BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

“PLANIFICACIÓN Y CONTROL DE RUTAS PARA ESTACIONAMIENTOS TIPO PILA DE ALTA DENSIDAD”

TESIS

PARA OBTENER EL TÍTULO DE

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA

C. BEATRIZ MÁRQUEZ RODRÍGUEZ

DIRECTOR DE TESIS

DR. FERNANDO ZACARIAS FLORES

Puebla, Pue 18 octubre 2019

Otoño

Dedicatoria

A mis tíos, Gabriel y Lourdes por su apoyo incondicional. Por guiarme con el ejemplo y así alentarme a nunca rendirme y lograr todos mis objetivos. Gracias, tío por nunca soltar mi mano aun en los momentos más difíciles, por siempre alentarme cuando más lo he necesitado. Tía gracias a ti elegí el camino universitario, gracias por retarme y así hacer que diera lo mejor de mí.

A mi madre, Magdalena gracias nunca será una palabra que describa todo lo agradecida que estoy, por todo lo que me diste, gracias por estar a mi lado siempre que te necesite, en las buenas y en las malas, por sacarme adelante, gracias, mamá.

A mis hermanos, Jonathan y Omar por su ejemplo, siempre quise seguir sus pasos gracias por guiarme y mostrarme el buen camino, siempre fueron un ejemplo a seguir.

A mi cuñada y sobrino, Elisabeth y Gabriel A. por ser parte de mi familia y apoyarme siempre en todo momento.

A mi esposo e hijo, Brandon y Michel mis dos amores en la vida. Brandon gracias por apoyarme todos estos años por nunca negarme nada por dar siempre más de lo que podías gracias por estar en las buenas, pero sobre todo en las malas, Te amo. Michel veo en tus ojos una luz que me lleva a seguir y ser cada día mejor, gracias por ser mi inspiración y mi motivación para lograr todo lo que me preponga, te amo hijo.

A María José, después de tantos años de amistad solo puedo decirte "gracias". Por todo el apoyo incondicional, por darme la mano cuando más te necesitaba, por siempre estar ahí, te quiero amiga.

Agradecimiento

Al Dr. Fernando Zacarias Flores
Gracias por el apoyo en la realización de este trabajo de tesis.

Índice general

Capítulo 1 **Introducción**

Los centros históricos de ciudades como la nuestra enfrentan problemas con los estacionamientos existentes debido al espacio tan reducido para atender de manera eficiente a sus clientes. Los centros históricos de las ciudades, pese a la tendencia descentralizadora normalmente establecida en los planes de desenvolvimiento de sus áreas urbanas, constituyen siempre puntos inevitables de gran concentración de tráfico, originado tanto por la gran densidad de habitantes que tradicionalmente trabajan en ellas, como por las actividades allí implantadas y por las innumerables personas que diariamente se trasladan a estos centros. Debido a estas concentraciones de tráfico, además de las innumerables situaciones de congestión que provocan, y básicamente en las horas pico, crean problemas de estacionamiento bastante graves. Por esta razón, es necesario la adopción de medidas, muchas veces drásticas, por parte de las entidades municipales

El estacionamiento es un componente clave para la movilidad urbana eficiente con implicaciones en la congestión y el paisaje urbano. En el presente trabajo de tesis abordamos este problema como un trabajo inicial que permita brindar una solución eficiente a este problema. Así, planteamos como objetivo principal atacar el problema de planeación a través de lenguajes de acción tales como: C+, CCalc, Dlv^{K}, Smodels, etc.

El problema de planificación en Inteligencia Artificial consiste en la toma de decisiones realizada por entes inteligentes como robots, humanos o programas de computadora cuando intentamos alcanzar algún objetivo. Se trata de elegir una secuencia de acciones que transformarán el estado del mundo, paso a paso, de modo que alcance el objetivo (o meta). Se suele considerar que el mundo consiste en hechos atómicos, y las acciones hacen que algunos hechos sean verdaderos y otros falsos. Así, solo nos centraremos en el problema de planificación de inteligencia artificial más simple, caracterizado por la restricción a un agente en un entorno determinista que se puede observar en su totalidad. De esta manera, en el presente trabajo se abordará el problema de planificación considerando como caso de estudio los “estacionamientos tipo pila” (existentes en los centros históricos de las ciudades como la nuestra) [1, 2, 3].

1.1 Antecedentes del proyecto

La construcción de un plan de proceso puede considerarse como un ejercicio de programación lógica. Dados los hechos de la situación y las reglas de las condiciones que rigen los procesos y dada la declaración de un estado objetivo, un programa lógico intentará probar ese estado objetivo. La serie de reglas y hechos utilizados para la prueba será el plan de proceso. Cuando se usa un lenguaje de programación lógico como Prolog, si el problema es soluble, se encontrará una prueba. Una vez que se establecen los hechos, las reglas y el objetivo, el lenguaje invoca los procesos de cómputo necesarios para encontrar la prueba.

Este enfoque es especialmente útil cuando se deben desarrollar nuevos planes de forma rápida y flexible. Sin embargo, si uno intenta resolver el problema utilizando un enfoque simple de prueba de teoremas, entonces el tiempo y la memoria necesarios para generar el plan de proceso son excesivos. Una solución a un problema realista requiere atención al control del programa lógico. El control se logra mediante el uso de procedimientos, basados en varias técnicas heurísticas posibles, para ahorrar tiempo y memoria.

1.2 Objetivos Generales y Específicos del Proyecto.

Modelar e implementar una solución de planificación y control de rutas para estacionamientos tipo pila de alta densidad.

Objetivos específicos:

- Analizar y modelar la representación de conocimiento en un lenguaje basado en la programación lógica, como Dlv^{K}.
- Experimentar con el problema de planeación bien conocido como “El mundo de los bloques”.
- Determinar las principales características de representación de conocimiento, así como, la metodología de razonamiento en el lenguaje de acción Dlv^{K}.
- Describir de manera detallada cada uno de los componentes necesarios para la descripción de la solución de un problema de planeación.

1.3 Justificación

En este trabajo de tesis se propone resolver el problema de planeación desde una nueva perspectiva basada en la programación lógica. En los últimos años, se ha visto el desarrollo

de lenguajes de acción que proveen herramientas flexibles y expresivas para describir la relación entre fluentes y acciones. Este tipo de lenguajes han recibido considerable atención entre los investigadores de la comunidad de razonamiento y representación de conocimiento. El estudio de este tipo de lenguajes ha sido profundo y se han desarrollado lenguajes que han mejorado en mucho el modelado de problemas de planeación en diversas áreas del conocimiento.

Así, en este trabajo se muestra un ejemplo del mundo de la robótica, conocido como el mundo de los bloques [4], como paso inicial para entender como modelar problemas de planeación a través de la programación lógica.

El modelo de planificación clásico es el modelo más común para la planificación automatizada y se basa en los siguientes supuestos:

1. La tarea de planificación a resolver tiene un espacio de estado finito y completamente observable.

2. Las acciones son deterministas y causan transiciones de estado instantáneas.

3. Las metas son condiciones referidas al último estado alcanzado por un plan de solución

Por lo tanto, una solución para una instancia de planificación clásica es una secuencia de acciones aplicables que transforma un estado inicial dado en un estado objetivo, es decir, un estado que satisface un conjunto de condiciones de objetivos previamente especificado [5].

Si bien la mayoría de los sistemas de planificación existentes se basan en lenguajes de planificación "clásicos" como STRIPS (Fikes & Nilsson, 1971) y PDDL (Ghallab et al., 1998; Fox & Long, 2003), los últimos años han visto el desarrollo de lenguajes de acción que proporcionan herramientas expresivas y flexibles para describir la relación entre fluentes y acciones. Los lenguajes de acción han recibido considerable atención en la comunidad de representación y razonamiento del conocimiento y sus propiedades formales (complejidad, etc.) se han estudiado en profundidad. Se ha invertido menos esfuerzo en utilizar las construcciones ofrecidas por estos lenguajes para la resolución de problemas.

En este trabajo, abordamos esta deficiencia y elaboramos la representación y el razonamiento del conocimiento con lenguajes de acción, que son significativamente diferentes de los marcos estrictos basados en operadores de STRIPS y PDDL. Para ello, presentamos el lenguaje de planificación K [3] a través de su realización como Front-End sobre el sistema de planificación DLV^k [6], disponible en http://www.dbai.tuwien.ac.at/proj/dlv/K/. Discutiremos problemas de representación del conocimiento y proporcionamos pautas generales para la codificación de dominios de acción y ejemplos detallados para la ilustración.

Dentro de los resultados esperados tenemos los siguientes:

- Primero, mostrar cómo hacer la representación del mundo a través de un lenguaje como Dlv^K, para esto usaremos uno de los problemas típicos de planeación, conocido como "El mundo de los Bloques".
- Segundo, Ejemplificar y documentar de manera detallada la forma en que se debe modelar un problema de planeación usando Dlv^K.
- Tercero, resolver a través del lenguaje disyuntivo lógico (conocido como Dlv^K) el problema de un estacionamiento de tipo pila, describiendo cada uno de los componentes siguientes:
    - Los *Fluentes* que nos permiten caracterizar la configuración del mundo en que se sitúa el problema.
    - Las *Acciones* que permiten modificar el estatus del mundo en que se desarrolla el proceso.
    - Las *Causas* que son provocadas por las acciones válidas en nuestro mundo y que deben reflejar el nuevo estado de nuestro mundo.

Etc.

El propósito de este trabajo es lograr llevar el uso de la programación lógica al modelado de problemas de planeación. Y poder considerar este tipo de propuestas en áreas como la robótica o el desarrollo de agentes inteligentes. El sistema permitirá caracterizar como se modela la planeación clásica desde lenguajes basados en la programación lógica. Finalmente, nos interesa que en un futuro podamos modelar no solo problemas reales, sino más bien, problemas reales, completos y complejos.

1.4 Metodología

Se analizarán las diversas propuestas existentes en la literatura que sobre el estado del arte existe para identificar aquellos estándares pertinentes y útiles para el modelado de problemas de planeación en lenguajes de acción como Dlv^{K}. También, se estudiarán y analizarán los diferentes sistemas existentes, de tal forma que podamos justificar nuestra elección del lenguaje de acción para el modelado de problemas de planeación clásica. Se determinarán los requerimientos necesarios para la representación de conocimiento, considerando la abstracción necesaria del entorno a modelar. Por otro lado, se instalará el lenguaje de acción elegido para el modelado de los problemas de planeación. Para la evaluación del sistema, utilizaremos diversas configuraciones que nos permitan validar la robustez de nuestra solución, así como la flexibilidad y generalidad de nuestra solución.

Finalmente, se caracterizarán las principales propiedades de cada uno de los elementos que conforman la solución hallada. Así como, plantear trabajo a futuro basado en nuestra implementación.

De manera general debemos seguir los pasos siguientes:

- *Definición del problema*: Es decir, en esta primera fase nos interesa mostrar cómo debemos determinar cuál es la representación más adecuada del entorno en que se presenta el problema para poder hacer una correcta abstracción de este.
- *Objetos relevantes en nuestro mundo*: incluye como determinar que objetos son los mínimos necesarios para modelar correctamente nuestro problema.
- *Caracterización de configuración*: Identificar la mejor manera de caracterizar (a través de los llamados “Fluents”) los diversos escenarios como: el estado inicial; la configuración final; así como los escenarios intermedios o parciales alcanzados con la ejecución de las diversas acciones que modifican un estado.
- *Definir las acciones*: Determinar las acciones mínimas necesarias para la solución del problema, así como los diversos efectos que las acciones provocan y que deben ser consideradas.

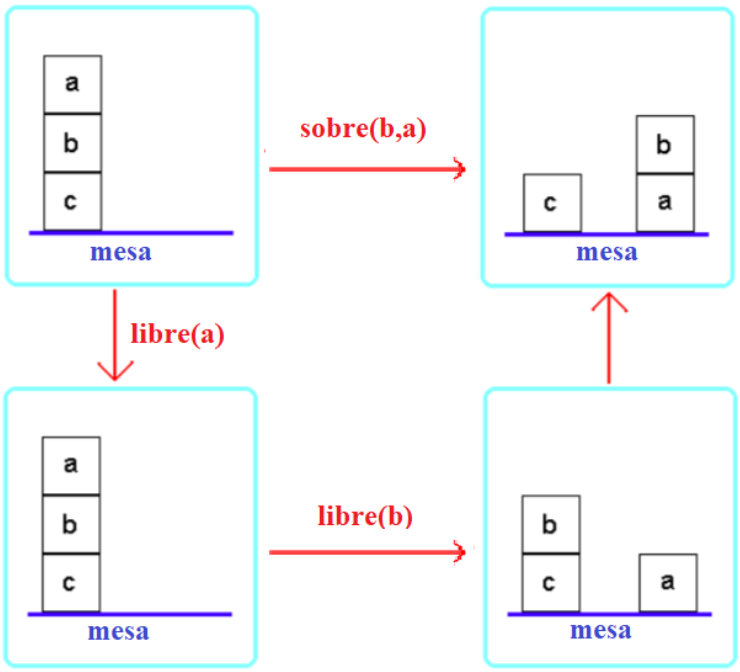

Figura 1. Modelo general de Planeación

En la figura 1 se muestra un esquema general del problema clásico del mundo de los bloques. En este esquema se puede observar cómo a través de los fluentes (sobre) podemos describir la configuración de estado que guarda cada uno de los objetos en un estado x en el proceso de planeación.

Dentro de las aportaciones relevantes de nuestra propuesta están:

- Lograr que una metodología dominio tecnológico pueda ser utilizada juntamente con una de dominio de los negocios para poder insertarlas en las etapas de un proyecto de desarrollo de software.
- Desarrollo de un sistema novedoso y vanguardista que emplea la inteligencia de negocios en apoyo a la toma de decisiones.
- Aplicación de sistemas de negocios inteligentes en el manejo de procesos de asignación de recursos financieros a las sucursales.
- Detonar el desarrollo de sistemas inteligentes incorporando todo el potencial que las nuevas tecnologías web y móviles ofrecen, tales como ser usado en cualquier lugar, en todo momento y con información en tiempo real.

1.5 Narrativa de capítulos

El presente trabajo de tesis se compone de cuatro capítulos: Introducción, Answer Set Programming, DLV y Planificación y Control de Rutas para Estacionamientos Tipo Pila de Alta Densidad.

El capítulo uno nos da un panorama general de lo que se trata este trabajo de tesis, cuáles fueron los objetivos, los antecedentes, la justificación y la importancia de la metodología empleada para lograr llegar a la meta. El capítulo dos da un panorama general de lo que es ASP (Answer Set Programming) el cual es la base de este trabajo de tesis, donde se abordan temas como: Conceptos teóricos, lógica proposicional, como es la realización del cálculo de los answer sets y algunos ejemplos que harán que quede más claro dicho calculo. El capítulo 3 trata sobre el sistema DLV el cual nos hablará sobre un poco de su historia, algunos fundamentos teóricos y ejemplos de programación en DLV. El capítulo 4 es la medula del presente trabajo de tesis, Planificación y control de rutas para estacionamientos tipo pila de alta densidad, se abordarán temas como DLV^k el cual es el lenguaje utilizado en este trabajo, ejemplos que se utilización para llegar a la resolución exitosa de nuestro caso de estudio. Al finalizar los capítulos presentaremos nuestro trabajo a futuro.

Por último, al final de este trabajo daremos nuestras conclusiones después de la realización del presente trabajo.

Capítulo 2 **Answer Set Programming (ASP)**

El objetivo de este capítulo es explicar la función que tiene *Answer Set Programming* (ASP) el cual tiene como objetivo solucionar problemas específicamente de planeación; cuyo enfoque de planificación independiente del dominio introduce a un programa lógico cuyas salidas serán las soluciones al problema de planeación o planificación dado. Existen programas que ayudan a solucionar este tipo de problemas tales como Smodels, Cmodels o DLV entre otros los cuales dan soluciones eficientes [7].

ASP tiene una forma de programación declarativa la cual nos ayuda a resolver problemas que son difíciles o llamados o conocidos como NP. Hablando de forma general ASP contiene todas las aplicaciones de conjuntos de respuestas [7].

En la figura 2.1 podemos algunos sistemas en los que Answer Set Programming ha sido implementado para la realización del cálculo de modelos.

ASSAT	assat.cs.ust.hk/
CLASP 1	potassco.sourceforge.net/#clasp/
CMODELS	www.cs.utexas.edu/users/tag/cmodels/
DLV 2	www.dbai.tuwien.ac.at/proj/dlv/ or www.dlvsystem.com/
GNT	www.tcs.hut.fi/Software/gnt/
SMODELS	www.tcs.hut.fi/Software/smodels/
XASP	xsb.sourceforge.net/, distributed with XSB

[1] + CLASPD, CLINGO, CLINGCON, among others; http://potassco.sourceforge.net/
[2] + DLVHEX, DLVDB, DLT, DLV-COMPLEX, ONTO-DLV, and others.

Figura 2.1 *Ejemplo de modelos que usan ASP*

Dentro de la literatura encontramos que ASP tiende a trabajar de la siguiente forma [8] (Pedro Cabalar, David Pearce y Agustín Valverde, 2009):

- Describe un dominio o problema utilizando reglas lógicas
- Utiliza restricciones para eliminar o seleccionar soluciones (modelos)
- El sistema computa los modelos estables (Answer Sets) que corresponden a las soluciones correctas del problema

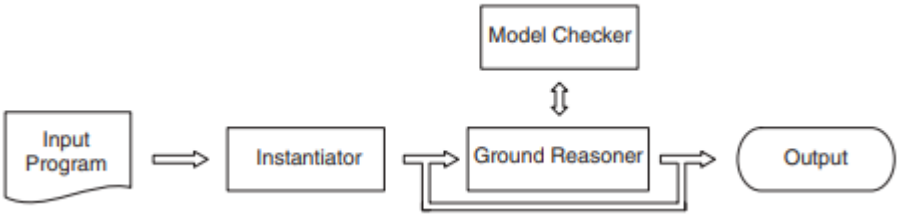

Figura 2.2 *Arquitectura General de ASP.*

Como vimos en la Figura 2.1 existen algunos sistemas que implementan ASP, si bien dichos sistemas no utilizan las mismas técnicas básicamente están de acuerdo con la arquitectura mostrada en la figura 2.2.

El flujo de evaluación de la computación se describe en detalle. Al inicio, la entrada *"Input Program"* especificada por el usuario se analiza y transforma en las estructuras de datos internas del sistema. En general, un programa de entrada P contiene variables, y el primer paso de un cálculo de un sistema ASP es eliminar estas variables, generando un *ground de instantiation ground*(P) de P. Este proceso de eliminación de variables se llama instanciación del programa (o grounding) y es realizado por el módulo "*instantiator"* [9].

En este capítulo podremos observar cómo es que ASP es un lenguaje de programación muy parecido a la programación lógica, pero tiene distintos métodos computacionalmente hablando también se hablará sobre algunos conceptos y definiciones para mejor entendimiento, los cuales fueron utilizados a lo largo de este trabajo.

2.1 Perspectiva Answer Set Programming (ASP)

En 1999 llega un nuevo paradigma el cual es identificado como una forma de programación declarativa: *Answer Set Programming.* Originalmente habría sido propuesto solo como la semántica para programas Prolog con negación, después se extendió a programas más generales [10].

Answer Set Programming (ASP) es un modelo lógico diseñado para la solución de problemas difíciles de búsqueda, razonamiento, restricciones, planificación entre otros tiene sus orígenes en las bases de datos deductivas, la programación lógica, la lógica basada en la representación y el razonamiento del conocimiento, la resolución de restricciones y las

pruebas de satisfacción [10]. ASP es un lenguaje con fuertes fundamentos teóricos ha tenido gran éxito en su desarrollo durante los últimos años desarrollado originalmente para aplicaciones en Inteligencia Artificial relacionadas con el conocimiento.

Ha tenido un gran impulso en la comunidad Científica en diferentes áreas tales como Bioinformática, Robótica, Sistemas de apoyo a la decisión utilizados por la NASA [10].

Dentro de nuestro contexto diremos que una teoría se compone por 3 elementos los cuales consideramos fundamentales los cuales son:

1. El conjunto de axiomas lógicos estos corresponderán a la lógica que se haya elegido.
2. El conjunto de axiomas propios dentro de nuestro contexto estos corresponden al conjunto de cláusulas que comprenden un programa lógico.
3. Reglas de inferencia las cuales son empleadas para obtener el nuevo conocimiento para este trabajo utilizaremos la regla "Modus ponens".

A continuación, un ejemplo de lo que sería una teoría dentro de nuestro contexto:

Ejemplo 2.1 Supongamos que P es un programa que solo contiene átomos positivos en el cuerpo, A y B letras proposicionales, entonces:

$P = \{A_1 \leftarrow B_{11}, ..., B_{1n}.$

...

...

...

$A_k \leftarrow B_{k1}, ..., B_{kn}.\}$

Consideremos a la lógica clásica como nuestro contexto teórico, por tanto, nuestros axiomas lógicos son:

Axioma 1) $A \rightarrow (B \rightarrow A)$ A1

Axioma 2) $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$ A2

Axioma 3) $(\neg B \rightarrow \neg A) \rightarrow ((\neg B \rightarrow A) \rightarrow B)$ A3

Y como axiomas propios las cláusulas de nuestro programa P:

$A_1 \leftarrow B_{11}, ..., B_{1n}.$

...

...

...

$A_k \leftarrow B_{k1}, ..., B_{kn}.$

Como única regla de inferencia el Modus Ponens. Así, tendríamos nuestra teoría.

2.2 Definición formal de Sintaxis y Semántica de ASP

La programación lógica disyuntiva involucra dos tipos de negación(referida como negación fuerte y negación como falla) bajo la semántica de los "*Answer Sets"* [9].

2.2.1 Sintaxis

Siguiendo los antecedentes de Prolog, las cadenas que comienzan con letras mayúsculas denotan variables lógicas, mientras que las cadenas que comienzan con letras minúsculas denotan constantes; esto nos dice que un *término* puede ser una constante o una variable.

Un *átomo* es es una expresión de la forma $P(t_1,...,t_n)$ de donde P es un predicado con aridad n y $t_1,...,t_n$ son términos. Una literal clásica L es un átomo p(en este caso es positiva), o un átomo negativo ¬p(en este caso es negativa). Una literal L es considerada una literal de negación como falla (NAF por sus siglas en inglés) si es de la forma: *l* o *no l*, donde *l* es una literal clásica. De primer instancia *l* será considerada positiva y en último caso negativa a menos que se especifique lo contrario [9].

En ASP generalmente se requiere que las reglas utilizadas para la solución de los programas sean seguras. Esta seguridad proviene de las bases de datos, donde la seguridad se introdujo como un medio para garantizar que las consultas (En el caso de ASP son lo programas) no dependan del universo (el conjunto de constantes considerado) [9].

Una regla será considerada segura si cada variable en esa regla aparece por lo menos una literal positiva dento del programa en el cuerpo de esa regla. Como se menciono

anteriormente un programa ASP será seguro si cada una de sus reglas es segura, al final diremos que el resultado es un programa seguro y por lo tanto funcional [9].

2.2.2 Semántica

A continuación se describe la semántica de los programas utilizada en ASP. Se ha notado que este supone la disponibilidad de algunos predicados preinterpretados como: ¼; <; >: sin embargo, esto llevo a considerarlos como "hechos" por lo que no son tratados de manera especial.

Para cualquier programa P, el universo de Herbrand, denotado por U_P, es el conjunto de todas las constantes que ocurren en *P*. Si no ocurre una constante en P, U_P consiste en una constante arbitraria. La base literal de Herbrand B_P es el conjunto de todos los literales básicos (clásicos) construibles a partir de símbolos predicados que aparecen en P y constantes en U_P (se debe tener en cuenta que, para cada átomo *P*, B_P contiene también el literal fuertemente negado ¬p).

Ejemplo 2.2 Consideremos el siguiente programa:

P_0 = {

r1: a(X) ∨ b(X) ← c(X,Y).

r2: c(X) ← c(X,Y), not b(X).

r3: c(1,2).

}

Entonces en universo es U_{P0} = {1,2}, y la base es B_{P0} = {a(1), a(2), b(1), b(2), c(1), c(2), c(1,1), c(1,2), c(2,1), c(2,2), ¬a(1), ¬a(2), ¬b(1), ¬b(2), ¬c(1), ¬c(2), ¬c(1,1), ¬c(1,2), ¬c(2,1), ¬c(2,2)}.

2.3 Lógica Proposicional

La lógica proposicional trata sobre la verdad o la falsedad de una proposición. Las palabras por sí mismas no dicen nada, pero cuando se encuentran en el contexto de una proposición es cuando estas empiezan a tomar un sentido. Las palabras no tienen un valor de verdad, este lo adquieren cuando forman parte de una proposición. En cualquier ciencia se necesita un lenguaje, el cual permita eliminar todo aquello que no interesa y ponga en manifiesto todo

aquello que sí. En la lógica proposicional se necesita de unos símbolos que indiquen la forma en que se combinan de acuerdo con su significado, al conjunto de estos símbolos, lo llamaremos "Lenguaje Formal" [12]. Un lenguaje proposicional contiene símbolos proposicionales a_0, a_1, conectivos:

- ∧ Conjunción,
- ∨ Disyunción,
- ← Derivación también denotada como: →,
- ⊥ Contradicción,
- T Tautología.

^ y ∨ son conectivos lógicos.

De dónde la negación por falla o negación débil se denota como: '¬' o también por la palabra 'not'.

Los símbolos proposicionales son llamados átomos. Una regla es un par ordenado como: [9]

$$Head(r) \leftarrow Body(r)$$

2.4 Tipos de Cláusulas

Como se menciona anteriormente en el punto 2.3 dentro de la literatura se puede encontrar que una fórmula de la forma $H \leftarrow B$ donde H y B son conocidas como la cabeza (*head*) y el cuerpo (*body*) de la cláusula respectivamente. Los hechos y las restricciones son otros dos casos particulares de cláusulas y tienen la forma siguiente $H \leftarrow$ T que representa los hechos y $\perp \leftarrow$ B que representa las restricciones. [12]

Las cláusulas libres son conjuntos de literales en la cabeza y el cuerpo de la fórmula, pero en la cabeza dichas literales se muestran en forma de disyunción y en el cuerpo en forma de conjunción como se muestra a continuación: [12]

$$h_1 \vee \ldots h_n \leftarrow b_1 \wedge \ldots \wedge b_m$$

Donde cada hi y bj son una literal de dónde podemos observar "i" va de 1 a "n" y "j" va de 1 a "m" respectivamente.

Cláusula disyuntiva, este tipo de cláusulas no tienen la cabeza vacía, esto quiere decir que no son una restricción.

Cláusula aumentada, esta forma de cláusula no es tan restringida y se pueden usar los conectivos ^ y ∨ y la negación ¬ tanto en la cabeza como en el cuerpo de la cláusula.

Ejemplos de cláusulas:

$a \vee b \leftarrow c \wedge d \wedge \neg e$	Disyuntiva, general, libre, aumentada
$\bot \leftarrow p \wedge q$	General, libre, aumentada (Restricción)
$a \vee \neg b \leftarrow p \wedge \neg q$	Libre, aumentada
$a \vee \neg a$	Libre, aumentada (hecho)
$\neg\ (p \wedge \neg q) \leftarrow a \vee (\neg b \wedge c)$	Aumentada

Los conectivos ⊥ y T son llamados formulas elementales.

Las fórmulas construidas por los conectivos ∧ y ∨ sobre formulas elementales se les conoce como formulas básicas.

2.5 ASP Conceptos Teóricos

El modelo de ASP para solucionar un problema consta de dos pasos [9]:

1. Convertir un programa de primer orden en uno proposicional, mediante la sustitución sistemática de variables con valores concretos de algún dominio.
2. Un solucionador toma el programa básico y asigna valores de verdad a los átomos para obtener los modelos estables del programa.

Diferentes formas de realizar código darán diferentes tiempos al resultado. También depende de cómo se especifique un problema es como obtendrá la solución.

ASP ayuda a comprender los problemas siendo útil para la elaboración de prototipos.

Enseguida, definimos algunos de los conceptos teóricos empleados en el desarrollo de este trabajo.

2.6 Definiciones de ASP

Definición 2.1 Todo lenguaje se compone de símbolos y reglas las cuales dirán si la combinación realizada es correcta o no. Para eso las siguientes reglas para formación de fórmulas bien formadas (FBF) nos dicen que [12]:

Regla 1: Toda proposición atómica es una fbf

Regla 2: Si A es una fbf, entonces ¬A también es una fbf

Regla 3: Si A y B son fbf, entonces $(A \wedge B)$, $(A \vee B)$ y $(A \rightarrow B)$ también son fbf

Definición 2.2 Dado P un programa básico constituido por fórmulas básicas. Un conjunto de letras proposicionales X se dice cerrado bajo P si para toda cláusula $H \leftarrow B \in P$ tenemos que $X \models H$ siempre y cuando $X \models B$ esto es, si $X \models B$ entonces $X \models H$ [13].

Definición 2.3 Sea X un conjunto de letras proposicionales y P un programa constituido por fórmulas básicas, X es denominado un Answer Set de P si el conjunto X es mínimo entre los conjuntos de átomos cerrados bajo P [13].

Definición 2.4 (Answer Set) La reducción de un programa normal (disyuntivo, libre, aumentado) relativo al conjunto de letras proposicionales X, es definido de la siguiente manera [13]:

Para una formula elemental F:

$F^x = F$

$F \wedge G) = F^x \wedge G^x$

$(F \vee G)^x = F^x \vee G^x$

$(\neg F)^x = \perp$ si $X \models F$; T en otro caso.

$P^x = (H \leftarrow B)^x$ tal que $H \leftarrow B \in P$

Definición 2.5 Sea P un programa y X un conjunto de átomos. X es un answer set de P si este es un answer set de la reducción P^x [13].

Sea P un programa, $M \subseteq LP$, M answer set de P y LP un conjunto de átomos:

$M \cup \neg\{L_P - M\}$ es el modelo de P [13].

Definición 2.6 Dado un conjunto de literales A y P un programa. Denotamos $\neg A = \{\neg a \mid a \in A\}$. También definimos $A^* = A \cup \neg\{Lit_P - M\}$ [12].

Definición 2.7 Decimos que dos reglas r1 y r2 están en conflicto (denotado por $r_1 \nabla \sim r_2$) sí y sólo si ambas tienen la misma cabeza, pero una es la contrapuesta de la otra, es decir, si H(r1) = -H(r2) [13].

Definición 2.8 (Principio de rechazo causal). Sean P_1 y P_2 programas y sea S answer set de la actualización de P_1 y P_2. Definimos y denotamos al conjunto de reglas a rechazar como:

$Rej(S, (P_1, P_2)) = \{r \in P_1 \mid \exists\, r' \in P_2$, tal que $r \nabla r'$ y $S \models B(r) \cup B(r')\}$ [13]

2.7 Cálculo de Answer Set

Como vimos en la sección 2.5 para cada programa *p,* los conjuntos de respuestas se define usando el campo de instanciación (*P*) en dos pasos: primero se definen los conjuntos de respuestas de los programas disyuntivos positivos, y luego los conjuntos de respuestas de los programas generales se definen mediante una reducción a programas disyuntivos positivos y una estabilidad condición. Después se tiene la reducción de "*Gelfond–Lifschitz transform"* de un programa *P* [9].

Los sistemas no-monotónicos tienen la habilidad de introducir nuevos axiomas retrayendo teoremas existentes. Esto se ha conseguido gracias al uso e inclusión de la negación como falla dentro de las cláusulas. La negación por falla (Negation as failure, NAF) permite expresar restricciones, excepciones y realizar deducciones a partir del conocimiento incompleto [14].

La semántica del modelo estable es unas de las más aceptadas ya que provee la semántica adecuada para los programas lógicos con NAF. La semántica del modelo estable permite aceptar múltiples modelos mínimos (A*nswer Set*) a diferencia de la lógica clásica que solo aceptaba un único modelo [14].

Esto ha llevado a ASP como un paradigma novedoso que permite obtener una colección de soluciones a través de cláusulas lógicas.

2.8 Ejemplos de cálculo Answer Set

Ejemplo 2.2: Sea P_1 el programa definido por las siguientes reglas:

$$p \leftarrow p$$

$$q \leftarrow \neg q$$

De acuerdo con la definición 2.5 de Answer Set, concluimos que el único que existe para P_1 es {q}.

Ejemplo 2.3: Ahora, consideremos el siguiente ejemplo.

$$P: a \leftarrow \neg a$$

Supongamos que $X = \varnothing$, aplicando definición 2.5 a P temenos:

$$P^x: a^x \leftarrow (\neg a)^x$$

Del cual resulta el programa siguiente:

$$P^x: a \leftarrow T$$

Por lo tanto, $X = \varnothing$ no es un answer set de P.

Ejemplo 2.4: El siguiente es un programa lógico no tradicional (tiene negación como falla, NAF)

$$p \leftarrow \text{not } q$$

$$q \leftarrow \text{not } r$$

En el ejemplo 2.4, si X= {q}, el reducto de π^X consiste en un solo hecho, q. El "answer set" de π^X es {q} y consecuentemente {q} es un "answer set" de π, y {q} es el único "answer set" de π.

Ejemplo 2.5: Considere el siguiente ejemplo.

$$\text{Sea P: } \neg b \leftarrow \neg a$$

Veamos si $X = \varnothing$ es un answer set de P

P^x: $T \leftarrow T$

Por lo tanto, $X = \varnothing$ es un modelo (answer set) de P

Ejemplo 2.6 Consideremos el siguiente ejemplo.

Sea P: $a \leftarrow \neg\neg a$.

$\neg b \leftarrow c \vee b$.

Propongamos que nuestro answer set sea $X = \varnothing$ entonces, aplicando la reducción a nuestro programa P para verificar si lo satisface.

P^x: $a^x \leftarrow \neg\neg a^x$

$\neg b^x \leftarrow (c \vee b)^x$

P^x: $a \leftarrow T$

$T \leftarrow c \vee b$

Así, $X = \varnothing$ no satisface a P^x por lo tanto X no es un answer set de P

Si $X = \{a\}$

P^x: $a^x \leftarrow \neg\neg a^x$

$\neg b^x \leftarrow (c \vee b)$ *

P^x: $a^x \leftarrow T$

$T \leftarrow c \vee b$

por lo tanto, X es un answer set de P.

Ejemplo 2.7 Consideremos el siguiente ejemplo.

Sea P: $a \leftarrow \neg b$

$b \leftarrow \neg a$

$p \leftarrow \neg p$

$p \leftarrow \neg a$

Sea $X = \emptyset$, trataremos de ver si este es un answer set de P

P^x: $a \leftarrow T$

$b \leftarrow T$

$b \leftarrow T$

$b \leftarrow a$

por lo tanto, $X = \emptyset$ no es un answer set de P.

Ahora, sea $X = \{a\}$

P^x: $a \leftarrow T$

$b \leftarrow \perp$

$b \leftarrow T$

$b \leftarrow a$

por lo tanto, $X = \{a\}$ no es un answer set de P.

Ahora veamos si $X = \{b\}$ es un answer set de P

P^x: $a \leftarrow \perp$

$b \leftarrow T$

$b \leftarrow T$

$b \leftarrow a$, por lo tanto, $X = \{b\}$ no es un answer set de P

Sea $X = \{p\}$

P^x: $a \leftarrow T$

$b \leftarrow T$

$b \leftarrow \perp$

$b \leftarrow a$

por lo tanto, X = {p} no es un answer set de P

Sea X = {p, a} y busquemos probar que es answer set de P

P^x:$a \leftarrow T$

$b \leftarrow \perp$

$b \leftarrow \perp$

$b \leftarrow a$

por lo tanto, X = {p, a} si es un answer set de P

Finalmente, veamos si X = {p, b} también es answer set de P

P^x: $a \leftarrow \perp$

$b \leftarrow T$

$b \leftarrow \perp$

$b \leftarrow a$

por lo tanto, X = {p, b} no es un answer set de P

2.9 Intuicionismo Lógico

Siendo de las lógicas más importantes la cual se basa en los conceptos de *prueba* o *conocimiento* a diferencia de la lógica clásica que se centra en la verdad y es más restrictiva que la lógica intuicionista [14].

Se utiliza el término *lógica intermedia* para denotar a todas las lógicas obteniendo una clásica tautología bajo *modus ponens* y una sustitución proposicional, que contienen todos los teoremas intuicionistas.

2.9.1 Lenguaje de la Lógica Intuicionista

El cual consta de los siguientes elementos:

- Alfabeto proposicional que requiere de los siguientes elementos para ser conformado:
    - Variables proposicionales.
    - Conectivas lógicas las cuales pueden ser de los siguientes tipos:
        - Constantes.
        - Binarias.
    - Símbolos auxiliares.
- Fórmulas proposicionales.

Lo que lleva a que la conjunción, disyunción y bicondicional se pueden ver como abreviaturas de fórmulas con las conectivas anteriores. Al igual que la constante T tendrá el significado de verdadero [15].

Lo que nos da como resultado la siguiente definición:

Definición 2.9 El lenguaje del cálculo intuicionista proposicional es el mismo que en la lógica clásica [15]:

Variables proposicionales, conectivos lógicos: conjunción $\wedge$, disyunción $\vee$, condicional $\rightarrow$, símbolos auxiliares como los paréntesis que nos ayudan a saber el alcance de los conectivos lógicos y la negación clásica "not" que se denota por $\neg$.

Modus ponens como la única regla de inferencia y el siguiente esquema de axiomas:

1. $A \rightarrow (B \rightarrow A)$
2. $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$
3. $A \wedge B \rightarrow A$
4. $A \wedge B \rightarrow B$

5. $A \rightarrow (B \rightarrow (A \wedge B))$

6. $A \rightarrow (A \vee B)$

7. $B \rightarrow (A \vee B)$

8. $(A \rightarrow C) \rightarrow ((B \rightarrow C) \rightarrow ((A \vee B) \rightarrow C))$

9. $(A \rightarrow B) \rightarrow ((A \rightarrow \neg B) \rightarrow \neg A)$

10. $\neg A \rightarrow (A \rightarrow B)$

Capítulo 3 **DLV: Disjunctive Datalog System**

Como se mencionó en el capítulo 2 se pudo ver como Answer Set Programming (ASP) ha demostrado que es un paradigma de programación importante con un alto impacto en la programación declarativa y en la representación del conocimiento completo o incompleto. Ya que Answer Set Programming tiene fuertes fundamentos teóricos ha hecho que todo lo anterior se logre esto también incluye la complejidad y el alto poder de expresión. Así también ha llevado relación con el paradigma de la programación de restricción. Con todo esto se han obtenido aplicaciones que demuestran que los resultados obtenidos son bastante competitivos si lo comparamos con otros lenguajes de programación como Java, C o C++.

DLV es un lenguaje declarativo lo que significa que en lugar de escribir un programa para resolver un determinado problema el programador escribe o especifica lo que puede ser la solución. Esto mediante un traductor el cual encontrará una solución eficiente al problema dadas las especificaciones del programador. Para encontrar dicha solución se requieren de los llamados “hechos y reglas”. Los hechos son datos que servirán de entrada y de las reglas se pueden obtener más hechos, a partir de estos se llegará a la solución óptima.

DLV siendo un sistema de bases de datos deductivo, basado en la programación lógica disyuntiva admite un lenguaje basado en formalismos lógicos, de modo que puedan representar problemas prácticos relevantes basados en el conocimiento ya sea completo o incompleto. El sistema en su primera versión ha estado disponible desde 1997 y se ha ido mejorando después de varios años de investigación teórica. El sistema admite un lenguaje basado en formalismos lógicos con un alto poder expresivo para que los programas puedan representar problemas prácticos relevantes. Ha tenido mejoras año con año y se han incorporado nuevas características y técnicas de optimización relevantes en todos los módulos [16]. Este sistema cuenta con front-ends para trabajar con aplicaciones específicas de Inteligencia Artificial. [17]

El sistema DLV ha tenido implementaciones sólidas ya que ha dedicado esfuerzo a algoritmos y diferentes técnicas para mejorar el rendimiento, por lo que se describe como un sistema para la programación de conjuntos de respuestas (ASP). [17]

La Figura 3.1 muestra la arquitectura del sistema DLV. Al inicio la entrada es especificada por el usuario la cual se analiza y se transforma en las estructuras de datos internas del

sistema. La entrada se puede leer desde archivos de texto, pero DLV proporciona una interfaz para bases de datos. [ref 2 cap 3 18].

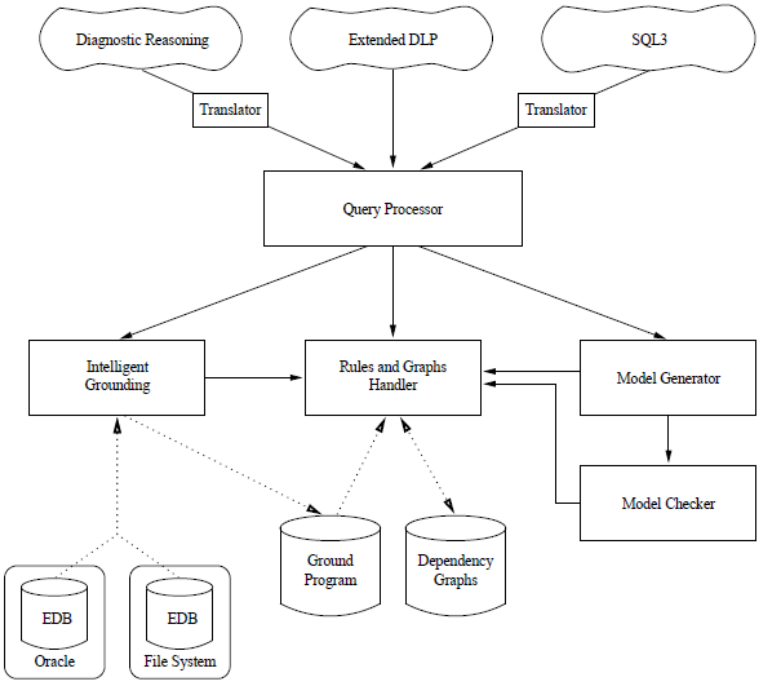

Figura 3.1 *Arquitectura de DLV*

3.1 Lenguaje base de DLV

DLV maneja un lenguaje de programación declarativo, lo que significa que como se mencionó en el punto anterior el programador no escribirá un programa para resolver determinado problema, sino que especificará lo que puede llegar a ser la solución.

Se darán a conocer los elementos básicos de DLV como las constantes, variables y términos. Con todos estos elementos se realizará la correcta escritura de un programa en DLV.

En DLV los elementos más básicos son las constantes, por lo tanto, a continuación, se darán los siguientes conceptos por definición [19]:

Definición 3.1 Las constantes deben comenzar con una letra minúscula y pueden tener otras letras, guion bajo y dígitos. También los números son constantes también.

Se debe tener en cuenta que *not* es una palabra reservada y not es una constante valida.

Definición 3.2 Las variables comienzan con una letra mayúscula y pueden tener otras letras, guion bajo y dígitos.

Existe una variable especial que es diferente a la descrita anteriormente y se llamada *variable anónima* denotada por "_" (guion bajo) que no ocurre en la misma regla respectivamente. Cada que aparece "_" representa una nueva y única variable, el propósito de estas variables es especificar que un argumento puede ser ignorado o no importa en la regla actual.

Ya que las variables anónimas representan variables únicas, no es conveniente usarlas en la cabeza de la regla o en el cuerpo de dicha regla como una literal negativa, ya que estas reglas no son seguras.

Definición 3.3 Un término simple es una constante o una variable.

Ejemplos de términos simples:

Constante: a1, aRt5, 1994, an_

Variable: Var, Xi_Y, C7r

Literal: m, ~cr(1, X), not sirve(X,1,roto) **[DLV-user Manual 16]**

Definición 3.4 Término complejo.

Un término complejo es un término funcional o un término de lista.

Un término funcional es un símbolo de función seguido de una lista de términos entre paréntesis. Un símbolo de función debe comenzar con una letra minúscula y puede estar compuesto de letras, guiones bajos y dígitos.

Un término de lista puede tener las dos formas siguientes:

[*t1* , ..., *tn*], es decir, una lista de términos entre corchetes;

[*h* | *t*], donde *h* (el encabezado de la lista) es un término, y *t* (la cola de la lista) es un término de lista.

Ejemplo de términos complejos:

Términos funcionales: f (1), g_1 (a), fUN (1, a, f (5)), f (X), f (a, f (gibon (X), 1, Y), Zulú).

Lista de términos: [a, [1,2], c, [2,3]], [1, f (1), 2, f (a)], [ ], [1 | [2,3]], [a | [b | [c]]], ["Zulu", [ ], f ([ ])].

Definición 3.5 Átomos

Una literal es un átomo. DLV a diferencia de otros sistemas permite dos tipos de negación: la negación verdadera (o explícita) y la negación como fallo. Escribiremos la negación verdadera con los siguientes símbolos – o ~. Mientras que la negación por fallo se escribirá con la palabra "not".

Ejemplos de átomos válidos:

a

b(8,k)

weight(X,1,kg)

Definición 3.6 Hechos.

Con fines de simplicidad diremos que un hecho se trata de una proposición y lo que encontramos dentro del paréntesis serán los argumentos.

De manera sintáctica se escribirá primero el nombre de la proposición comenzando con una letra minúscula seguido de los argumentos dentro de un paréntesis, los argumentos estarán separados por comas, al final del hecho deberá ir un punto.

Ejemplos de hechos válidos:

weigth(apple,100, gram).

-valid(1, equals, 0).

Definición 3.7 Como definir comentarios.

Como en cualquier lenguaje de programación DLV también permite al programador colocar comentarios entre las líneas de código. Para esto, utilizaremos el símbolo "%" el cual nos denotará que es un comentario y no una línea de código.

Ejemplo de comentario válido:

%Esto es un comentario

% Esto también es un comentario

%----------Esto sigue siendo un comentario-----

3.1.2 Lenguaje básico de DLV

La construcción principal del lenguaje de DLV está basado en reglas las expresiones son de la forma Head :- Body, donde Body es una conjunción de literales y Head es una disyunción de átomos. De manera informal diremos que la expresión anterior podría leerse como: "Si el cuerpo es verdadero, entonces la cabeza es verdadera". Se le llama "hecho" a una regla sin cuerpo ya que ésta modela una verdad incondicional donde por simplicidad se omitirá ":-", mientras que una regla con una cabeza vacía es llamada "restricción fuerte" y es usada para modelar una condición que debe ser falsa en cualquier solución posible. A todo el conjunto de reglas se le llama "Programa". La semántica del programa es dada por los "*answer sets*" que se mencionaron el capítulo anterior. Un programa entonces podrá utilizarse para modelar la solución de un problema, las soluciones dependerán del conjunto de respuestas del programa (*answer sets)* y estos son calculados por DLV. Por lo tanto, un programa puede no tener un conjunto de respuestas si el problema no tiene solución, uno, si el problema tiene solución única o varios si el problema tiene más de una solución posible [20].

3.2 Base de datos y Programación lógica

El sistema DLV se ve como un sistema de programación lógica o un sistema de base de datos deductivo, en el mundo real a menudo el razonamiento se realiza en fuentes de datos existentes hablando en este contexto los sistemas de bases de datos deductivas muestras algunas limitaciones como las siguientes. [20]:

1. La cantidad de datos que pueden manejarse es limitada ya que la mayoría de ellos funcionan en la memoria principal.

2. La interacción con fuentes de datos externas (y autónomas), como las bases de datos, no es trivial y, en varios casos no es permitida.
3. La eficiencia de las soluciones existentes aún no es suficiente para su utilización en tareas de razonamiento complejo que involucran gran cantidad de datos.

DLV con ayuda de DLVDB tiene el objetivo de ayudar a superar estos inconvenientes.

En DLV existen dos tipos de bases de datos: la “Base de datos Extensional" BDE y la “Base de datos Intencional" BDI [19].

3.2.1 Base de datos Extensional

Ya que Datalog es el resultado de la combinación de la programación lógica y las bases de datos y de ahí el nombre se puede observar que DLV puede verse como un sistema de programación lógica o un sistema de bases de datos deductivo. Este tipo de bases de datos es una colección de hechos que la hace diferente o separa del tipo de base de datos Intencional de la cual hablaremos más adelante.

Algo que describe a las bases de datos extensionales es que se puede dar datos a un archivo, pero también se pueden importar hechos desde una base de datos relacional por medio del controlador ODBC [19].

Ejemplo 3.1 Mochila Binaria

La mochila binaria (0-1 *knapsack*) es un problema típico de programación dinámica. Éste consiste en llenar una mochila con varios objetos, cada objeto (*Object*) tiene un peso (*weight*) y una ganancia (*profit or val*), el peso de los objetos elegidos no puede exceder la capacidad máxima de la mochila y la suma de sus ganancias debe ser óptima. La formulación del problema es el siguiente [21]:

$$\mathrm{Ks(n,k)} \;=\; \text{máx}\begin{cases} 0 & ,n=0 \\ ks(n-1,k) & ,1 \le n \\ profit(n)+kd\big(n-1,k-weight(n)\big) & ,weight \le k \; y \; 1 \; \le n \end{cases}$$

Donde *n* es el número de objetos y *k* es la capacidad de la mochila.
Con esta formulación no se recupera que objetos se seleccionan, solo la ganancia óptima. A continuación, se muestra la solución en DLV con la EDB.

```
%Declaración de los objetos y de sus características
%nombre, val, costo
Obj(a,4,2).
Obj(b,3,7).
Obj(c,2,5).
%Definir la capacidad de la mochila
capacidad(10).
%formación de los posibles conjuntos de objetos para introducir en la mochila
in_sack(X,V,C) :- obj(X,V,C), not not_in_sack(X,V,C).
not_in_sack(X,V,C) :- obj(X,V,C), not in_sack(X,V,C).

%midiendo que el valor total no sobre pase la cantidad de la mochila
s1 :- capacidad(K), #sum {C : in_sack(X,V,C)} <=K.
:- not s1.
%maximizar el número de elementos dentro de la mochila o
%minimizar el número de elementos que no están dentro de la mochila
:~ not_in_sack(X,V,C). [V:1]
```

Salida:

> dlv -nofacts knapsackConAset.dl

DLV[build BEN/Jun 26 2003 gcc 2.95.3-10 (cygwin special)]

*Best model: {**in_sack(a,4,2), in_sack(b,3,7),** not_in_sack(c,2,5), s1}*

Cost ([Weight:Level]): <[2:11]>

El programa nos dio el modelo con los objetos 1 y 2 también podemos observar que DLV nos proporciona el modelo óptimo para la solución del problema dado.

3.2.2 Base de datos Intencional

El conocimiento que hará uso de estas bases de datos puede depender de la BDE. Esta parte de la entrada se le conoce como BDI o programa en algunas ocasiones [19].

Las reglas siguen esta forma:

h1 v… v hn :- b1, … , bm.

De h1 a hn representa literales explícitamente negadas, donde n > 0, es decir al menos una de ellas debe existir. ":-" este símbolo representa la implicación y b1, … , bm representa las literales, donde m >= 0, es decir el cuerpo debe omitirse. Lo que esta antes de ":-" se refiere a la cabeza y lo que esta después de esos símbolos es el cuerpo. La negación como falla solo ocurre en el cuerpo y además las reglas deben ser seguras.

Si el cuerpo es evaluado como verdadero, la cabeza también se evalúa de la misma forma. La negación como falla puede leerse como "no hay evidencia de que xxx evalúe a verdadero", como en la lógica de default. También la cabeza representa una disyunción y el cuerpo una conjunción.

Reglas y hechos

Se tiene almacenado el archivo *alarm*. *hot_furnace* especifica que el horno está caliente, *valve_closed* define que la válvula de presión está cerrada, *alarm_on* debe existir [19].

La siguiente construcción es una regla:

```
hot_furnace. valve_closed.

alarm_on :- hot_furnace, valve_closed.
```

Ejemplo 3.2 Reglas:

```
-ok :- not -hazard.
male(X) ∨ female(X) :- person(X).
fruit(P) ∨ vegetable(P) :- plant_food(P).
true ∨ false :- .
employee(P) :- personnel_info(_,P,_,_,_).
```

Los símbolos ":-" (implicación) pueden leerse como "si", y la coma se lee como "y", lo que esta después de la implicación es el cuerpo y lo anterior a esta es la cabeza.

3.3 Negación y suposición de un mundo completo

La negación es tratada como negación como falla. Esto quiere decir que, si un átomo no es cierto en algún modelo, entonces la negación se considera verdadera en ese modelo [19].

Un ejemplo de esto es el siguiente:

```
node(X) :- arc(X,_).

node(Y) :- arc(_,Y).

comparc(X,Y) :- node(X), node(Y), not arc(X,Y).
```

donde la regla comparc es el conjunto de arcos en el grafo. Un arco debe ir de un nodo a otro o posiblemente al mismo, y este arco no está contenido en el conjunto de nodos original. Node(X) y node(Y) deben estar incluidos en el cuerpo en orden para que satisfaga los siguientes requerimientos de seguridad para las reglas. Variables, las cuales ocurren en una literal negada, deben también ocurrir en una literal positiva en el cuerpo.

3.4 Negación verdadera

Otra implementación de noción es la negación verdadera [19].

Anteriormente se habla de la negación como falla no soporta aseveración explicita. En otras palabras, si no hay evidencia de que un átomo es verdadero, este es considerado como falso.

Hay ocasiones donde la negación como falla no es apropiada porque se necesita que algunas cosas se conozcan como falsas. Por esta razón, la negación verdadera es algunas veces referida como *negación explícita*.

La negación verdadera se denota presidiendo un átomo con el símbolo "-" o con "~".

Ejemplo 3.3

Imagine una simple situación, en la cual un agente tiene que cruzar una vía de ferrocarril. El agente deberá cruzar esta si ningún tren se aproxima. Con esta descripción, uno puede especificarlo a través del siguiente programa:

```
cruza_vía :- not tren_aproxima.
```

Dado que no se tiene ninguna otra información, tren_aproxima no sucederá, y cruza_vía será derivado, es decir, si ejecutamos ese programa tendremos:

```
$ DLV -silent rail_naf
{cross_railroad}
```

Pero esto es un problema, ya que la regla indica que el agente cruce, lo que puede causar que el agente sea arrollado por el tren si este se aproxima dado que el agente cruzará (según indica el programa).

La forma de resolver este tipo de situaciones es a través de la negación verdadera, la cual nos permite modelar este mismo problema de la manera siguiente:

```
cruza_vía :- -tren_aproxima.
```

En este caso, -tren_aproxima puede ser considerado como un átomo separado, y dado que no hay información de que esto suceda, cruza_vía no puede ser derivado, es decir, si ejecutamos el programa este dará:

```
$ DLV -silent rail_naf
{ }
```

3.5 Restricciones de Integridad

Las restricciones son formulaciones de posibles inconsistencias. Esto es de bastante utilidad en conexión con reglas disyuntivas. Las reglas disyuntivas ayudan como generadores para modelos diferentes y las restricciones se usan para seleccionar solo lo deseado [19].

La sintaxis de las restricciones es simple ya que es similar a la de las reglas, pero estas no tienen cabeza, se dice que es similar como con las reglas, porque las restricciones deben reunir requerimientos de seguridad.

Ejemplo 3.4 El problema consiste en colorear un mapa que es representado por un grafo. La restricción es, que dos nodos conectados por un arco no pueden tener el mismo color.

Para formular este problema con reglas DLV, reconsideraremos algunas de las reglas previas.

nodo(X) :- arco(X,_).

nodo(Y) :- arco(_,Y).

color(X,rojo) v color(X,verde) v color(X,azul) :- nodo(X).

Los modelos (respuestas) de este programa corresponden a todas las posibles combinaciones de coloreado. Entonces, nosotros solo necesitamos agregar una restricción la cual descartará las combinaciones donde dos nodos adyacentes tengan el mismo color. Esto corresponde a la siguiente restricción.

:- arcoX,Y), color(X,C), color(Y,C).

Esta restricción nos brindará el resultado deseado, ya que eliminará las combinaciones que no deseamos (nodos adyacentes no pueden ser del mismo color).

3.6 Restricciones Débiles

Las restricciones débiles permiten formular problemas de optimización de manera fácil y natural. Mientras que las restricciones de integridad siempre deben estar satisfechas se satisfacen si es posible, y si se viola alguna de ellas no afecta a los modelos [19].

Los answer sets de un programa con un conjunto de restricciones débiles son aquellos answer sets del programa los cuales minimizan el número de restricciones violadas. A esto se le considera como los mejores modelos del programa y el conjunto de restricciones. Un programa no solo tiene un mejor modelo si no que puede tener varios mejores modelos. La representación sintáctica de las restricciones débiles es la siguiente:

:~ Conj. [Weight:Level]

Conj, es una conjunción de literales que pueden estar negadas, Weight y Level son enteros positivos.

Ejemplo 3.5 Considere el siguiente programa:

```
a v b.
c :- b.
:~ a.
:~ b.
:~ c.
```

Dado que los pesos y niveles de prioridad son omitidos, sus valores son asignados a 1 por default. Si ejecutamos este programa en DLV, obtenemos la siguiente salida.

```
$ DLV -silent example1
Best Model: {a}
Cost ([Weight:Level]): <[1:1]>
```

Note que los answer sets de { a v b. c :- b. } son {a} y {b, c}. La presencia de restricciones débiles descarta {b, c} porque este viola dos restricciones (mientras que {a} viola solo una restricción).

3.7 Predicados

Existen predicados definidos por el sistema además de los predicados definidos por el usuario. Estos predicados están definidos y disponibles en DLV para todos los programas. Tienen nombres que los usuarios pueden definir y algunos que no. Los átomos pueden hacer uso de los predicados y estos también se pueden negar con la negación por falla de la que se habló anteriormente [19].

3.7.1 Predicados Comparativos

Existen predicados incorporados en DLV lo que es muy útil ya que las constantes se pueden comparar con ayuda de estos.

<,>, <=,> =, = (con == como alternativa en desuso),! =

Ejemplo:

```
in_range(X,A,B) :- X>=A, <(X,B).
pair(X,Y) :- Y>X, color(X,green), color(Y,green).
```

En el ejemplo, se garantiza que el par define una relación asimétrica. Es decir, si el par (A, B) se mantiene, el par (B, A) no lo hace. Como consecuencia directa, no existe A , de modo que se mantiene el par (A, A) .

Todas las variables que aparecen en predicados comparativos están obligadas a satisfacer una condición de seguridad [19].

3.8 Aritmética de predicados en DLV

Existen en DLV predicados que ayudan al razonamiento y el cómputo de un conjunto finito de rangos enteros, estos predicados son: #int, #succ, #prec, #mod, +, *, -, /.

Los operadores aritméticos se usan si la aritmética entera es turnada a activa dando un número límite N sobre la línea de comando. DLV solo trabaja con números enteros, es decir, mayores o iguales que cero y menores o iguales que la N definida en la línea de comando.

Ejemplo 3.4 Definamos el predicado de Fibonacci.

```
fibonacci(N,F) :-   #succ(N2,N1),
                    #succ(N1,N),
                    fibonacci(N1,F1),
                    fibonacci(N2,F2),
                    +(F1,F2,F).
```

Como podemos observar, la manera en que se formula esta definición es literalmente escrita de manera directa sobre DLV [19].

Ejemplo 3.5 La división y módulo no han sido implementados aún. Sin embargo, estos pueden ser fácilmente simulados usando las primitivas existentes como sigue:

```
div(X,Y,Z) :- XminusDelta = Y*Z,
              X = XminusDelta + Delta,
              Delta < Y.

mod(X,Y,Z) :- div(X,Y,XdivY),
              XminusZ = XdivY * Y,
              X = XminusZ + Z.
```

Note que, al contrario de la construcción de funciones, la adición de estas reglas predicado div y mod en el modelo (esto es, si #maxint es 100, entonces cada answer set del programa contendrá 100*100 hechos para ambos div y mod).

3.9 Listar predicados

Hay varios predicados disponibles para razonar y computar sobre los términos de la lista [19]:

#append, #delnth, #flatten, #getnth, #head, #insLast, #insnth, #last, #length, #member, #reverse, #subList, #tail

Todas las listas integradas (excepto #member y #subList) tienen varios argumentos de entrada y exactamente un argumento de salida. El argumento de salida es siempre el último y su valor se calcula a partir de los argumentos de entrada. Debe ser una variable o un término constante, es decir, no puede ser un término complejo que no sea constante. Los elementos incorporados #member y #subList no producen ningún resultado, solo realizan una comprobación de sus argumentos de entrada.

Por mencionar algunos elementos de esta lista se tienen los siguientes:

#Last (X , Y) es verdadero si y sólo si Y es el último elemento de X . X debe ser un término de lista.

#length (X , Y) es verdadero si y sólo si Y es el tamaño de X . X debe ser un término de lista.

#member (X , Y) es verdadero si y sólo si X es un miembro de Y . X puede ser cualquier término, mientras que Y debe ser un término de lista.

#reverse (X , Y) es verdadero si y sólo si Y es una lista obtenida de revertir X . X debe ser un término de lista.

#subList (X , Y) es verdadero si y sólo si X es una lista secundaria de Y . X e Y deben ser términos de lista.

#tail (X , Y) es verdadero si y sólo si Y es la lista que contiene todos los elementos, pero el primero de X . X debe ser un término de lista.

Ejemplo: Lista de predicados.

newList (X): - #append ([a, b, c], [d, e], X).

newList (X): - #delnth ([a, b, c], 1, X).

flattenedList (X): - #flatten ([a, b, [c, [d]]], X).

anElemento (X): - #getnth ([a, b, c], 2, X).

headElement (X): - #head ([a, b, c], X).

newList (X): - #insLast ([a, b, c], d, X).

newList (X): - #insnth ([a, b, c], d, 4, X).

lastElement (X): - #last ([a, b, c], X).

tamaño (X): - # longitud ([a, b, c], X).

Sí: - #miembro (c, [a, b, c]).

reversedList (X): - #reverse ([a, b, c], X).

sí: - #subList ([c], [a, b, c]).

tailList (X): - #tail ([a, b, c], X).

3.10 Constantes incorporadas

#maxint es igual al límite entero superior (como se indica en la línea de comandos con -N). El uso de #maxint si no hubo un rango entero de resultados definidos en un error [19].

Además de la opción de línea de comando -N , #maxint también se puede configurar directamente en la entrada, como en el siguiente ejemplo:

```
# maxint = 19.

bignumber (#maxint).
```

3.11 Constantes nombradas

Un nombre puede ser asignado a una constante. Esto puede ser útil cuando la misma constante se usa muchas veces en su programa y desea cambiar su valor [19].

Para definir una nueva constante nombrada use la siguiente sintaxis:

```
#const namedConstant = constante.
```

donde namedConstant es un nombre constante (debe comenzar con una letra minúscula y puede estar compuesto de letras, guiones bajos y dígitos) y constante es cualquier término constante legal. Siguiendo esta definición, todas las apariciones de namedConstant se reemplazan con constante. La definición de un nuevo valor para una constante nombrada ya definida no está permitida.

Ejemplo: Constante de nombre numérico.

```
#const rate = 5.

due(2). due(10).

pay(X) :- due(Y), X=Y*rate.
```

La salida será la siguiente:

```
$ DLV -silent -N=50 numconst
```

{due(2), due(10), pay(10), pay(50)}

3.12 Funciones agregadas

Como en varios lenguajes de programación DLV también incluye una lista de funciones que el usuario podrá utilizar en cualquier programa para los fines que este requiera [19].

Una función agregada se aplica al conjunto simbólico y devuelve un número.

Ejemplo:

La función agregada #count devuelve la cardinalidad del conjunto simbólico al que se aplica.

#count {<1, d (1,3)>, <3, d (3,3)>}

el valor devuelto es 2 .

Así como existe #count, también podemos utilizar #sum, #times, #min y #max.

Existe otro conjunto de características igual de importantes que las tratadas en este capítulo, sin embargo, invitamos al lector a referirse a los documentos originales que pueden ser accesados en la siguiente dirección electrónica con acceso libre, http://www.dlvsystem.com/html/DLV_User_Manual.html.

3.13 Comparación de implementaciones

Podemos observar que en answer set programming la solución a los problemas se representan por “answer sets” que son los llamados conjuntos de respuestas lo cual es completamente diferente a las respuestas que produce un query como se puede ver en la lógica convencional. Algunos sistemas hacen posible que el paradigma ASP llegue a nivel se software, algunos de estos sistemas son smodels, cmodels y DLV. Así, DLV es el sistema de software que hemos elegido sistema formal para modelar nuestro problema de tesis.

DLV como ya pudimos observar puede ser utilizado como implementación para resolver problemas de representación declarativa. En el presente trabajo de tesis se ha empleado DLV como nuestro lenguaje principal ya que en él podemos realizar la representación del conocimiento relacionado al ejemplo del mundo de los bloques. De manera particular, hemos

representado la forma en la que los bloques serán acomodados. Este tipo de problemas son clásicos en diversas áreas del conocimiento, tal como se verá en el siguiente capítulo.

Capítulo 4 **Planificación y control de rutas para estacionamientos tipo pila de alta densidad**

En el capítulo anterior se pudo dar un panorama general de DLV el cual como ya vimos es un lenguaje declarativo lo que significa que en lugar de escribir un programa para resolver un determinado problema el programador escribe o especifica lo que puede ser la solución. Esto mediante un traductor el cual encontrará una solución eficiente al problema dadas las especificaciones del programador. Para encontrar dicha solución se requieren de los llamados "hechos y reglas". Los hechos son datos que servirán de entrada y de las reglas se pueden obtener más hechos, a partir de estos se llegará a la solución óptima. En la actualidad DLV tiene varios *front-end* para trabajar en diferentes áreas.

Uno de ellos está basado en el lenguaje de planificación *k* el cual está basado en principios y métodos de la programación lógica. Recibió el nombre de *k*, ya que este describe las transiciones entre los estados del conocimiento en vez de entre los estados del mundo. Algunos lenguajes de programación como C realizan esto último. *K* es un lenguaje flexible por lo que es completamente adecuando para resolver problemas difíciles de planificación [22].

4.1 Planeación

Por definición se dice que a través de la planeación una persona u organización se fijan metas u objetivos que. Mediante una secuencia de pasos se llegará a ellos. Es en este proceso que puede tener una duración variable, donde serán tomadas a consideración diversas cuestiones como pueden ser los recursos con los que se cuentan entre las diversas situaciones externas.

"Toda planeación consta de distintas etapas, ya que es un proceso que supone tomar decisiones sucesivas. Es frecuente que la planificación se inicie con la identificación de un problema y continúe con el análisis de las diferentes opciones disponibles. El sujeto o la compañía deberá escoger la opción que le resulte más propicia para solucionar el problema en cuestión e iniciar la puesta en marcha de un plan." https://definicion.de/planeacion/

Para llegar al uso y motivación de planeación en este presente trabajo de tesis es que debemos considerar que en todo momento nos encontramos planeando actividades, acciones, etc. Por

ejemplo, al levantarnos de la cama en muchas ocasiones planeamos que es lo que vamos a desayunar, cuál será el camino que tomaremos para el trabajo o escuela.

Todas esas acciones nos motivaron y a continuación hablaremos de la planeación, pero ahora en nuestro contexto y como es que se utilizó dentro de nuestro caso de estudio.

En planeación, dada una descripción del mundo, una situación inicial y una situación deseada, la meta es hallar una secuencia de acciones (las cuales pueden cambiar las situaciones), tal que la situación deseada es alcanzada desde el estado inicial. Además, no todas las acciones son aplicables en todas las situaciones, es decir, que las acciones no sean ejecutadas de manera secuencial como es esperado y éstas tengan que ejecutarse de manera diferente para ello recurriremos a la re-planeación la cual describiremos más adelante.

En la Figura 4.1 podemos observar de forma sencilla la representación de la planeación donde podemos observar que se cumple lo antes mencionado, de un estado inicial llegamos al estado final o situación meta a través de una secuencia de acciones.

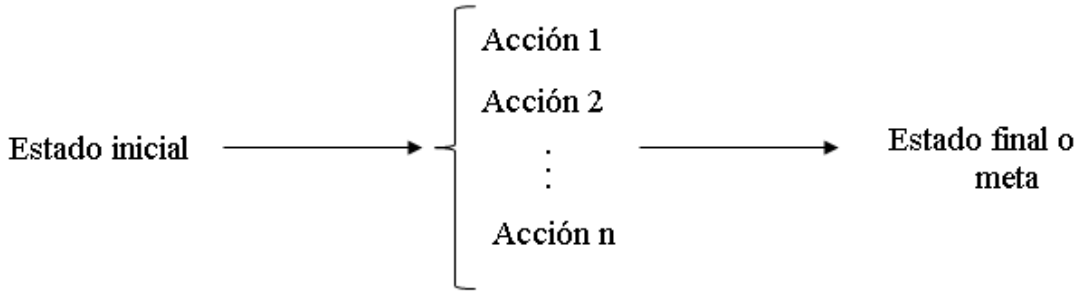

Figura 4.1 Representación de Planeación

Definición 4.1 Planeación es un proceso de toma de decisiones para alcanzar un futuro deseado, teniendo en cuenta la situación actual, los factores internos y externos que pueden influir en el logro del objetivo [23].

Definición 4.2 Un problema de planeación es una cuádrupla < ***F, A, I, G*** > donde:

- ***F*** es un conjunto de fluentes, los cuales caracterizan las situaciones

- ***A*** es un conjunto de acciones, con una definición de sus respectivas precondiciones y efectos o causas.
- ***I*** es un conjunto de fluentes describiendo la situación inicial
- ***G*** es un conjunto de fluentes describiendo la situación meta o deseada

4.1.1 Planeación Clásica

Un problema de planeación clásico consiste en los siguientes tres elementos:

- Estado inicial.
- Acciones que serán las que cambien los estados.
- Estado final (objetivo o meta).

El objetivo de la planeación clásica es encontrar un método efectivo para obtener un plan, es decir, que a través de una secuencia de acciones se llegue a un estado meta o final aplicadas desde el estado inicial. Los problemas de planeación pueden ser vistos como un problema de búsqueda de rutas en un grafo, de donde los nodos pueden verse como los estados posibles en el mundo que se está trabajando y las aristas pueden ser las transiciones posibles con las acciones disponibles [24].

Existen muchos problemas que pueden ser expresados en la planeación clásica, por ejemplo, un problema de logística que involucra la recolección y entrega de paquetes se puede modelar de la siguiente manera:

- La situación inicial describe la ubicación inicial de los paquetes, camiones, aviones. Las acciones incluyen cargar y descargar un paquete desde un tren o un avión, y mover camiones y aviones entre ubicaciones y ciudades.
- El objetivo codifica la posición final deseada de los paquetes.
- Un planificador clásico usa tal codificación del problema para devolver una secuencia de acciones que finalmente entregan los paquetes a sus destinos deseados

Uno de los enfoques exitosos de la planeación clásica es intentar encontrar un plan de N pasos que al crear un conjunto de proposiciones contenga dichos planes [24].

Si bien la mayoría de los sistemas existentes se basan en lenguajes de planificación "clásicos" los últimos años se ha visto el desarrollo de lenguajes de acción que proporcionan herramientas expresivas y flexibles para describir la relación entre los fluentes y las acciones. Los lenguajes de acción han recibido considerable atención en la comunidad de representación del conocimiento y razonamiento por sus propiedades formales (complejidad, etc) que ha sido estudiadas a profundidad. El esfuerzo que se ha dedicado de cómo utilizar las herramientas brindadas por estos sistemas han sido menores [6].

A continuación, hablaremos de un lenguaje de acción utilizado para el desarrollo del presente trabajo de tesis

4.2 Lenguaje de Acción *k*

Es un poderoso lenguaje para la planificación bajo conocimiento completo e incompleto. Se le llama *k* para hacer énfasis a que describe las transiciones entre los estados del conocimiento en lugar de entre los estados del mundo. El lenguaje es muy flexible capaz de modelar transiciones entre estados del mundo (conocimiento completo) y razonar sobre ellos como un caso particular. Comparado con otros lenguajes de planeación, *k*, está más cerca a la esencia de la semántica de *"answer set"* que a la de la lógica clásica. Permite el uso de la negación por falla, explotando el poder de los *"answer set*" (conjunto de respuestas) para tratar con el conocimiento incompleto [25].

El sistema de planificación DLV^k, proporciona una implementación del lenguaje *K* como front-end del sistema DLV [26]. Para el presente trabajo de tesis se utilizó DLV^k que es un poderoso sistema de planificación que se encuentra disponible de manera gratuita en http://www.dlvsystem.com

4.3 DLV^k

En la actualidad surge la necesidad de modelar el comportamiento de robots de una manera formal, esto condujo a la definición de lenguajes basados en la lógica para razonar sobre las acciones y planificación. Estos lenguajes permiten buscar una secuencia de acciones que dado un estado inicial se llegue a un estado final u objetivo. La mayoría de estos lenguajes se basan en extensiones lógicas clásicas y como habíamos mencionado antes describen transiciones entre los estados del mundo donde cada "*fluent"* es necesariamente verdadero o falso. Sin embargo, un robot no tiene una visión completa del mundo. Incluso si su

conocimiento es incompleto (cierta cantidad de *"fluents"* puede ser desconocida, por ejemplo, si una puerta se encuentra abierta), deben tomarse decisiones, ejecutar acciones y razonar sobre la base de su información (incompleta) disponible. Cuando el robot se encuentra en una situación desconocida se deberá actuar, por ejemplo, si se encuentra frente a una puerta que se encuentra cerrada este podría detenerse o retroceder [26].

DLV^k es un sistema basado en DLV que implementa el lenguaje k para resolver problemas de planeación. Se ha nombrado *k* para dar énfasis a las transiciones entre los estados de conocimiento en lugar de entre los estados del mundo. Comparado con otros lenguajes de programación *k* es uno de los sistemas que mejor implementan ASP. Para ayudarnos a entender cómo es que podemos trabajar con DLVk recurrimos a un ejemplo clásico llamado *"el mundo de los bloques"* el cual nos permitirá explicar de manera breve y sobre todo clara como representar un problema de planeación en este lenguaje

DLV^k a diferencia de otros sistemas de planificación basados en lógica cuenta con las siguientes características [25]:

- Conocimientos previos explícitos: el dominio de planificación tiene antecedentes (representados por un programa estratificado de registro de datos) que describe predicados estáticos.
- Declaraciones de tipos: se escriben los argumentos de predicados cambiables, llamados fluidos, y átomos de acción.
- Negación fuerte y débil: el sistema DLV^K proporciona dos tipos de negación familiares de la semántica del conjunto de respuestas, a saber, la negación débil (o predeterminada) "no" y la negación fuerte (o clásica) "¬", también denotada por "-". La negación débil permite una declaración simple e intuitiva de reglas de inercia para fluidos, o para la declaración de valores predeterminados para fluidos en el dominio.
- Estados completos e incompletos: de manera predeterminada, los estados en DLV^K son conjuntos consistentes de literales básicos, en los que no todos los átomos deben aparecer y, por lo tanto, representan estados de conocimiento. Sin embargo, mediante construcciones adecuadas, DLV^K también permite representar transiciones entre posibles estados del mundo (que pueden verse como estados de conocimiento completo).

- Ejecución paralela / secuencial de acciones: es posible la ejecución simultánea de acciones, y de hecho el modo predeterminado. Todas las acciones para ejecutar deben cumplir con una condición de ejecución. La exclusión mutua de acciones puede hacerse cumplir en un modo de planificación secuencial.
- Planificación segura (conforme): DLV^K puede calcular planes seguros (a menudo llamados planes conformes en la literatura). Informalmente, un plan es seguro, si es aplicable a partir de cualquier estado inicial legal y hace cumplir la meta, independientemente de cómo evolucione el estado. Con esta función, también podemos modelar la planificación de mundos posibles con un estado inicial incompleto, donde el mundo inicial solo se conoce parcialmente, y estamos buscando un plan que alcance la meta deseada de cada mundo posible de acuerdo con el estado inicial.

4.3.1 Sintaxis de DLV^k

Para describir mejor la sintaxis del sistema DLV^k se tomará como ejemplo un problema clásico llamado "el mundo de los bloques".

Ejemplo 4.1 El mundo de los bloques (versión simple)

El mundo de los bloques o mejor conocido como el *"The blocks world problem"* es un problema clásico en Inteligencia Artificial. Imaginemos que sobre una mesa encontramos un conjunto de bloques de madera (o cualquier material). El objetivo es formar una pila con dichos bloques, el problema es que solo podrá moverse un bloque a la vez; estos solo podrán colocarse sobre la mesa o sobre otro bloque. Debido a estas condiciones, cualquier bloque que se encuentre en un momento dado debajo de otro bloque no puede moverse.

La Fig. 4.2 muestra cómo se encuentran inicialmente los bloques, en nuestra versión de este primer ejemplo de manera sencilla pensaremos que los bloques son acomodados solos, sin ayuda de ningún otro objeto. Se realizarán una secuencia de acciones que harán que los bloques queden apilados tal y como se muestra en la Fig. 4.3. Las figuras antes mencionadas nos muestran el estado inicial y el estado final de los bloques.

Figura 4.2 Estado Inicial

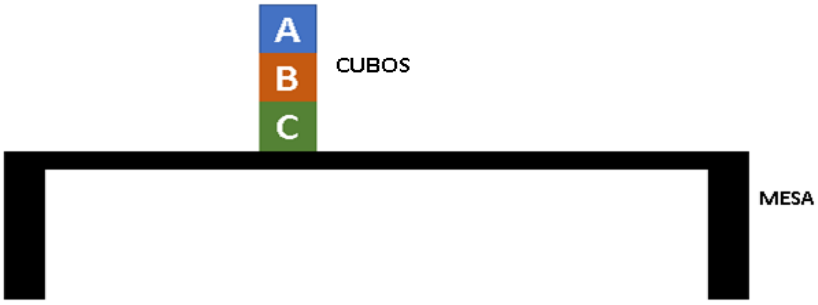

Figura 4.3 Situación final o meta

Dado el escenario anterior, se realiza el modelado de todos y cada uno de los elementos necesarios para que la solución del problema se lleve a cabo. Esos elementos tienen un nombre y son los siguientes:

- Fluentes.
- Acciones.
- Condiciones de ejecución.
- Reglas que describen las causas de las acciones *"Causation rules".*
- La situación inicial.
- La situación final o meta.

Gracias a todos y cada uno de estos elementos el sistema DLV^k encontrará un plan óptimo que contenga la solución a nuestro problema.

En nuestro problema del mundo de los bloques los elementos a modelar son los siguientes: los cubos y la mesa.

Dentro de nuestro programa utilizamos los elementos antes mencionados para encontrar un plan óptimo, describiremos cada uno de ellos y como es que son utilizados en nuestro ejemplo.

Fluentes: Son aquellos que nos permiten caracterizar la configuración del mundo en que se sitúa el problema (caracterizan el contexto). Los cuales se declaran de la siguiente forma:

fluents: p(X1, …, Xn) requires t1, …, tm

En nuestro ejemplo los fluentes son:

fluents: sobre(B,L) requires cubo(B), lugardisponible(L).

ocupado(B) requires lugardisponible(B).

Como se mencionó antes los fluentes están caracterizando la descripción que tiene cada uno de los elementos en nuestro mundo. El primer fluente nos describe que sobre requiere un cubo y un lugar disponible y el segundo fluente nos dice que ocupado requiere de un lugar disponible, ambas funciones son esenciales para la solución del problema.

Acciones: Son aquellas que permiten modificar el estatus del mundo en que se desarrolla el proceso. Se declaran de manera similar a los fluentes, por lo que solo pondremos la representación que tienen en nuestro ejemplo.

actions: mover(B,L) requires cubo(B), lugardisponible(L).

Nuestra acción “mover” indica que para poder mover un cubo necesitamos de un cubo y de un lugar disponible donde colocarlo, dada la descripción inicial del problema solo podremos mover un cubo encima de otro cubo o encima de la mesa, dada la descripción inicial del problema la mesa no puede moverse en ninguna forma.

Condiciones de ejecución. Se declaran de la siguiente forma:

executable a if b1, ..., bm, not bm+1, ..., not bn

Causation rules (reglas que describen las causas de las acciones). Son aquellas que son provocadas por las acciones válidas en nuestro mundo y que deben reflejar el nuevo estado de nuestro mundo, las cuales se declaran de la siguiente forma:

cuased f if b_1, …, b_k, not b_{k+1}, …, not b_l

after a_1, …, am, not a_{m+1}, …, not a_n

En nuestro problema las condiciones y las causas son las siguientes:

executable mover(B,L) if not ocupado(B), not ocupado(L), B <> L.

inertial sobre(B,L).

caused ocupado(B) if sobre(B1,B), cubo(B).

caused sobre(B,L) after mover(B,L).

caused -sobre(B,L1) after mover(B,L), sobre(B,L1), L <> L1.

La condición de ejecución para mover B de tipo bloque a L de tipo lugar se realizará si B, nuestro bloque no se encuentra ocupado, es decir, que no haya otro bloque encima de este, y que el lugar a donde deseamos mover tampoco se encuentre ocupado. Esto causara que ese bloque se encuentre ocupado. La segunda causa es que ahora nuestro cubo se encuentre sobre un lugar diferente. La tercera causa es que nuestro bloque ya no se encuentre en la posición donde antes estaba lo cual se puede escribir con el símbolo de negación "-". De esta manera podemos observar que cada acción que ocurra en nuestro mundo tiene una causa la cual modifica los estados de mundo.

Situación inicial. Es el estado en el que encontramos por primera vez a nuestros elementos del problema. En nuestro ejemplo se describen de la siguiente manera:

initially: sobre(a,mesa). sobre(c,mesa). sobre(b,mesa).

Lo que nos dice que inicialmente a, c y b se encuentran sobre la mesa. Como se mostró en la Figura 4.2.

Situación final o meta. Es el objetivo al que se llega después de realizar todas las especificaciones dadas. En nuestro ejemplo se describen de la siguiente manera:

goal: sobre(a,b),sobre(b,c),sobre(c,mesa) ? (2)

La meta indica la forma en la que los bloques quedarán apilados en este caso nos dice que a quedará sobre b, b quedará sobre c y c quedará sobre la mesa. Tal y como se mostró en la

Figura 4.3. El número 2 que observamos después del signo de interrogación nos dice el número de pasos que deben realizarse para dar solución al problema especificado, en este caso hemos puesto 2, ya que si ponemos menos el programa puede quedarse a la mitad y si ponemos más, la salida en pantalla nos dirá "*no actions"* lo que quiere decir que aunque parezca que ocurrió una acción realmente el programa no hizo nada y nos seguirá dando el mismo plan con solo 2 pasos.

A continuación, se muestra la ejecución del programa en DLV^k: Como podemos observar es muy sencillo ejecutar un programa en este sistema, solo debemos tener correctamente escritas nuestras especificaciones y debemos tener en cuenta que el orden de ejecución de los archivos es importante, hablaremos más delante de esto.

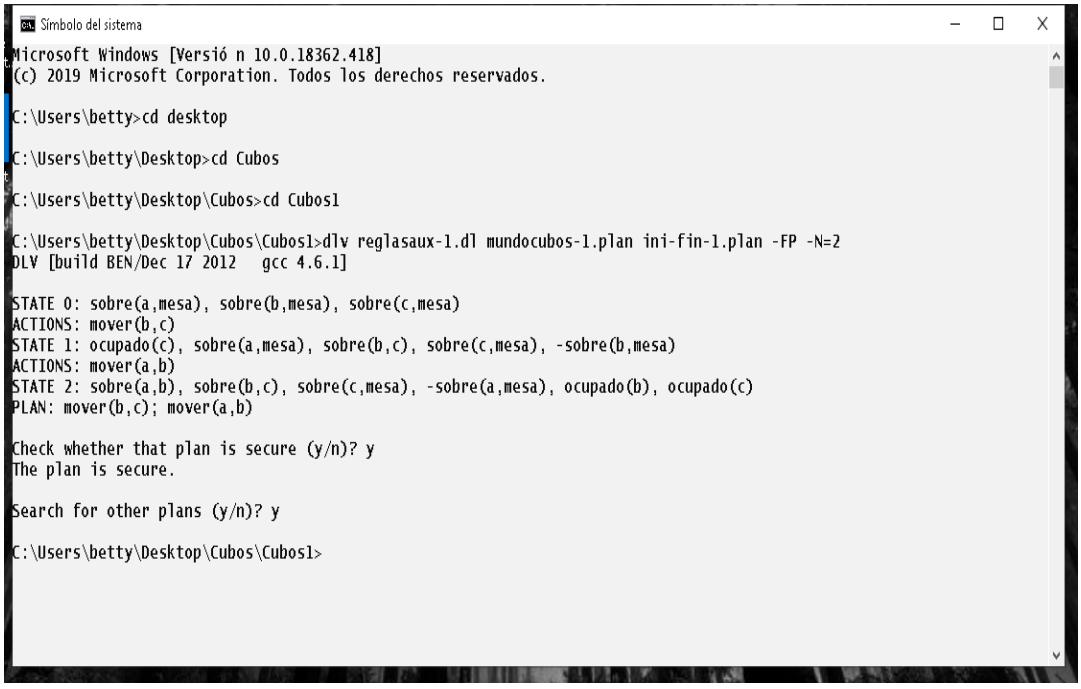

Figura 4.4 Ejecución del ejemplo "El mundo de los bloques"

La Figura 4.4 nos muestra que el sistema nos dio una solución. Después de los comandos ejecutados podemos observar que el *"STATE 0"* es la situación inicial de nuestros bloques, después de este vienen las acciones *"ACTIONS"* las cuales nos dicen paso a paso cuales fueron los movimientos realizados, podemos observar la primer acción fue mover el bloque "b" a "c", después podemos observar el "STATE 1" o bien el siguiente estado es cómo quedo

después de realizada esa acción, y describe perfectamente cómo es que ahora el bloque c se encuentra ocupado ya que el bloque b ahora se encuentra sobre c. También podemos observar cómo es que el cubo a sigue sobre la mesa y que el bloque b ya no se encuentra en el lugar inicial. La siguiente acción es mover el bloque a sobre b. El siguiente estado nos dice cómo es que se encuentran los cubos, a sobre b, b sobre c y c sobre la mesa. Observemos como los estados de los bloques también han cambiado pues ahora nos dice que a ya no se encuentra sobre la mesa, y que b y c ya se encuentran ocupados, pues ya hay sobre ellos un bloque. Finalmente tenemos el *"PLAN"* que es realmente lo que buscamos, el conjunto de pasos que debemos realizar para dar solución a nuestro problema, en este caso nos dice que primero debemos mover el bloque b sobre el bloque c y luego debemos mover el bloque a sobre b. De donde podemos darnos cuenta de que ese plan resuelve el problema con 3 bloques.

Como podemos observar el sistema DLV^k después de darnos el plan nos da la oportunidad de decirle si el plan arrojado es seguro y si queremos buscar otro plan que dé solución al problema. En este caso solo tenemos una solución dadas las especificaciones que tiene el programa y aunque nosotros le digamos al programa que, si queremos buscar otro plan, este no nos dará nada puesto que como ya se mencionó por las especificaciones dadas inicialmente solo tendremos una solución. Para corroborar que el plan es el correcto podemos hacer uso de el en una prueba de escritorio a mano y darnos cuenta de que es correcta.

Ejemplo 4.1.2 El mundo de los bloques (con brazo)

Como vimos en la descripción anterior del problema del mundo de los bloques, hicimos la suposición de que los bloques se movían por si solos, en esta versión agregamos un brazo que es ahora el que mueve a los bloques. Enfatizando que es muy fácil agregar elementos a los programas en DLV^k. Dentro de este programa tenemos los mismos elementos: Fluentes, acciones, condiciones de ejecución, reglas que describen las causas de las acciones *"Causation rules",* situación inicial y la situación final o meta.

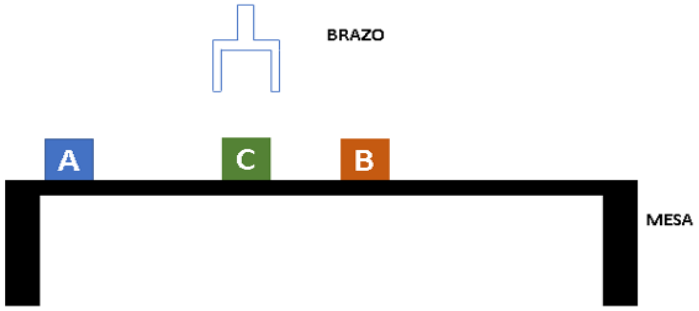

Figura 4.5 Situación inicial con brazo

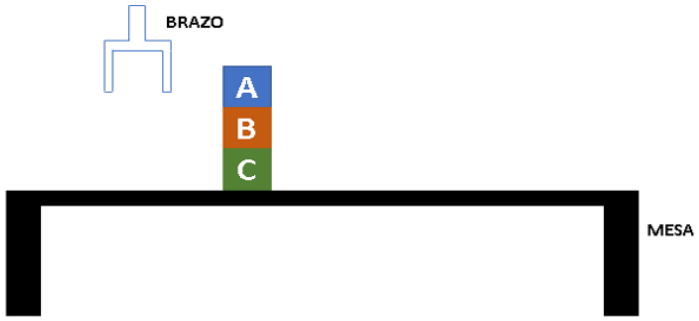

Figura 4.6 Situación final o meta

Las figuras 4.4 y 4.5 nos muestran como es que la situación inicial y final es completamente la misma que la del ejemplo 4.1, esta vez nos encontramos con el aumento del brazo.

Para fines prácticos solo mostraremos la salida del programa, ya que podremos darnos cuenta de que agregar otro elemento es muy sencillo.

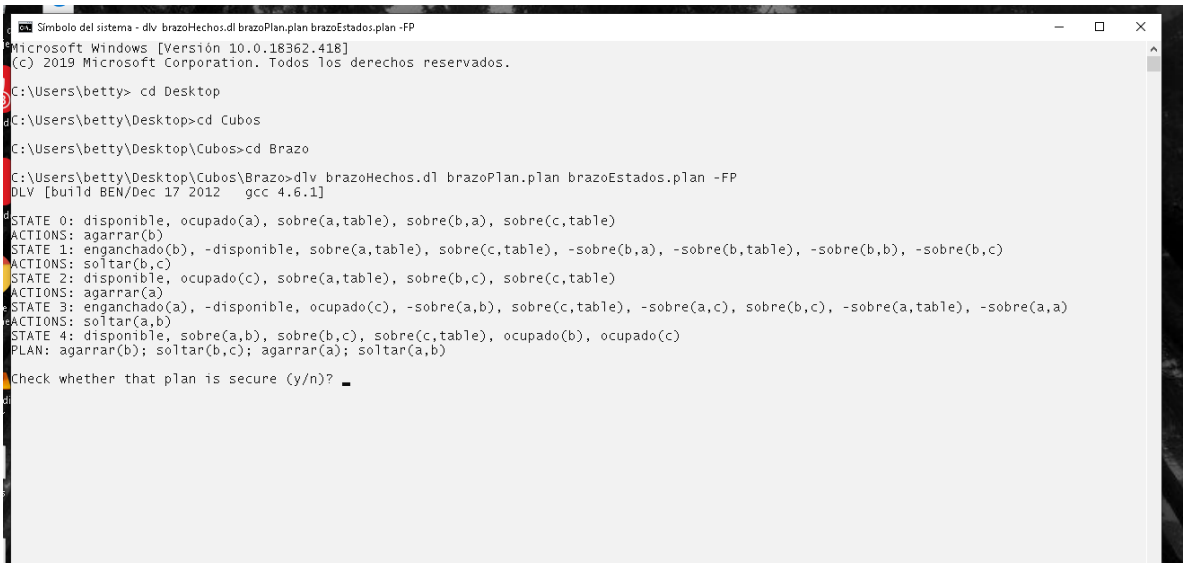

Figura 4.7 Salida del programa en DLV[k] con el brazo

La Figura 4.7 nos muestra la salida los estados y las acciones realizadas para la solución del problema del mundo de los bloques con brazo. Como en el ejemplo anterior el “STATE 0” y no solo aplica para estos casos sino para todos en general, nos muestra el estado inicial de los cubos y del brazo. Ahora describiremos el “STATE 4” el cual nos dice que el brazo se encuentra disponible, el cubo “a” ya se encuentra encima de “b”, “b” ya se encuentra sobre “c”, y “c” ya se encuentra sobre la mesa y nos debemos dar cuenta cómo es que los estados de los cubos que ya tienen un cubo encima de ellos ha cambiado, tal es el caso de “b” y “c”.

El sistema en este ejemplo también nos va mostrando paso a paso cuando el brazo se encuentra disponible y cuando se encuentra ocupado. Debemos tener en cuenta que no basta con simplemente agregar elementos, sino que estos tengan sus acciones y consecuencias de manera correcta, podemos observar que para el brazo se agregaron acciones que permiten en movimiento de los bloques.

Así como en el ejemplo anterior el programa nos pregunta si el plan es seguro y si queremos buscar otro, también en esta versión solo tenemos un único plan dadas las especificaciones del programa.

4.4 Caso de estudio: Estacionamientos tipo pila de alta densidad

Vista la sintaxis en el ejemplo anterior sin más preámbulo entraremos a nuestro caso de estudio. Como vimos en el capítulo uno, los centros históricos de ciudades como la nuestra enfrentan problemas con los estacionamientos existentes debido al espacio tan reducido para atender de manera eficiente a sus clientes. Los centros históricos de las ciudades, pese a la tendencia descentralizadora normalmente establecida en los planes de desenvolvimiento de sus áreas urbanas, constituyen siempre puntos inevitables de gran concentración de tráfico, originado tanto por la gran densidad de habitantes que tradicionalmente trabajan en ellas, como por las actividades allí implantadas y por las innumerables personas que diariamente se trasladan a estos centros. Debido a estas concentraciones de tráfico, además de las innumerables situaciones de congestión que provocan, y básicamente en las horas pico, crean problemas de estacionamiento bastante graves. Por esta razón, es necesario la adopción de medidas, muchas veces drásticas, por parte de las entidades municipales.

En este trabajo de tesis se propone resolver el problema de planeación desde una nueva perspectiva basada en la programación lógica. Nuestro caso de estudio como ya se mencionó representa el modelado de un estacionamiento de tipo pila o *"lifo" (last in first out)* el cual contará con las acciones básicas realizadas en un estacionamiento como el de los centros históricos que son: meter y sacar un coche. En nuestro caso de estudio se supondrá que como tradicionalmente en los estacionamientos de los centros históricos contamos con alguien que es el encargado de realizar dichas acciones. Para la realización de las acciones haremos uso de la calle que es como comúnmente se realizan, para poder mover un coche y de esta manera sacar el coche solicitado. Existen varios elementos que fueron tomados en cuenta para modelar este caso de estudio.

En la Figura 4.8 podemos observar la situación inicial de estacionamiento. Los coches ya se encuentran dentro, contamos con una pared o fondo el cual nos ayudará a dar una especie de tope para que los coches en el modelado del estacionamiento no se sigan de largo. Podemos observar la calle que como ya se había mencionado nos ayudara a la realización de las actividades "meter y sacar". La Figura 4.9 nos muestra la situación final o meta después de realizadas todas las acciones necesarias para poder sacar un coche.

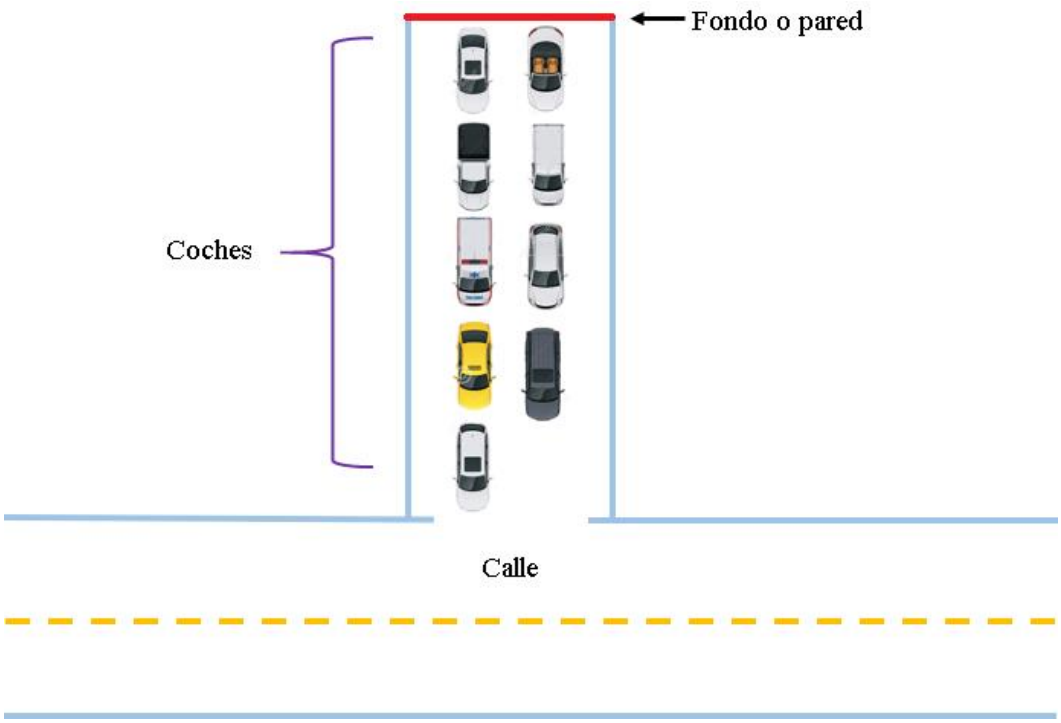

Figura 4.8 Situación inicial estacionamiento

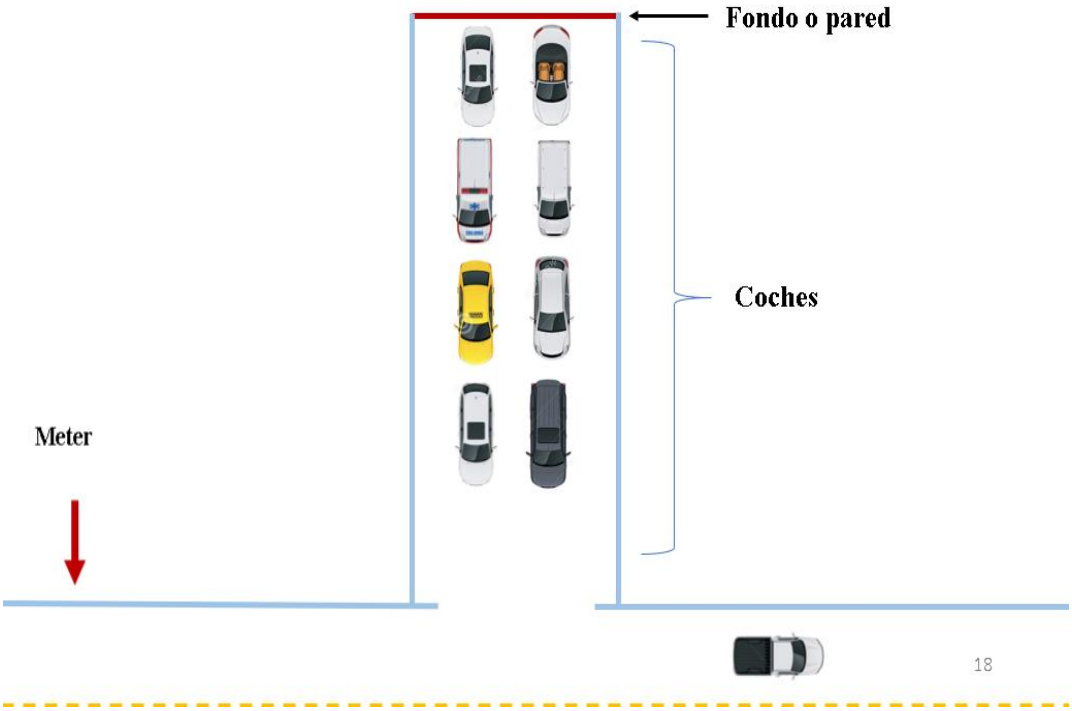

Figura 4.9 Situación final o meta

Para poder ejecutar un programa en DLV^k se deben tener tres archivos, en nuestro caso de estudio el primer archivo es llamado “estacionamiento.dl” en el cual se declaran las variables que utilizamos, como lo podemos observar en la Figura 4.10:

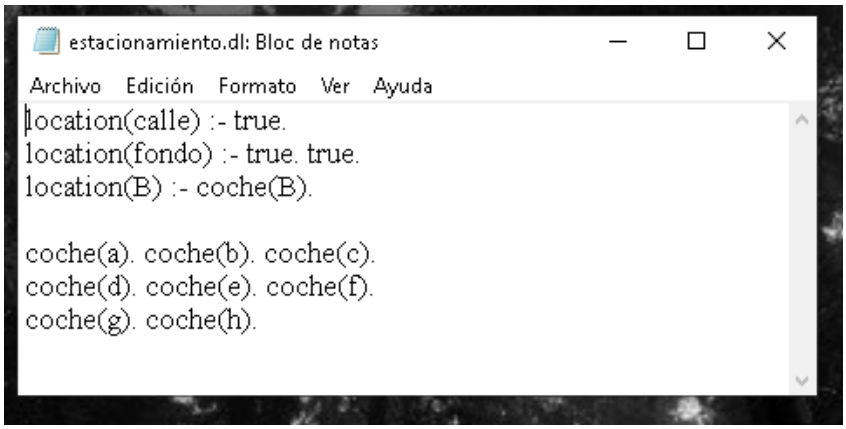

Figura 4.10 Variables utilizadas

La figura 4.10 muestra las variables utilizadas en nuestro caso de estudio, como podemos observar necesitamos de varios lugares, en este caso la calle, un fondo o pared que para nuestro caso nos sirve para entender que el estacionamiento no es infinito, también necesitamos de un cajón donde se encontrarán estacionados los coches.

El segundo archivo es llamado “estacionamiento.plan”. En este archivo declaramos nuestra situación inicial y la meta tal y como se observa en la Figura 4.8 y 4.9.

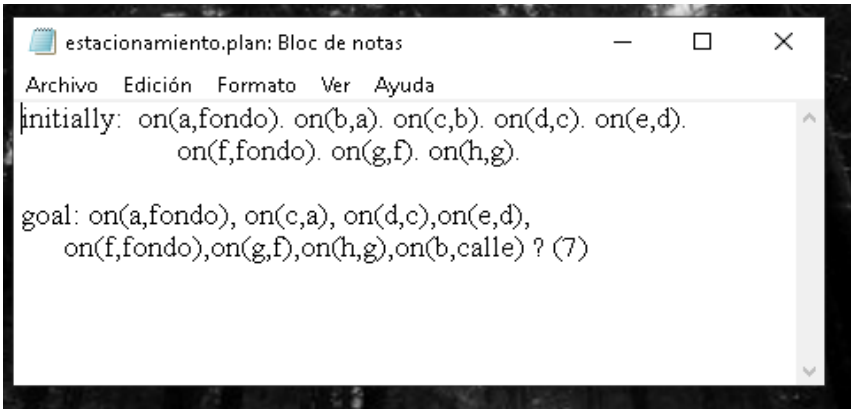

Figura 4.11 Situación inicial y meta en DLV^k

La figura 4.11 nos muestra cómo es que se declaró la situación inicial y meta en DLV^k. Observemos que inicialmente los coches se encuentran como en la Figura 4.8 y que la meta muestra cómo es que hemos sacado un coche del estacionamiento tal y como se muestra en la Figura 4.9.

El tercer archivo que necesitamos es llamado “estacionamiento_world.plan” en el se encuentran todas las reglas, acciones y causas que nuestro programa necesitara para que el sistema nos dé el plan óptimo para resolverlo.

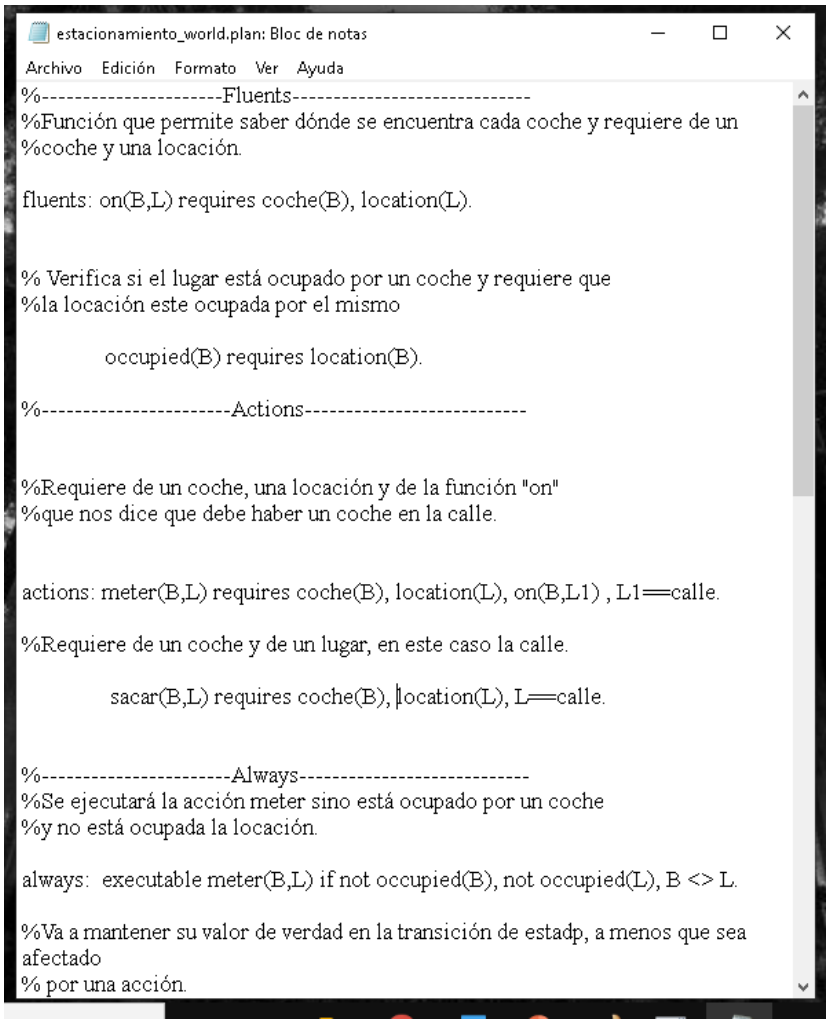

Figura 4.12 Elementos necesarios en el estacionamiento

La Figura 4.12 contiene los fluentes, las acciones y las causas de realizar las acciones básicas del estacionamiento que como ya lo habíamos visto son: meter y sacar un coche. Este archivo es el más importante, pues aquí hemos escrito todas las reglas que ha resuelto el problema.

Como mencionamos en capítulos anteriores estas reglas deben ser segura, de esta manera obtendremos un plan seguro y por consiguiente tendremos un plan seguro.

En nuestra simulación después de la situación inicial podemos observar la salida del ultimo coche para poder sacar el coche señalado.

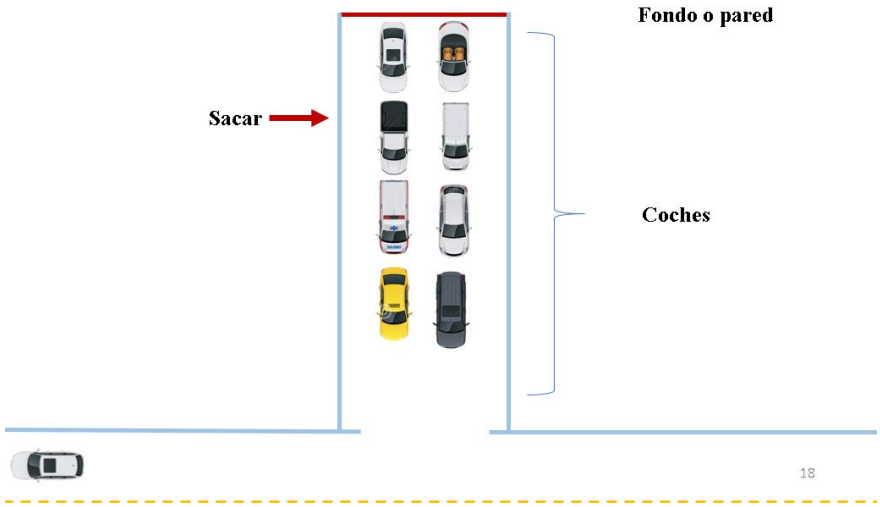

Figura 4.13 Salida del último auto de la primera fila

Cuando los coches empiecen a salir de esta manera se respetará el principio de una pila de tipo "*lifo*". La Figura 4.13 nos muestra cómo es que se coloca el coche en la calle para poder realizar las siguientes acciones y así sacar el coche señalado.

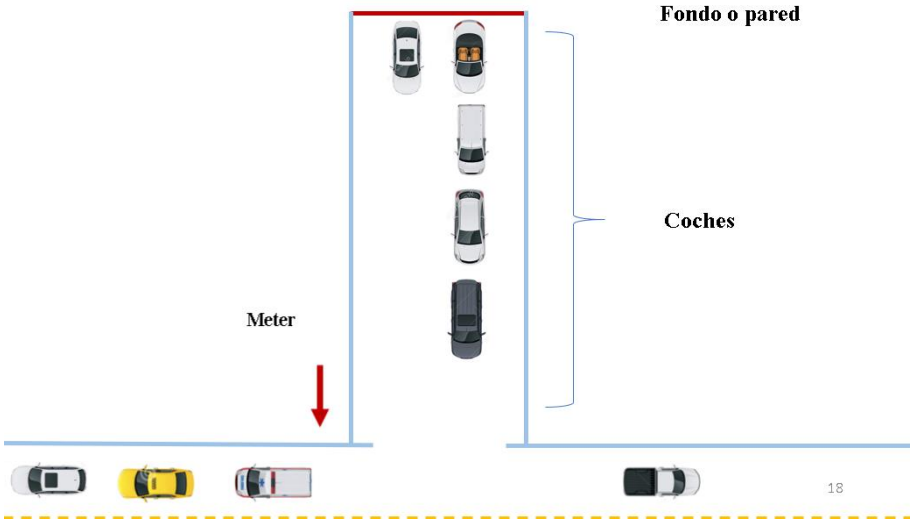

Figura 4.14 Salida del coche señalado

La Figura 4.14 muestra cómo es que se ven todos los coches en la calle para que el coche señalado pudiera salir. Después veremos que, así como salieron los coches deben entrar, de esta forma cumplimos con la característica de la estructura “pila”. Y así poder llegar a la situación final o meta que muestra la Figura 4.9. Dónde el coche seleccionado ha salido y los demás coches ya se encuentran estacionados nuevamente.

Para poder ejecutar los archivos en el CMD el orden es importante ya que si los colocamos de manera incorrecta la consola nos dejará esperando sin resultado alguno. En este caso el orden y comando de ejecución es el siguiente:

dlv estacionamiento.dl estacionamiento_world.plan estacionamiento.plan -FP

A continuación, se muestra salida en pantalla de la ejecución de nuestros archivos.

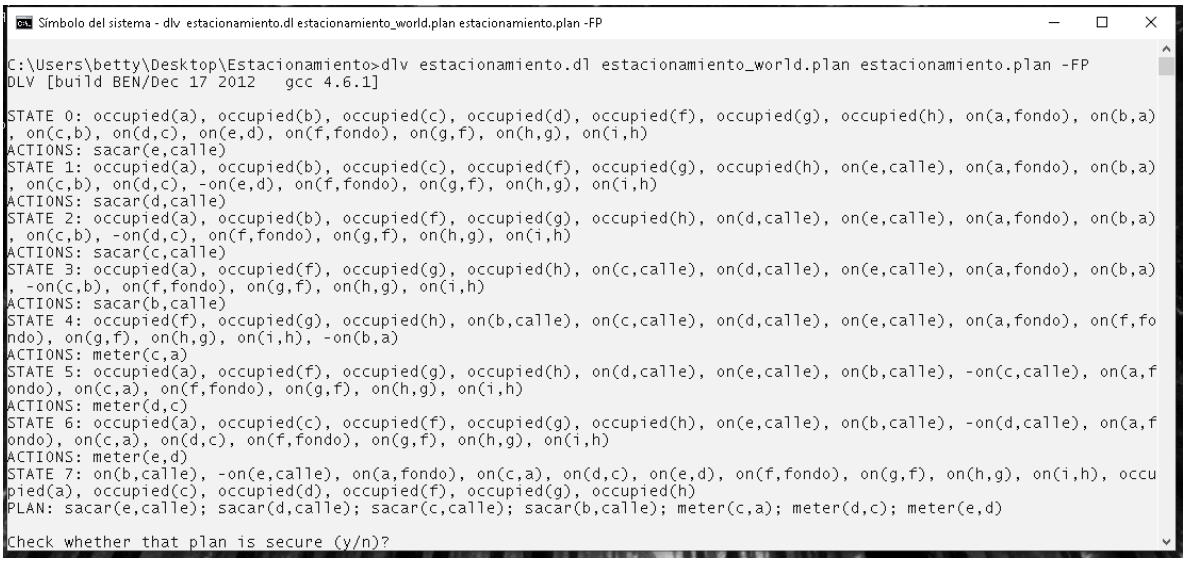

Figura 4.15

Como podemos observar al igual que en el ejemplo de “el mundo de los bloques” el sistema nos da el “*state 0”* en el cual podemos observar la situación inicial, se puede notar que los coches que tienen un coche detrás de ellos se marcaron como ocupado, para que la codificación del problema fuera más clara y sencilla, después podemos ver la forma en la que se encuentran estacionados, para facilitar la escritura del programa los autos son identificados con una letra diferente. La primera acción para sacar el coche seleccionado en la Figura 4.13 es sacar el último coche de esa fila que en nuestro programa en DLV^k es el coche con la letra e. Para esto debemos preguntar si ese coche se encuentra ocupado, al no estarlo podemos

sacar el coche. La siguiente acción es sacar el coche d a la calle, de la misma manera que la acción pasada preguntamos si el coche se encuentra ocupado al ya no estar el coche e detrás del coche d podemos realizar la acción, noten como es que cuando un coche sale, su estado también se modifica pues en este caso el coche d ya no se encuentra detrás del coche c por lo que su estado es "-on(d,c)". La siguiente acción es sacar el coche c, como el coche d ya no se encuentra detrás de este, puede salir. También cambia el estado y ahora es "-on(c,b)". Ahora vamos a sacar el coche seleccionado que es el b, para esto al igual que con los otros coches se verifica que no haya otro coche detrás de este, cuando el coche sale a la calle, cambia su estado y ahora es "-on(b,a)". Hasta este momento se han realizado las acciones de forma correcta pues el coche seleccionado ha salido. Ahora es momento de volver a estacionar los coches para esto se utiliza la acción meter. En el *"state 5"* después de aplicada la acción de meter al coche c, podemos observar cómo nuevamente cambian los estados de los coches, ahora el coche c ya se encuentra detrás del coche a y su estado cambia, ahora ya es "on(c,a)". La siguiente acción es meter el coche d y se realiza exactamente igual, se puede observar que también su estado cambia al ser estacionado, ahora ya es "on(d,c)". La última acción es meter el coche e por lo que se sigue la misma dinámica y al ingresar cambia su estado a "on(e,d)". De esta manera han ingresado todos los coches que fueron sacados para sacar el coche señalado "b". Al final de las acciones podemos observar que DLV^k nos da el plan que debemos seguir para dar solución al problema. El cual cómo podemos observar es correcto.

Dadas las especificaciones de nuestro programa la solución para este problema es única por lo tanto el sistema nos seguirá dando la misma solución cada vez que lo ejecutemos.

Trabajo a futuro

La Benemérita Universidad Autónoma de Puebla en el área de medicina cuenta con un estacionamiento automatizado para guardar vehículos de manera segura y sobre todo funcional [27]. Nuestro objetivo es llevar a la realidad el modelado del estacionamiento ya que este optimizaría muchos establecimientos de centros históricos. Mediante esto también se pretende desarrollar un sistema novedoso y vanguardista que emplea la inteligencia de negocios en apoyo a la toma de decisiones.

Conclusiones

Como podemos observar demostramos de manera exitosa a través de uno de los problemas típicos de planeación conocido como el "mundo de los bloques" la representación de un mundo a través del lenguaje DLV^k. Mostramos la documentación detallada de la forma en la que se debe modelar un problema de planeación de este estilo. Pudimos llegar al objetivo planteado inicialmente que fue el modelado exitoso del estacionamiento de tipo pila a través de un lenguaje disyuntivo lógico en DLV^k. A través de este trabajo de tesis pudimos detonar el desarrollo de sistemas inteligentes incorporando todo el potencial que las nuevas tecnologías web y móviles ofrecen, tales como ser usado en cualquier lugar, en todo momento y con información en tiempo real.

Y como un resultado sobresaliente se aceptó la publicación de un artículo de planeación basado en este presente trabajo de tesis en colaboración el cual lleva por nombre "Modeling and planning through answer set programming" e*n "International Journal of Engineering and Applied Sciences"*. Con lo cual podemos recalcar que se ha puesto un enorme hincapié para resolver problemas de este estilo.

Bibliografía

[1] Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., & Wilkins, D. PDDL — The Planning Domain Definition language (Tech. Rep.). Yale Center for Computational Vision and Control. (Available at http://www.cs.yale.edu/pub/ mcdermott/software/pddl.tar.gz), 1998.

[2] Niemelä, Ilkka & Simons, Patrik. SMODELS – an implementation of the stable model and well-founded semantics for normal logic programs. 10.1007/3-540-63255-7_32. 2006.

[3] Eiter T., Faber W., Leone N., Pfeifer G., Polleres A. System Description: The DLVK Planning System. In: Eiter T., Faber W., Truszczyński M.. (eds) Logic Programming and Nonmotonic Reasoning. LPNMR 2001. Lecture Notes in Computer Science, vol 2173. Springer, Berlin, Heidelberg. 2001.

[4] Eiter T., Faber W., Leone N., Pfeifer G., Polleres A. System Description: The DLVK Planning System. In: Eiter T., Faber W., Truszczyński M.. (eds) Logic Programming and Nonmotonic Reasoning. LPNMR 2001. Lecture Notes in Computer Science, vol 2173. Springer, Berlin, Heidelberg. 2001.

[5] Hector Geffner and Blai Bonet. A concise introduction to models and

methods for automated planning. Synthesis Lectures on Artificial Intelligence and Machine Learning, 8(1):1–141, 2013.

[6] Eiter, Thomas & Faber, Wolfgang & Pfeifer, Gerald & Polleres, Axel. (2004). Declarative Planning and Knowledge Representation in an Action Language. 10.4018/9781591404507.ch001.

[7] – Answer Set Programming. https://en.wikipedia.org/wiki/Answer_set_programming

[8] – Pedro Cabalar, David Pearce y Agustín Valverde, 2009

[9] – Wiley Encyclopedia of Computer Science and Engineering, edited by Benjamin Wah. Copyright # 2008 John Wiley & Sons, Inc

[10] – Garcia-Mata, Carmen & Marquez, Pedro. (2009). Answer Set Programming y el Problema de Asignación de Frecuencia, FAP.

[11] – J. McCarthy, Programs with common sense, Proc. Teddington Conference on the Mechanization of Thought Processes, Her Majesty's Stationery Office, 1959.

[12] – Mendelson, Elliott (1997), Introduction to Mathematical Logic (4th ed.), London: Chapman & Hall, ISBN 978-0-412-80830-2.

[13] – T. Eiter and M. Fink and G. Sabattini and H. Thompits, Considerations on

Updates of Logic Programs, Proceedings of the seventh European Workshop on Logic in Artifitial Intelligence (JELIA 00), 2000, M.O. Aciego and I.P. de Guzman and G. Brewka and L. M. Pereira, Springer LNAI, Vol. 1, 1919.

[14] M. Osorio and J.A. Navarro and J. Arrazola, Applications of Intuitionistic Logic in Answer Set Programming (Extended version), TPLP 2003.

[15] Sierra Rodríguez, P. (2018). *Lógica minimal, intuicionista y clásica.* [Ebook]. Sevilla: Patricia Sierra Rodríguez. Retrieved from https://idus.us.es/xmlui/bitstream/handle/11441/77581/Sierra%20Rodr%C3%ADguez%20Patricia%20TFG.pdf?sequence=1&isAllowed=y

[16] – DLVSystem Spin-Off Of University of Calabria http://www.dlvsystem.com/

[17] – Leone, Nicola & Pfeifer, Gerald & Faber, Wolfgang & Calimeri, Francesco & Dell'Armi, Tina & Eiter, Thomas & Gottlob, Georg & Ianni, Giovambattista & Ielpa, Giuseppe & Koch, Christoph & Perri, Simona & Polleres, Axel. (2002). The DLV system. 537-540. 10.1007/3-540-45757-7_50.

[18] – Citrigno, Simona & Eiter, Thomas & Faber, Wolfgang & Gottlob, Georg & Koch, Christoph & Leone, Nicola & Mateis, Cristinel & Pfeifer, Gerald & Scarcello, Francesco. (2004). The dlv System: Model Generator and Application Frontends. Proceedings of the 12th Workshop on Logic Programming.

[19] – Manual de usuario de DLV en línea http://www.dlvsystem.com/html/DLV_User_Manual.html

[20] – Leone, Nicola & Pfeifer, Gerald & Faber, Wolfgang & Perri, Simona & Alviano, Mario & Terracina, Giorgio. (2005). The Disjunctive Datalog System DLV*. Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy. [21] – M. Heidt. Developing an Inference Engine for ASET-Prolog. Master tesis University of Texas El Paso, 2001.

[22] – T. Eiter and W. Faber and N. Leone and G. Pfeifer and A. Polleres, A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity, Institut für Informationssysteme, Abteilung Wissensbasierte Systeme, INFSYS Reserch Report 184301-December 2001; October 2002.

[23] – Wikipedia
https://es.wikipedia.org/wiki/Planeamiento

[24] – Héctor Luis Palacios Verdes. (2009). Translation-based approaches to Conformant Planning (Tesis Doctoral UPF). Universitat Pompeu Fabra, Barcelona, España.

[25] – Eiter, Thomas & Faber, Wolfgang & Leone, Nicola & Pfeifer, Gerald & Polleres, Axel. (2000). Planning under Incomplete Knowledge. Proceedings of the First International Conference on Computational Logic. 10.1007/3-540-44957-4_54.

[26] – Eiter, Thomas & Faber, Wolfgang & Leone, Nicola & Pfeifer, Gerald & Polleres, Axel. (2002). A Logic Programming Approach to Knowledge-State Planning, II: The DLV System. Artificial Intelligence. 144. 10.1016/S0004-3702(02)00367-3.

[27] – BUAP – Estacionamiento robotizado
http://www.estacionamientorobotizado.buap.mx/