la enseñanza de Inteligencia Artificial. In *IV Congreso de Tecnología en Educación y Educación en Tecnología*, pages 372 – 381, La Plata, Julio 2009. Universidad Nacional de La Plata.
- [12] Chomsky, N. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.
- [13] Deransart, P. and Cervoni, L. and Ed-Dbali, A. *Prolog: the standard: reference manual*. Springer-Verlag, London, UK, UK, 1996.
- [14] Edelkamp, S. and Hoffmann, J. PDDL2.2 The Language for the Classical Part of the 4th International Planning Competition. Technical report, ALbert Ludwigs University Institute for Informatik, Freiburg, Germany, 2003.

- [15] Fikes, R. and Nilsson, N. STRIPS: A new approach to the application of theorem proving to problem solving. *J. Artificial Intelligence*, 2:189–208, 1971.
- [16] Fox, M. and Long, D. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res. (JAIR)*, pages 61–124, 2003.
- [17] Fox, M. and Long, D. Modelling Mixed Discrete-Continuous Domains for Planning. *Journal of Artificial Intelligence Research*, page 235â297, 2011.
- [18] Inc. Free Software Foundation. LGPL. Website, 2007. <http://www.gnu.org/licenses/lgpl.html>. Disponible en Septiembre de 2012.
- [19] Free Software Foundation, Inc. GPL. Website, 2007. <http://www.gnu.org/copyleft/gpl.html>. Disponible en Septiembre de 2012.
- [20] Gazen, B. and Knoblock, C. Combining the expressivity of UCPOP with the efficiency of Graphplan. In *PROC. 4TH EUROPEAN CONFERENCE ON PLANNING*, pages 221–233, 1997.
- [21] Gerevini, A. and Long, D. BNF Description of PDDL3.0. Technical report, 2005.
- [22] Gerevini, A. and Long, D. Preferences and soft constraints in PDDL3. Technical report, Department of Electronics for Automation, University of Brescia, 2006.
- [23] Hanks, S. and Firby, J. Issues in architectures for planning and execution. In *Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 59–70, Scheduling and Control, San Diego, CA, November 1990.
- [24] International Organization for Standarization. ISO/IEC 14977:1996. Website, 1996. [http://www.iso.org/iso/iso\\_catalogue/catalogue\\_tc/catalogue\\_detail.htm?csnumber=26153](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=26153). Disponible en Septiembre de 2012.
- [25] Kogan, P., Moya, M., Torres, G., et al. RAKIDUAM: Un Equipo de Fútbol con Licencia GNU General Public License. In *V Campeonato Argentino de Fútbol con Robots*, 2007.
- [26] Kogan, P., Yañez, J., Campagnon, C. et al. Aspectos de diseño y de implementación del equipo de fútbol con robots RAKIDUAM. In Interamericana U. A., editor, *Proceedings del Campeonato Argentino de Fútbol con Robots (CAFR)*, 2006.
- [27] Kovacs, D.L. BNF definition of PDDL 3.1. Technical report, 2011.
- [28] Kutluhan, E. *Hierarchical task network planning: formalization, analysis, and implementation*. PhD thesis, 1996.
- [29] Kutluhan, E., Hendler, J. and Nau, D.S. UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. In *Proceedings of the 2nd International Conference on AI Planning Systems*, pages 249–254, 1994.
- [30] Mcallester, D. and Rosenblitt, D. Systematic Nonlinear Planning. In *In Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639, 1991.
- [31] McCarthy, J. and Hayes, P. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, pages 463–502. 1969.
- [32] Mcdermott, D. A Heuristic Estimator for Means-Ends Analysis in Planning. In *In Proc. Third Int. Conf. on AI Planning Systems (AIPS-96)*, pages 142–149, 1996.

- [33] McDermott, D. PDDL, the Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, 1998.
- [34] McDermott, D. OPT Manual. Technical report, 2005.
- [35] McDermott, D., Ghallab, M., Howe, A., et al. PDDL The Planning Domain Definition Language Version 1.2. Technical report, Yale Center for Computational Vision and Control, 1998.
- [36] Moya, M. Control de Agentes Basado en Planificación Continua. Tesis de Licenciatura en Ciencias de la Computación, Universidad Nacional del Comahue, 2009.
- [37] Moya, M. and Vaucheret, C. Planificador Continuo como Controlador de Agentes Robots. In *X Workshop de Investigadores en Ciencias de la Computación*, General Pico, La Pampa, Argentina, 2008. Universidad Nacional de la Pampa.
- [38] Moya, M. and Vaucheret, C. Un Planificador Continuo Concurrente para Agentes Robots. In *V Workshop de Inteligencia Artificial aplicada a la Robótica Móvil*, Buenos Aires 1400, Neuquén, Argentina, 2008. Universidad Nacional del Comahue.
- [39] Moya, M. and Vaucheret, C. Agentes deliberativos basados en planificación continua. In *X Workshop Agentes y Sistemas Inteligentes (WASI)*, Martiarena esquina Italia - S.S. de Jujuy, Octubre 2009. Universidad Nacional de Jujuy - Facultad de Ingeniería.
- [40] Nebel, B. On the Compilability and Expressive Power of Propositional Planning Formalisms. *Journal of Artificial Intelligence Research*, 12:271–315, 2000.
- [41] Nebel, B. Logic-based artificial intelligence. chapter On the expressive power of planning formalisms: Conditional effects and Boolean preconditions in the STRIPS formalism, pages 469–488. Kluwer Academic Publishers, Norwell, MA, USA, 2001.
- [42] Parr, T. Antlr v3. Website, 1989. <http://www.antlr.org>. Disponible en Septiembre de 2012.
- [43] Parr, T. The BSD License. Website, 2010. <http://www.antlr.org/license.html>. Disponible en Septiembre de 2012.
- [44] Pednault, E. ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the First International Conference on Principles of knowledge representation and reasoning*, pages 324–332, 1989.
- [45] Poole, D. and Mackworth, A. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, first edition, 2010.
- [46] Rao, A. and Georgeff, M. Modeling Rational Agents within a BDI-Architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.
- [47] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Inteligence. Prentice Hall, second edition, 2003.
- [48] Russell, S. and Norvig, P. *Artificial Intelligence: A modern approach*. Prentice Hall, New Jersey, third edition, 2009.
- [49] Saigol, Z. PDDL grammar for ANTLR v3. Website, 2007. <http://www.antlr.org/grammar/1172246598728/Pddl.g>. Disponible en Septiembre de 2012.

- [50] Sasák, R. Planner prolog. Website, 2009. <http://artax.karlin.mff.cuni.cz/~sasar5am/pddl>. Disponible en Septiembre de 2012.
- [51] Steele, Jr. and Guy, L. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.
- [52] Sterling, L. and Shapiro, E. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, second edition, 1997.
- [53] Trevisani, D., Braun, G., Moya, M., et al. RAKIDUAM: sistema multiagentes para el fútbol de robots. In *VII Campeonato Argentino de Fútbol con Robots*, Cabildo 134, Morón, Buenos Aires, 2009. Facultad de Informática, Ciencia de la Comunicación y Técnicas Especiales de la Universidad de Morón.
- [54] World Wide Web Consortium (W3C). RDF Vocabulary Description Language 1.0: RDF Schema. Website, 2004. <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>. Disponible en Septiembre de 2012.
- [55] World Wide Web Consortium (W3C). RDF 1.1 Concepts and Abstract Syntax. Website, 2011. <http://www.w3.org/TR/2011/WD-rdf11-concepts-20110830/>. Disponible en Septiembre de 2012.
- [56] Warren, D.H.D. An abstract Prolog instruction set. *SRI International*, 1983.
- [57] Weld, D.S. Recent Advances in AI Planning. *AI MAGAZINE*, 20:93–123, 1999.
- [58] Wielemaker, J. Swi-prolog home page. Website, 1990. <http://www.swi-prolog.org/>. Disponible en Septiembre de 2012.
- [59] Wilkins, D. E. *Using the SIPE-2 Planning System: A Manual for Version 4.17*. SRI International Artificial Intelligence Center, 1997.