Anexo I: El lenguaje de descripción de dominios HTN-PDDL

1. Introducción

En este documento se describe la extensión del lenguaje PDDL para la descripción de dominios jerárquicos que se denomina HTN-PDDL. HTN-PDDL es un lenguaje con soporte temporal que permite controlar y dirigir la búsqueda mediante el uso de heurísticas.

El lenguaje HTN-PDDL está basado en el estándar *de facto* PDDL [pddl2-1] [pddl2.2] [pddl-ext] [PDDL 3.0] (*Planing Domain Description Language)* utilizado en la *International Planning Competition (IPC),* que es una competición que se realiza de forma bianual y de forma conjunta con la Conferencia Internacional de Planificación y Scheduling *(ICAPS International Conference on Automated Planning and Scheduling)* y que trata de comparar los planificadores más modernos enfrentándolos a una batería de problemas en diversos dominios descritos en PDDL.

La versión del lenguaje a partir de la cual se extiende HTN-PDDL es la 2.2 en su nivel 3, versiones posteriores no están todavía soportadas en el lenguaje.

Las principales características del lenguaje PDDL son las siguientes:

- Las acciones o operadores primitivos tienen el estilo usado por el planificador STRIPS [strips] [strips2].
- Se pueden definir efectos condicionales.
- Dispone de cuantificadores universales.
- Permite la descripción de axiomas sencillos.
- Se pueden describir restricciones de integridad.
- En la primera versión del lenguaje PDDL, la 1.2, existía la posibilidad de definir acciones jerárquicas. El formalismo introducido era muy complejo y eso hizo que ninguno de los planificadores más conocidos hiciera uso de esta característica. Esta capacidad del lenguaje además no fue ampliada ni modificada en versiones posteriores por lo que ha quedado abandonada.
- Se permiten utilizar distintas extensiones del lenguaje, que los planificadores pueden o no utilizar dependiendo de la potencia expresiva de estos.

El lenguaje HTN-PDDL además aporta las siguientes características:

- Aumenta la expresividad temporal de PDDL 2.2 nivel 3.
- Añade un soporte para las acciones jerárquicas, con una expresividad distinta y más clara que la usada en PDDL 2.1.
- Añade la posibilidad de invocar a lenguajes de scripting, como Python, para hacer llamadas externas o realizar ciertos cálculos, durante la fase de planificación.

Aunque HTN-PDDL es un superconjunto de PDDL se basa en un formalismo distinto al utilizado por los planificadores “planos” que son los que compiten en la IPC. Por lo tanto probablemente ningún planificador jerárquico pueda resolver los dominios y los problemas usados en la IPC sin introducir ciertos cambios sobre los mismos, a pesar de que un parser de HTN-PDDL pueda parsear los dominios escritos en PDDL. De hecho los objetivos en HTN-PDDL se describen de una manera distinta que en PDDL.

El lenguaje HTN-PDDL está en constante desarrollo y evolución, por lo tanto el lenguaje descrito en este documento puede ser ampliado en futuras versiones, aunque siempre se tratará de mantener la compatibilidad hacia atrás. Esta documento específica las características del lenguaje HTN-PDDL en su **versión 1.1**.

2. Notación

La notación que se utilizará para describir las reglas será EBNF (*Extended BNF).* Se utilizarán los siguientes formalismos para describir las reglas sintácticas del lenguaje HTN-PDDL

- Las reglas tienen la siguiente forma <elemento>: <expansion>
- Se usarán ángulos para delimitar los nombres de los elementos sintácticos.
- Se usan los corchetes [ ] para delimitar aquellos elementos que son opcionales.
- Se usan paréntesis para delimitar bloques dentro del lenguaje HTN-PDDL o para indicar parámetros de una regla en concreto. Los paréntesis forman parte del lenguaje.
- Como en BNF el asterisco * denota 0 o más veces y el símbolo + denota 1 o más veces.

3. Comentarios

Los comentarios en HTN-PDDL empiezan con un “;” y terminan con un retorno de carro.

Ejemplo 1.
```
; Esto es un comentario escrito en HTN-PDDL
```

4. Términos y átomos

Los términos pueden ser constantes (objetos del dominio) o variables.

```
<symbol>:                                   <name>

<term>:                                     <constant>
                                                        | <variable>
                                                        | <number>

<variable>:                                 ?<name>

<constant>:                                 <symbol>

<predicate>:                          (<symbol> <term>*)

<typed-predicate>:          (<symbol> <typed-variable>*)

<typed-variable>:                 <variable> - <type>
<typed-variable>:                 <variable>

<type>:                                     <symbol>
                                                        | (either <type>*)
```

Un símbolo es una cadena alfanumérica que comienza con una letra y que puede contener letras, dígitos, guiones y subrayados. Los caracteres acentuados y la ñ del castellano también es aceptada. HTN-PDDL no es sensible a las mayúsculas, es decir, el símbolo “hola” es equivalente al símbolo “HoLa”.

Una constante se identifica con un símbolo y representa un objeto del dominio. Una variable se identifica con un símbolo que comienza por un carácter de cierre de interrogación.

Al igual que en el PDDL estándar tanto símbolos, como variables, como predicados, pueden tener un tipo asociado. Se permite definir una jerarquía de tipos que efectúa las relaciones de herencia de propiedades entre los objetos del dominio.

Ejemplo 2.

```
; objetos
ferrari - automovil
; predicados (atributos de un objeto)
; instanciados y no instanciados
(velocidad_maxima ?vehiculo - automovil ?x - number)
(velocidad_maxima ferrari 220)
```

En HTN-PDDL si no se especifica expresamente un tipo se supone que hereda por defecto del tipo especial “object”.

5. Dominios

Para que un planificador disponga de toda la información para operar con el lenguaje HTN-PDDL éste recibe dos ficheros de entrada. Un fichero con la especificación del dominio, que es la descripción del mundo (tareas jerárquicas, operadores primitivos, objetos, atributos y tipos) y otro fichero con el problema que queremos resolver.

La descripción de un dominio en HTN-PDDL tiene la siguiente sintaxis:

```
<domain>:                    (define (domain <symbol>)
                                     [<require-def>]
                                     [<customization-def>]
                                     [<types-def>]
                                     [<constants-def>]
                                     [<predicates-def>]
                                     [<functions-def>]
                                     [<structure-def>*])
```

En HTN-PDDL Un dominio consta de siete bloques todas excepto el bloque de personalización “customization” ya existen en PDDL 2.2. La sección de personalización será analizada en detalle más adelante. Se empezará estudiando la sintaxis de la declaración de tipos.

6. Declaración de tipos

```
<types_def>:                 (:types <type-def>*)
<type-def>:                          <symbol>+ - <type>
<type-def>:                          <symbol>+
```

Se usa una lista de tipos para declarar todos los tipos que heredan (son subtipo) de un tipo determinado. Los tipos aparecen precedidos de un guión (-), todo tipo que preceda al guión (subtipo) queda declarado como del tipo que aparece tras el guión (supertipo). Si no se especifica un supertipo explícitamente un tipo siempre se supone por defecto que hereda del tipo especial “object”.

Se permite realizar herencia múltiple mediante el uso de clausula either.

Ejemplo 3.

```
; declaración de tipos
(:types
   persona
   persona_anorexica – persona
   persona_guapa – persona
   ; herencia múltiple
```

```
   modelo - (either persona_anorexica persona_guapa)
)
```

7. Declaración de constantes

```
<constants-def>:          (:constants <constant-def>*)
<constant-def>:           <symbol>+ - <type>
<constang-def>:           <symbol>+
```

Las constantes se representan mediante símbolos y modelan los objetos del dominio. Se permite que las constantes sean declaradas como pertenecientes a una serie de tipos determinados, como se puede apreciar en el siguiente ejemplo:

Ejemplo 4.

```
; declaración de constantes
(:types
   paz_vega - actor
   ana_belen - (either actor cantante)
)
```

8. Declaración de predicados

```
<predicates-def>:         (:predicates <typed-predicate>*)
```

Los predicados son usados para definir atributos de los objetos. También se usan para definir relaciones de cualquier aridad entre los objetos. En planificación jerárquica HTN también se pueden usar ciertos predicados que no forman parte del estado como herramientas para generar heurísticas de control del procesamiento del planificador.

Ejemplo 5.

```
; declaración de predicados
(:predicates
   (altura ?p - persona ?x -number)
   (casado ?amargado_x -persona ?amargado_y - persona)
   (jefe_de ?x -persona ?mala - persona)
   (director ?la_peor - persona)
)
```

9. Declaración de funciones

```
<functions-def>:          (:functions <function-def>*)
<function-def>:           (<typed-predicate>*) [- <type>]
                                                    [<python-code>]
<python-code>:                  {<python_script>}
```

Las funciones utilizadas en PDDL no se corresponden totalmente con el concepto que tenemos de función, por eso en PDDL se denominan "*fluents literals*". Un "*fluent*" es un predicado que almacena un determinado valor, que en la mayoría de los casos suele ser un número, de forma que el lenguaje permite realizar operaciones aritméticas y de comparación entre estos *fluents* como veremos más adelante.

En HTN-PDDL se permite asociar un literal a una verdadera función descrita en un lenguaje de scripting. El lenguaje de script más usado en los dominios descritos en HTN-PDDL es Python (http://www.python.org).

Las convenciones a usar para el paso y retorno de parámetros al script están todavía en una fase

experimental.

Actualmente sólo se permite que el script devuelva (mediante la cláusula return) un número o un símbolo (objeto constante) ya predefinido en el dominio.

El script debe poder usar las variables PDDL declaradas como argumento de la función, en los cálculos internos. El parser de HTN-PDDL debe hacer una sustitución de estas variables por los valores que efectivamente toman en tiempo de planificación.

De esta forma si se consigue tener funciones reales, que realizan cálculos o llamadas externas en pleno proceso de planificación.

Ejemplo 6.

```
; declaración de funciones y llamadas a Python
(: functions
; un predicado python
(distance ?x1 ?y1 ?x2 ?y2)
{
 import math
 return math.sqrt ( (?x2 - ?x1) * (?x2 -?x1) + (?y2 - ?y1) * (?y2 - ?y1))
}
; un predicado pddl
(distance ?x ?y -location) - number
)
```

10. Sección principal

La sección principal recoge la definición de operadores primitivos, tareas compuestas y axiomas, que el planificador puede utilizar para resolver el problema de planificación.

```
<structure-def>:    <action-def>
                    | <durative-action-def>
                    | <derived-def>
                    | <htn-task-def>

<derived-def>:      (:derived <derived-body>)
<derived-def>:      (:derived <atomic-formula(variable)>
                                                    <goal-def>)
                                                    | (:derived <atomic-formula(variable)
                                                    {<python-code>}
```

Las acciones y las tareas se estudiarán con mas detalle en las siguientes secciones.

Los axiomas sirven para definir nuevos predicados a partir del uso de predicados ya existentes en una cláusula más compleja.

Ejemplo 7.

```
; declaración de axiomas
(:derived (ganga ?x – producto)
   (and (bueno ?x) (bonito ?x) (barato ?x))
)
```

11. Operadores primitivos

Los operadores primitivos (acciones) en HTN-PDDL son muy similares a los de PDDL, con algunas pequeñas ampliaciones. De hecho se pueden usar acciones descritas en PDDL directamente en HTN-PDDL.

```
<action-def>:                                   (:action <name>
                                                                    :parameters (<typed-variable>*)
                                                                    [<metags>]
                                                                    [<preconditions-def>]
                                                                    [<effect-def>])

<preconditions-def>:                    :precondition <goal-def>

<effect-def>:                                   :effect <effect>

<metatags>:                                             :meta (<tag>*)
```

<tag>: (:tag <name> <text>)

(*) Nota: <*text*> es una cadena de caracteres delimitada por comillas dobles (“”).

La principal diferencia la constituye la posibilidad de usar metatags, cuyo uso se explicará en las siguientes secciones.

11.1. *Parámetros*

Los parámetros son una lista de variables que el planificador instancia durante la fase de planificación, y que son los argumentos utilizados dentro del cuerpo del operador.

11.2. *Metatags*

Los *metatags* son una extensión del lenguaje que actualmente está en fase experimental, por lo que puede cambiar en futuras versiones. El concepto subyacente a los metatags es poder incluir información extra en el dominio, que aunque no sea directamente utilizable por el planificador, pueda ser utilizada por otros módulos, o en un análisis posterior del plan resultante.

Cuando el planificador termina puede incluir esta información extra en plan resultante. En concreto, si se supone que el planificador obtiene un plan como un documento XML, los metatags podrían ser incluidos como tags XML, que podrían ser parseados y analizados con posterioridad.

Ejemplo 8.

```
; Ejemplo de acción durativa (temporizada) con
; metatags
(:durative-action move
 :parameters (?obj - object ?loc -location)
 :meta(
   (:tag prettyprint "?start ?end > mover ?obj desde ?lx hasta ?loc. Llegada a las ?end (?duration min)")
   (:tag cost "low")
 )
 :duration (= ?duration 22)
 :condition((and
    (location ?obj ?lx)
    (not (same ?lx ?loc))
 ))
 :effect((and
    (not (location ?obj ?lx))
    (location ?obj ?loc)
 ))
)
```

La salida XML que se obtendría podría ser similar a la siguiente:

```
<primitive name="move" id="170" indx="186" start="27/01/2006 08:11:00" end="27/01/2006 08:33:00" duration="22" start_point="64"
end_point="65">
        <meta name="prettyprint">
        27/01/2006 08:11:00 &gt; mover BRI112 desde BASE01 hasta WAITAREA. Llegada a las 27/01/2006 08:33:00 (22.000000 min)
        </meta>
        <meta name="cost">
        low
        </meta>
        <parameters>
          <parameter pos="0" name="g">
            <constant name="BRI112" id="715">
              <type name="Brigade" id="73">
              </type>
            </constant>
          </parameter>
.....
```

Existe un *metatag* que tiene un carácter especial que se llama *prettyprint.* P*rettyprint* es una cadena de texto parseable, pensada para presentar las acciones resultantes del proceso de planificación, de una forma más legible al usuario final.

prettyprint, o en general cualquier metatag, debe permitir incluir dentro de la cadena de texto variables, que son sustituidas por el valor al que estas variables son ligadas. Las variables deben incluir al menos los parámetros de la acción, aunque un planificador en concreto. puede incluir variables extras a costa de la pérdida de la generalidad del dominio.

Cuando un planificador encuentra una variable que es capaz de interpretar, por defecto no debe hacer la sustitución.

Cuando termina el proceso de planificación y las acciones resultantes son presentadas al usuario (ya sea a través de la salida estándar o a través de un fichero XML o de texto), el planificador debe usar el formato especificado en *prettyprint.*

Otra alternativa también permitida es que en lugar de usar los identificadores de las variables para hacer la sustitución, se utilice el número de argumento dentro de la lista de argumentos, precedido por el carácter "$" ($1, $2, ...), al igual que en el resultado del matching de una expresión regular en Perl.

11.3. *Precondiciones*

Las precondiciones son una lista opcional de objetivos que deben ser satisfechos en el estado del mundo justo antes de la ejecución de la acción. La descripción de objetivos es bastante expresiva, ya que se permiten expresiones en lógica de primer orden. Sino se especifican precondiciones se supone que la acción puede ejecutarse en cualquier estado del universo.

La sintaxis BNF para una condición es la siguiente:

```
<goal-def>:                    ( )
                                                    | (! <goal-def>)
                                                    | (:sortby <variable-ord>+  <goal-def>)
                                                    | (:bind <variable> <fluent-exp>)
                                                    | (:print <sgoal-def>)
                                        | (:print <text>)
                                                    | (:print <term>*)
                                        | <sgoal-def>
                                        | <timed-goal>


<sgoal-def>:                   (and <goal-def>*)
                                                    | (or  <goal-def>*)
                                                    | (not <goal-def>)
                                                    | (imply <goal-def> <goal-def>)
                                                    | (exists  (<typed-variable>* ) <goal-def>)
                                                    | (forall  (<typed-variable>* ) <goal-def>)
                          | <fluent-comp>
                          | <atomic-formula(term)>

<variable-ord>:           <variable> :asc
                                                    | <variable> :desc

<fluent-comp>:            (<binary-comp> <fluent-exp> <fluent-exp> )

<binary-comp>:            >
                                        | <
                                        | =
                                        | >=
                                        | <=
                                        | !=
```

Se han añadido algunos predicados más no incluidos en PDDL estándar como son los siguientes:

- bind: Permite asignar a una variable sin instanciar el contenido de un predicado numérico o un *fluent.*
- print: Es un predicado que siempre es cierto y permite imprimir un texto libre, el contenido de una variable, o el resultado de analizar la unificación de una condición. Este predicado es

útil a la hora de imprimir mensajes por pantalla durante el proceso de planificación y facilitar así la tarea de depuración.

- Cuando se produce una unificación el planificador puede escoger cualquiera de las posibles sustituciones de variables por valores constantes. A veces es interesante que el diseñador de dominios tenga el control sobre el orden en el que se quieren ir probando las unificaciones. Éste es precisamente el objetivo del predicado *sortby.*

Ejemplo 9.

```
; Un ejemplo de uso de precondiciones con sortby y bind
(:action recoger_pasajero
 :parameters (?p – person ?y - point)
 :precondition (and
              (esperando_taxi ?p)
              (sortby ?d :asc (and
                (libre ?t -taxi)
                (posicion ?t ?x - point)
                (bind ?d (calcula_distancia ?x ?y))
              ))
             )
 :effect (and (not (esperando_taxi ?p)) (not (libre ?t)))
)
```

Los objetivos en HTN-PDDL, no se describen en base a una condición que es necesario hacer cierta en el estado final, como ocurre en PDDL, sino como una red de tareas a descomponer. Por eso se tratarán más adelante, cuando se presenten las redes de tareas.

11.4. *Efectos*

Los efectos son una descripción completa de los cambios que la acción produce en el estado actual del universo. Los efectos pueden ser cuantificados universalmente y también se permiten efectos condicionales (dependientes del estado actual del universo), pero no todas las sentencias de la lógica de primer orden están permitidos (por ejemplo funciones de Skolem o expresiones disyuntivas). Es importante destacar que, en este sentido, el lenguaje es asimétrico, es decir, las precondiciones de las acciones son considerablemente más expresivas que sus efectos.

```
<effect>:                          ( )
                                                | (and <c-effect>*)
                                   | <c-effect>

<c-effect>:                (forall (<typed-variable>+) <effect>)
                                   | (when <goal-def> <cond-effect> )
                                   | <p-effect>
                                   | <timed-effect>


<p-effect>:                (<assign-op> <atomic-formula(term)>
                                                <fluent-exp>)
                                   | (not <af-effect>)
                                   | <af-effect>

<af-effect>:               <atomic-formula(term)>
                                                | (:maintain <atomic-formula(term)>)


<cond-effect>:     (and <p-effect>*)
                           | <p-effect>

<assign-op>:               assign
                           | scale-up
                           | scale-down
                           | increase
                                                | decrease


<fluent-exp>:              (<binary-op> <fluent-exp> <fluent-exp>)
                                                | (<unary-op> <fluent-exp>)
                           | <atomic-formula(term)>
                           | <term>
```

```
unary_op:                       -
                                | :abs
                                | :sqrt

binary_op:                      +
                        | -
                                                | *
                        | /
                        | :pow
```

A la hora de que el planificador evalúe un efecto se supone que todas las variables deben estar ligadas, bien por qué lleguen desde un parámetro, se hayan ligado al hacer una unificación en las precondiciones, o bien por que estén cuantificadas.

También se asume que como en STRIPS el valor de verdad de un predicado es persistente a lo largo del tiempo. En STRIPS por eso se define una lista de supresión para hacer falsos los predicados (retirarlos del estado, ya que cualquier predicado que no se encuentre en el estado se supone falso). En HTN-PDDL sin embargo no existe explícitamente una lista de supresión, en lugar de ello cuando se quiere hacer falso un predicado, simplemente se niega.

Ejemplo 10.
```
; hacer falso el valor de un predicado
(:action drop_item
 :parameters (?arm – robotic_arm ?obj - item)
 :precondition (holding ?arm ?obj)
 :effect ((not (holding ?arm ?obj))
))
```

Cómo es lógico ningún efecto de un operador primitivo puede alterar el valor de verdad de un predicado si no es mencionado explícitamente. Es decir si el efecto de un operador no cambia el valor de el predicado *P* el predicado *P* debe mantenerse inalterado tras la ejecución del operador.

Ejemplo 11.
```
; Algunos ejemplos autoexplicativos de efectos
; condicionales y cuantificadores universales.
(:action unload_cargo
 :parameters (?t – truck ?l - location)
 :precondition (at ?t ?l)
 :effect (forall ?x – obj
     (when (in ?obj ?t)
       (and (not (in ?obj ?t)) (in ?obj ?l))
     )
))
```

Observar que los objetivos en un efecto condicional tienen la misma sintaxis que en las precondiciones pero su función es distinta. Si un objetivo no se puede cumplir en las precondiciones el operador no puede ejecutarse, sin embargo si un objetivo no se cumple en un efecto condicional, la acción si puede ejecutarse, pero el efecto condicional no producirá cambios en el estado actual.

La clausula para proteger un predicado *maintain* pertenece a HTN-PDDL y no está soportada en el PDDL estándar. Esta cláusula sirve para evitar que otros operadores primitivos puedan alterar el predicado protegido. Si una acción trata de modificar el valor de verdad de un predicado mantenido, la aplicación de la acción fallará y el planificador hará backtracking. Para eliminar la protección simplemente hay que negarla.

Ejemplo 12.
```
; Ejemplo de cláusula maintain
(:action proteger
 :parameters (?x)
 :precondition ()
 :effect (:maintain (maintained ?x)))
```

```
(:action desproteger
 :parameters (?x)
 :precondition ()
 :effect (not (:maintain (maintained ?x))))
```

Los fluents de HTN-PDDL son iguales a los de PDDL salvo la adición de algunos operadores.

Ejemplo 13.

```
; Un ejemplo con fluents
(:action revender
 :parameters (?x -producto)
 :precondition (> (en_stock ?x) 0)
 :effect (assign (pvp ?x) (* (precio_compra ?x) 2))
)
```

12. Tareas abstractas

Una de las partes más interesantes de HTN-PDDL es la forma en como se introduce el soporte jerárquico como una capa más sobre PDDL. PDDL 1.2 tiene una sintaxis compleja para definir redes de tareas (quizá por su excesiva dependencia de la sintaxis de LISP), lo que hace que el dominio resultante sea poco legible. Uno de los objetivos principales de HTN-PDDL es que sea estructurado y fácilmente legible, sobretodo teniendo en cuenta el soporte temporal que añade un grado más de dificultad.

La notación BNF para definir redes de tareas y tareas abstractas es la siguiente:

```
<htn-task-def>:          (:task <name>
                          :parameters (<typed-variable>*)
                          [<meta-tags>]
                          <method-def>
                                                    )

<method-def>:                    <method>*
                         | (!<method>*)

<method>:                (:method <name>
                          [<meta-tags>]
                          [<preconditions-def>]
                                                     :tasks <task-network>
                                                    )

<task-network>:          <task-structure>
                         | ( )

<task-structure>:        <task-def>
                         | ! task_def
                         | \<<dur-constraints> <task-structure>+\>
                         | \<<task-structure>+\>
                         | (<dur-constraints> <task-structure>+)
                         | (<task-structure>+)
                         | [<dur-constraints> <task-structure>]
                         | [<task-structure>]


<task-def>:                      <atomic-formula(variable)>
                                          | (:achieve <goal-def>)
                                          | <inline-def>

<inline-def>:                    (:inline [<goal-def>] [<effect>])
                                                    | (:!inline [<goal-def>] [<effect>])

<atomic-formula(x)>:             (<name> x*)
```

Una red de tareas en HTN-PDDL, tiene un nombre unos parámetros y una lista de métodos de descomposición asociados. Cada método lleva asociado un nombre una precondición que debe hacerse cierta para su ejecución y una red de tareas para expandir. Las redes de tareas en HTN-PDDL son de tres tipos:

- Secuenciales: Este tipo de red de tareas se representa colocando las tareas hijas a expandir entre paréntesis. Obliga a que las tareas y sus descendientes mantengan el orden.
- Paralelas o independientes: Se representan colocándolas entre corchetes. El planificador podrá escoger cualquier orden entre las tareas y entre sus subtareas, pero se considerará que si una tarea de la red falla, la red de tareas enteras es fallida.
- Permutables o combinatorias: Se representan entre ángulos. El planificador deberá comprobar todos los órdenes entre las tareas incluyendo sus subtareas.

Esta sintaxis a la hora de definir redes de tareas hace que el código HTN-PDDL sea sencillo y fácilmente legible:

Ejemplo 14.

```
; Definición de una tarea abstracta
(:task moverse
:parameters (?quien ?destino)
  (!
    (:method en-destino
     :precondition (= (posicion ?quien) ?destino)
     :tasks ()
    )
    (:method andando
     :precondition (and (bound ?donde (posicion ?quien)) (<= (distancia ?donde ?destino) (limite_andando ?quien)))
     :tasks (ir_caminando ?quien ?destino)
    )
    (:method en_taxi
     :precondition ()
     :tasks (ir_en_taxi ?quien ?destino)
    )
  )
 )

(:task ir_en_taxi
  :parameters (?quien ?destino)
  (:method ir_en_taxi
   :precondition (= ?origen (posicion ?quien))
   :tasks (
      (esperar_taxi_en ?quien ?origen)
      (coger_taxi_a ?quien ?destino))
  )
 )
```

13. Mecanismos para el control de la búsqueda

Uno de los aspectos más interesantes de la planificación HTN es la posibilidad de intervenir en el proceso de búsqueda mediante el uso de heurísticas. HTN-PDDL introduce información de control mediante el uso de las tareas *inline* y el concepto de corte.

Las tareas *inline* son, a efectos del proceso de planificación, acciones primitivas que no tienen nombre. Tienen una precondición que debe ser cierta para su ejecución y un efecto asociado que altera el estado. Sin embargo, la semántica de las acciones *inline* es distinta para el escritor de dominios que la de las acciones normales. Este tipo de acciones se utilizan para incluir información de control de la búsqueda en el estado del planificador (que es distinto de la modelización dentro del estado del planificador del estado del universo) incrustada en la descripción de la red de tareas.

Distinguimos dos tipos de acciones inline:

- Las "*:inline"*: Estas acciones si sus precondiciones fallan, fallarán como una acción normal y producirán backtracking.
- Las "*:!inline*": Estas acciones no producen fallo, ni provocan backtracking aunque sus precondiciones sean falsas. Si la precondición es cierta la acción *inline* aplicará los efectos, en otro caso no hará nada.

Las acciones *inline* no deben incluirse en el plan resultante puesto que no tienen la semántica de una acción primitiva real. Los literales con los que operen las acciones *inline* no deben alterar el estado del mundo, puesto que estas acciones no están diseñadas para finalmente ejecutarse. Solamente podrán alterar los literales internos al proceso de planificación que tienen información de control.

Ejemplo 15.

```
; Uso de acciones inline

;; Esta tarea evita hacer múltiples llamadas a la función
;; calcula_distancia insertando en el estado un literal
;; con la distancia cacheada. Acelera la planificación en
;; dominios donde este cálculo es muy habitual

(:task Cachear_Distancias
  :parameters (?punto)
  (:method unico
   :precondition ()
   :tasks (
            (:inline () (forall (?x - Posicionable)
                              (when (and
                        (disponible ?x)
                        (posicion ?x ?pos)
                        (assign ?d (calcula_distancia ?punto ?pos)
                      )
                                          (assign (distancia_cacheada ?x ?punto) ?d)
            )))
   )
)
```

El corte (!) es otro de los conceptos introducidos para mejorar la eficiencia del proceso de planificación. Este concepto está tomado del lenguaje PROLOG [PROLOG] donde se utiliza para "olvidar" las unificaciones previas si se vuelve por backtracking. Obviamente el uso del corte afecta a la completitud del proceso de refutación, por lo que debe de ser utilizado con cuidado.

En HTN-PDDL es posible usar el corte para parar el backtracking en dos lugares distintos:

- En la definición de precondiciones: Su uso es exactamente el mismo que el corte de PROLOG. Se toma de la lista de posibles unificaciones una de forma no determinista y se descarta el resto. Si se vuelve por backtracking no se vuelven a considerar el resto de las unificaciones. En dominios con muchas unificaciones usado con cuidado puede utilizarse para acelerar mucho el procesamiento y reducir el backtracking, sin que por ello se vea afectada la completitud.
- En la lista de métodos de una tarea abstracta. Sirve para que una vez que se han probado como ciertas las precondiciones de un método se descarte probar con el resto de métodos. Nuevamente usado con cuidado este corte tampoco tiene por que afectar a la completitud del algoritmo. El escritor de dominios puede conocer que los métodos son mutuamente excluyentes y que una vez que se prueba con uno, el resto ya son inválidos.

Demostremos el uso del corte con un ejemplo:

Ejemplo 15.

```
; Uso de los cortes

(:task Dormir
  :parameters (?p – persona)
  (!
   (:method en-casa
    :precondition (and
                (en_ciudad ?p ?c)
                (residencia ?p ?c ?h)
     )
    :tasks (
          (Ira ?p ?h)
          (Ducharse ?p)
          (Ponerse_Pijama ?p)
          (Dormir ?p)
   )
  )
  (:method en-hotel
```

```
   :precondition (and
                (end_ciudad ?p ?c)
                (hotel ?c ?h)
                (camas_libres ?h)
                (not (residencia ?p ?c ?l)
    )
    :tasks (
         (Ira ?p ?h)
         (Reservar_habitacion ?p ?h)
         (Ducharse ?p)
         (Ponerse_Pijama ?p)
         (Dormir ?p)
    )
)
```

En este ejemplo es muy claro que ambos métodos son excluyentes. La exclusión sin embargo podría no ser tan obvia y ocurrir a niveles más bajos en la jerarquía. El proceso de unificación con los hoteles disponibles en la ciudad puede ser costoso (sobre todo si hay muchos) por lo que si ya sabemos que vamos a dormir en casa, no hace falta que el planificador pruebe todas las posibles unificaciones con todos los hoteles de la ciudad.

Otra forma alternativa de construir la precondición del método en-hotel podría ser la siguiente:

Ejemplo 16.

```
; Uso de los cortes
...
   (:method en-hotel
    :precondition (!(and
                (end_ciudad ?p ?c)
                (hotel ?c ?h)
                (camas_libres ?h)
                (not (residencia ?p ?c ?l)
    ))
...
```

Esta precondición al tener un corte en las precondiciones únicamente probará con el primer hotel que encuentre libre en la ciudad. Si se vuelve por backtracking la precondición fallará. Obviamente se pueden perder soluciones válidas, pero la decisión es responsabilidad del escritor de dominios.

14. Gestión del tiempo

HTN-PDDL usa una aritmética basada en el concepto de *timepoint* o referencia temporal para definir relaciones relativas de unas acciones con respecto a otras. Toda acción dispone de dos *timepoints* referenciados con las variables especiales del lenguaje ?start y ?end que marcan el comienzo y el final de la acción.

HTN-PDDL permite almacenar referencias a estos *timepoints* para posteriormente utilizarlos usando una aritmética especial de *timepoints*.

Cuando hacemos uso del tiempo en el dominio o en el problema es necesario declarar en el bloque de requisitos que el planificador debe ser capaz de realizar la gestión del tiempo (ver subsección de requisitos).

14.1. *Acciones durativas*

Las acciones durativas fueron introducidas en PDDL 2.2, para dar soporte temporal al PDDL estándar. Dependiendo de la expresividad que tenga el planificador se dice soporta PDDL al nivel 1 (versión anterior de PDDL 1.2) , 2 (soporte de fluents), 3 (soporte de operadores primitivos

durativos) o 4 (soporte de efectos continuos de las acciones en el tiempo). La expresividad de HTN-PDDL llega hasta el nivel 3, es decir hasta el soporte de acciones y literales con tiempo, por lo que los efectos numéricos de las acciones son tratados de una forma discreta.

HTN-PDDL sin embargo tiene una expresividad más rica en el uso del tiempo que PDDL 2.2 nivel 3, aunque las acciones primitivas durativas tienen prácticamente la misma sintaxis:

Las acciones durativas se definen en HTN-PDDL de la siguiente forma:

```
<durative-action-def>:      (:durative-action <name>
                                                        :parameters (<typed-variable>*)
                                                        [<meta-tags>]
                                                        :duration (= ?duration <fluent-exp>)
                                                        [:condition <timed-goal>]
                                                        [:effect <timed-effect>])

<timed-goal>:                       (<time-specifier> <sgoal-def>)
                                                        | <goal-def>

<time-specifier>:           <time-point>
                                                        | over all
                                                        | between <time-point> and <time-point>

<time-point>:                       at start
                                                        | at end
                                                        | <f-time-point>

<f-time-point>:             at <fluent-exp>


<timed-effect>:             (<time-point> <p-effect>)
```

La estructura de la acción durativa es muy similar a la de una acción sin tiempo, con la salvedad de que la durativa tiene asociada una duración codificada como un valor numérico fijo, o calculada a través de una expresión fluent (de esta forma se puede hacer el valor de la duración dependiente del estado). En cualquier caso es requisito indispensable que el valor numérico resultante sea mayor o igual a 0 (las acciones con duración negativa no tienen sentido).

Ejemplo 17.

```
; Poner la duración a una acción
; como se ve los efectos no son continuos
(:action correr
 :parameters (?fromx ?fromy ?tox ?toy)
 :duration (= ?duration (* 15000 (distance ?fromx ?fromy ?tox ?toy)))
 :precondition (in ?fromx ?fromy)
 :effect (and
         (at start (not (in ?fromx ?fromy)))
         (at end (in ?tox ?toy))
))
```

HTN-PDDL permite usar la cláusula *between* para manejar el concepto de *timeline* que se discutirá más adelante.

14.2. *Uso del timeline*

Los literales en el estado están temporizados y definen lo que se denomina un *timeline* o evolución del estado a lo largo del tiempo. La temporización se realiza de forma explícita en la declaración del problema en HTN-PDDL. Si en la declaración del problema un literal no está temporizado entonces se supone que es válido desde el instante 0 (instante inicial) y hasta que un operador primitivo lo niegue.

Podemos declarar tres tipos de literales distintos:

- Literales que se hacen válidos en un instante de tiempo determinado hasta que el efecto de un operador primitivo lo niegue. Se declaran utilizando la cláusula de PDDL *at: (*at *instante_temporal* (*literal_temporizado)).*

- Literales que son ciertos durante un intervalo de tiempo determinado. Se declaran utilizando la cláusula de HTN-PDDL *between:* (between *instante_temporal_1* and *instante_temporal_2* (*literal_temporizado*)).
- Literales que son ciertos de forma periódica. Se declaran utilizando la cláusula de HTN-PDDL *between:* (between *instante_temporal_1* and *instante_temporal_2* and every *unidades_de_tiempo (literal_temporizado*)).

Veamos ejemplos concretos de cada uno de ellos:

```
Ejemplo 18:
;;Declaración de distintos tipos de literales temporizados
(:init
  ;; literales que son válidos desde el instante 0
  (= (precio linea_bus11 110))
  (en alhambra_palace alhambra)
  ;; literales ciertos en un instante de tiempo determinado
  ;; literal válido 120 unidades de tiempo después del inicio
  (at 600 (disponible coche_alquiler))
  ;; literal disponible en un instante temporal fijo
  (at “12:00:00 12/05/2007” (disponible habitacion_101 alhambra_palace))
  ;; literal válido solamente en un intervalo de tiempo
  (between 600 and 120 (fiesta fiesta_cerveza alhambra_palace))
  ;; literal periodico
  (between “08:30:00 12/05/2007” and “18:00:00 12/05/2007” and every 870 (abierto alhambra))
...
)
```

Hay que hacer una serie de consideraciones sobre el ejemplo mostrado. Primero se observa como en HTN-PDDL es posible mezclar referencias a un instante temporal relativas al instante de tiempo con referencias absolutas expresadas como cadenas fecha/hora. De esta forma se facilita la escritura de dominios temporales y la legibilidad de los mismos, pero también nos obligan a introducir una serie de parámetros en el planificador para que este sea capaz de interpretarlos. Estos parámetros son introducidos en la sección de personalización (*customization*) (ver la sección dedicada al bloque de personalización más adelante) del fichero del problema y marcan la unidad temporal que usaremos (en los literales del ejemplo minutos), el formato de fecha/hora, y el instante temporal que se toma como inicio.

```
Ejemplo:
;; Ejemplo de sección de personalización, para los literales mostrados en
;; el ejemplo anterior
(:customization
          (= :time-format "%H:%M:%S %d/%m/%Y")
          (= :time-horizon-relative 1000)
          (= :time-start " 00:00:00 12/05/2007")
          (= :time-unit :minutes)
)
```

14.3. *Restricciones temporales en las redes de tareas*

Las restricciones temporales se imponen en la definición de las redes de tareas. La sintaxis que se usa en HTN-PDDL es la siguiente:

```
<dur-constraints>:          <dur-constraint>
                            | (and <dur-constraint>+ )
                            | ( )

<dur-constraint>:           (<time-operator> ?dur <fluent-exp>)
                                                    | (<time-operator> ?start <fluent-exp>)
                                                    | (<time-operator> ?end <fluent-exp>)

<time-operator>:            =
                                                | <
                                                | >
                                                | <=
                                                | >=
```

Cada acción de la red de tareas puede hacer referencia a sus *timepoints* usando las variables

especiales “?start” y “?end”. También puede hacer referencia a su duración mediante la variable “?dur” a la hora de definir restricciones. Los siguientes ejemplos muestran la definición de algunas restricciones en un ejemplo sencillo:

Ejemplo 19:

```
;; Tres tareas abstractas con relaciones de herencia
;; La tarea A (MA) no impone más restricciones que las de ordenación
(:task A
 :parameters ()
 (:method MA
  :precondition ()
  :tasks ((A1)(A2))
))

;; La tarea B (MB1) pone restricciones en el inicio y el fin de la
;; subtarea A2
(:task B
 :parameters ()
 (:method MB1
  :precondition ()
  :tasks ((A1)
         ((and (>= ?start 3)(<= ?end 5)) (A2)))
))

;; Definición de la tarea A2 utilizada por las tareas
;; abstractas anteriores
(:task A2
 :parameters ()
 (:method A2
  :precondition ()
  :tasks ((a21)(a22))
))
```

Observar cómo en el método MB1 de la tarea del ejemplo B, se impone la restricción de que la tarea A2 se tiene que ejecutar en el intervalo de tiempo [3,5]. Existe un mecanismo de herencia de restricciones temporales que hace que las tareas hijas en una red de tareas hereden las restricciones de sus tareas abstractas padre. De esta forma, en nuestro ejemplo si las tareas a21 y a22 durasen entre las dos (se tienen que ejecutar en secuencia) más de dos unidades temporales, se produciría una violación temporal que provocaría un backtracking en el planificador.

Las variables *?start* y ?*end* de una tarea también se pueden utilizar para definir sincronizaciones complejas entre tareas o límites máximos o mínimos en la distancia temporal entre dos tareas. Estas variables son referencias al inicio y al fin de la tarea en el modelo temporal subyacente

Estas referencias (o *landmarks*) forman parte de la estructura de tipos básica gestionada en HTN-PDDL. De este modo se permite que formen parte de los argumentos de un literal y por lo tanto pueden ser insertadas en el estado de planificación para ser posteriormente ser consultadas más adelante en el proceso de planificación y servir para definir restricciones en otra red de tareas. Por comodidad los *landmarks* suelen ser definidos en la descripción de dominios como *fluents* que pueden ser utilizados directamente en los operadores de comparación al describir una restricción.

El siguiente ejemplo muestra la inserción en el estado del planificador de los landmarks:

Ejemplo 20:

```
;; inserción de los landmarks de una tarea abstracta y de un operador primitivo en el estado
(:task A2
 :parameters ()
 (:method MA2
  :precondition ()
  :tasks ((:inline () (and (assign (start A2) ?start)
                           (assign (end A2) ?end)))
         (a21)
         (a22))
))

(:durative-action b
 :parameters()
 :duration (= ?duration 1)
```

```
 :condition()
 :effect(and (assign (start b) ?start)
        (assign (end b) ?end)
))
```

Observar cómo se hace uso de las acciones *inline* para introducir los *landmarks* de la tarea abstracta A2. Recordemos que las acciones *inline* se usan en HTN-PDDL para introducir conocimiento de control en el estado del planificador, como es la información temporal, y que estas acciones no deben ser contempladas en el plan final. Al operador primitivo b en cambio no le queda más remedio que insertar la información temporal como efectos.

Esta información temporal puede ser utilizada para definir restricciones en la red de tareas de un método distinto, como se aprecia en el siguiente ejemplo:

```
Ejemplo 21:
;; Uso de un landmark guardado con anterioridad para hacer una
;; sincronización.
(:task A3
 :parameters ()
 (:method MA3
  :precondition ()
  :tasks (((= ?start (start A2)) (b)))
))
```

En el ejemplo la tarea A3 en su método MA3 utiliza el landmark guardado en el ejemplo 20 del inicio de la tarea A2, para sincronizar el inicio de la acción b con dicha tarea.

15. Definición de un problema en HTN-PDDL

El fichero de problema en HTN-PDDL incluye el estado inicial del entorno y el objetivo a cumplir expresado como una red de tareas a resolver. En esto difiere del modelo planteado en PDDL, que está diseñado para dominios planos. También el manejo que se da a los timed initial literals es ligeramente distinto en HTN-PDDL.

La notación de la sintaxis de un fichero de problema de HTN-PDDL es la siguiente:

```
problem_definition:     (define <problem-name> <domain-name>
                                                    <problem-body>)

<problem-name>:         (problem <name>)

<domain-name>:          (:domain <name>)

<problem-body>:         [<require-def>]
                            [<customization-def>]
                                                    [<obj-declaration]
                                                    [<init>]
                                                    <goal>
```

Cada problema tiene un nombre para identificarlo y va asociado a un dominio también con un nombre determinado. Es un error sintáctico introducir en el planificador un problema y un dominio donde los nombres de dominio no coincidan.

15.1. *Declaración de constantes*

Las constantes (objetos) que se usan en la descripción del problema se declaran como una lista usando la siguiente notación:

```
<obj_declaration>:      (:objects <constant-def>* )
```

Es muy sencillo de entender mediante un ejemplo:

```
Ejemplo 22.
; Declaración de constantes
(:objects
   manuel_diaz – persona
   bugs_bunny – dibujo_animado
   A320 – vehiculo_aereo
   ....
)
```

15.2. *Declaración del estado inicial*

Los literales que son ciertos en el estado inicial se declaran en la sección de inicialización, cuya sintaxis es la siguiente:

```
<init>:                    (:init <init-el>* )

<init-el>:                 <atomic-formula(constant|number)>
                           | (= <atomic-formula(constant|number) number)
                                          | (<f-time-point> <atomic-formula(constant|number))
                           | (between <date> and <date>
                                          [and every <number>]
                                          <atomic-formula(constant|number)> )
```

La diferencia principal con respecto a PDDL es la posibilidad de declarar literales que se hacen ciertos de forma periódica, o en ciertos instantes de tiempo, como ya se mostró en la subsección dedicada al *timeline*:

```
Ejemplo 23.
; Declaración de literales
(:init
   ;; un literal normal
   (fabricante A320 Airbus_SAS)
   ;; un literal fluent
   (= (velocidad_maxima airbus_320) 0.82)
   ;; un literal temporizado que se hace cierto
   ;; en un instante determinado
   (at “28/03/1988 12:00:00” (disponible A320))
   ;; declaración de un evento periodico
   (between “01/01/2007 00:00:00” and  “01/01/2007 11:59:59” and every 12 (es-maniana))
   ;; si, antes del uno de enero del 2007 no había mañanas ;).
   ;; observar que estamos usando como unidad de tiempo las horas,
   ;; esto se fija en la sección de personalización (customization)
   ....
)
```

15.3. *Declaración del objetivo*

El objetivo a resolver se declara como una red de tareas. Se entiende que debe de haber al menos una tarea. En otro caso se obtendría un plan vacío. Las tareas a incluir en la red de tareas, normalmente serán abstractas. En el caso de que todas fueran primitivas, el plan resultante sería simplemente una ordenación de las tareas primitivas (con marcas de tiempo o no dependiendo si el dominio es temporal) que se han escrito (si estas son aplicables en el estado inicial).

La gramática del lenguaje que se usa en la construcción de un objetivo es la siguiente:

```
<goal>:                    (:tasks-goal
                                   [<comment>]
                                   :tasks <task-network>
                                          )
```

Un ejemplo sencillo de declaración de objetivo es la siguiente:

Ejemplo 24.

```
; Definición de objetivo
(:tasks-goal
 :tasks [
                                (Tarea1)
                                (Tarea2)
                                (Tarea3)
                        ]
)
```

16. Parámetros de usuario

Con la inclusión de la información temporal y otras características de HTN-PDDL y a fin de hacer más sencilla la escritura de dominios, se permite la personalización de algunos parámetros de usuario en la sección *customization.* La sección *customization* tiene la siguiente notación para su sintaxis:

```
<customization-def>:                    (:customization <customization-element>*)

<customization-element>:    (= :time-unit <time-unit>)
                            | (= :time-format <text>)
                            | (= :time-horizon-limit <date>)
                            | (= :time-horizon-relative <date>)
                            | (= :time-start <date>)

<time-unit>:                            :hours
                                        | :minutes
                                        | :seconds
                                        | :days
                                        | :years
                                        | :months
```

En la versión 1.1 todos los parámetros tienen que ver con información temporal, pero esta previsto que esta sección incluya muchos otros parámetros en la siguiente versión.

El significado de cada uno de los parámetros es el siguiente:

- El parámetro **time-unit** define la unidad de tiempo a utilizar cuando es necesario utilizar referencias temporales relativas. Las referencias temporales relativas se codifican mediante un valor numérico. Por ejemplo si (= :time-unit hours), cuando se utilice un 3 en una expresión temporal, significará 3 horas.
- El parámetro **time-format** define el formato que se usará para las referencias temporales absolutas. Por comodidad y legibilidad del dominio se permite definir fechas y horas utilizando cadenas de texto. time-format define el formato de estas cadenas utilizando el mismo formato que el usado por la función del estándar ANSI de C strftime, que es el más extendido.
- El parámetro **time-start** se usa marcar el punto de tiempo inicial, que las acciones tomarán como referencia a la hora de anclarse. Ninguna acción debería comenzar antes del time-start.
- El parámetro **time-horizon** define el horizonte temporal hasta el que el planificador permitirá que las acciones se deslicen. Este parámetro además es utilizado para cortar la generación de intervalos de tiempo para literales intervalares. En principio estos literales podrían generar infinitos intervalos, como cada intervalo supone un punto de anclaje, el planificador podría caer en un bucle infinito hacia adelante y hacia atrás cuando vuelva por backtracking.
- El parámetro **time-horizon-relative**: es equivalente al *time-horizon* pero en lugar de tomar una referencia de tiempo absoluta, la toma relativa al inicio.

Ejemplo 25.

```
; Ejemplo de sección de personalización.
```

```
(:customization
        (= :time-format "%d/%m/%Y %H:%M:%S")
        (= :time-horizon-relative 1000)
        (= :time-start "27/1/2007 8:10:0")
        (= :time-unit :minutes)
)
```

El lenguaje permite que la sección de personalización aparezca tanto en el fichero de dominio como en el de problema, aunque lo más recomendable es que los parámetros *time-start* y *time-horizon* aparezcan en el fichero de problema. De otra forma en cierta medida se pierde la independencia del fichero de dominio del problema.

17. Requisitos

La sección de requisitos (*requirements)* define un conjunto de flags con características que el planificador debe soportar para hacer frente al dominio o al problema. Como no todos los planificadores son capaces de soportar la expresividad de HTN-PDDL, esto permite al diseñador de dominio imponer requisitos al planificador y al planificador descartar de una forma rápida a los problemas que no es capaz de abordar.

HTN-PDDL añade algunos requisitos más a los ya existentes en PDDL 2.2:

```
<require-def>:          (:requirements <require-key>*)

<require-key>:          :strips
                        | :typing
                        | :negative-preconditions
                        | :disjuntive-preconditions
                        | :equality
                        | :existential-preconditions
                        | :universal-preconditions
                        | :quantified-preconditions
                        | :conditional-effects
                        | :fluents
                        | :adl
                        | :durative-actions
                        | :derived-predicates
                        | :timed-initial-literals
                        | :metatags
                        | :htn-expansion
```

Por defecto todo planificador tiene que soportar el modelo de STRIPS, por lo tanto este *requirement* siempre se supone aunque no se establezca explícitamente. La descripción de las características que cada *requirement* imponen al planificador es la siguiente:

- **strips:** Modelo de acciones similar al de STRIPS.
- **typing:** Soporte para tipar los objetos del dominio respetando una jerarquía de tipos.
- **netative-preconditions:** Permite usar negaciones en las precondiciones.
- **disjuntive-preconditios:** Permite usar el operador “or” en las precondiciones.
- **equality:** Permite utilizar el predicado de igualdad “=” para la comparación de constantes.
- **existential-preconditions:** Uso de precondiciones cuantificadas existencialmente.
- **universal-preconditions:** Uso de precondiciones cuantificadas universalmente.
- **quantified-preconditions:** Es “azúcar sintáctica” ya que es equivalente a declarar en un dominio el soporte para precondiciones existenciales y universales.
- **conditional-efects:** Permite el uso del operador “when” para definir efectos condicionales.
- **fluents:** Permite el uso de la aritmética con números enteros y reales, así como la utilización

de operadores matemáticos en las precondiciones. Permite asignar números a predicados y hacer operaciones con ellos en los efectos.

- **adl:** Incluye todas las capacidades expresivas del lenguaje ADL *(Action Description Language)* [adl] . Es equivalente a declarar en un dominio que el planificador debe soportar los requisitos de *Strips, typing, negative-preconditions, disjuntive-preconditions, equality, quantified-preconditions,* y *conditional-effects"*.
- **durative-actions:** Permite el uso de acciones durativas en el dominio. Cuando se impone este requisito implícitamente también se hace el de *fluents.*
- **derived-predicates:** Permite la definición de axiomas en el dominio.
- **timed-initial-literals:** Permite el uso de predicados temporizados. Destacar que la filosofía y el tratamiento de este tipo de predicados es **distinto** en HTN-PDDL que en el PDDL 2.2 nivel 3 estándar.
- **metatags:** Permite incluir tags en diferentes partes del dominio con información extra no imprescindible para el planificador, pero necesaria en procesamientos posteriores del plan por otras herramientas.
- **htn-expansion:** Requiere el manejo de redes jerárquicas y tareas abstractas.